Université de Nancy I — U.E.R Sciences Mathématiques

Centre de Recherche en Informatique de Nancy

THÈSE

pour l'obtention du

Doctorat de l'Université de Nancy I

(Spécialité Informatique)

par

Noureddine LAZRAK

Contribution à la Vérification de Spécifications Algébriques

Application à Certaines Propriétés des Programmes Parallèles

Présentée et soutenue le 2 Fevrier 1990

Devant la commission composée de :

Président	Pierre LESCANNE
Rapporteur	Michaël RUSINOWITCH
	Joseph SIFAKIS
Examinateur	Jean-Pierre FINANCE
	Pascal GRIBOMONT
	Guy-René PERRIN

Université de Nancy I U.E.R Sciences Mathématiques

Centre de Recherche en Informatique de Nancy

THÈSE

pour l'obtention du

Doctorat de l'Université de Nancy I

(Spécialité Informatique)

par

Noureddine LAZRAK

Contribution à la Vérification de Spécifications Algébriques

Application à Certaines Propriétés des Programmes Parallèles

Présentée et soutenue le 2 Fevrier 1990

Devant la commission composée de :

Président	Pierre LESCANNE
Rapporteur	Michaël RUSINOWITCH
	Joseph SIFAKIS
Examinateur	Jean-Pierre FINANCE
	Pascal GRIBOMONT
	Guy-René PERRIN

A travers ces lignes, je souhaiterai remercier les personnes qui ont contribué à la réussite de ce travail:

Guy-Renè PERRIN, Professeur à l'Université de besançon, qui a dérigé cette thèse. Sa compétence, ses suggestions construcives et ses encouragements m'ont été très precieux tout au long de ce travail. Je lui serai reconnaissant.

Jean-Pierre FINANCE, Professeur à l'Université de Nancy 1 et Directeur du C.R.I.N. Il a su nous communiquer depuis la Licence une méthodologie de travail et un regard critique pour résourdre les problèmes informatiques. Je le remércie sincèrement pour la patience avec laquelle il a lu ce document ainsi que pour ces remarques pertinentes.

Michaël RUSINOWITCH, Maître de conférence à l'Université de Nancy 1, qui a consacré beaucoup de temps pour être rapporteur de cette thèse. Il a été un lecteur critique, qui a su mettre l'accent sur les points clés concernant la théorie de la preuve automatique.

Joseph SIFAKIS, Directeur de recherche CNRS au Laboratoire de Genie Informatique de Grenoble, a accepté d'être rapporteur de ce mémoire. Je le remercie sincèrement pour l'enthousiasme avec lequel il m'a fait part de ses remarques.

Pascal GRIBOMONT, Directeur de recherche à PHILIPS Research Laboratory de Bruxelles. Je le remercie pour l'intérêt qu'il a temoigné pour ce travail.

Pierre LESCANNE, Directeur de recherche CNRS au C.R.I.N., a été président de ce jury. Je le remercie pour les suggestions qu'il m'a faites.

J'adresse mes sincères remerciements à Omar FASSI, Abderafiaa KOUKAM et Azeddine LAZREK qui m'ont donné beaucoup de leur temps et qui n'ont jamais cessé de me soutenir. Je remercie Nibal IDLEBI, Dominique MERY, Honoré TCHUENKAM, ainsi que tous les membres de l'équipe COMETE qui ont su écouter mes exposés et qui ont participé à l'avancement de ce travail avec leurs remarques et leurs questions. Je n'oublie pas de remercier aussi tous les amis pour leur fraternel appui.

Enfin je dédie ce travail à mes parents, à mon frère et mes sœurs à qui je serai éternellement reconnaissant pour leurs soutiens et leurs encouragements.

Verification of algebraic specifications
Application to some properties of parallel programs

Abstract

Our aim in this thesis is to prove the validity of equational formulas in an algebraic specification axiomatized by a set of equations. An equational formula is universally quantified formula of a first order language whose atomic formulas are simple equations.

We propose to extend the classical methods used in simple equations case, in order to prove the validity of equational formulas. So, to prove the inductive validity of an equational formula we use structural induction and we propose an automatic method in order to generate induction schemas in some particular cases.
Our method for proving equational validity of an equational formula is based on transforming and decomposing it into a set of simple equations such that the equational validity of those equations is equivalent to the equational validity of the initial formula. We could then use rewriting techniques.

We also propose transitive chaining in order to prove validity of equations containing ordering relations.

Theoretical studies of the principles proposed in this thesis have leaded to the theorem prover PAUSE which is used in order to prove some asynchronous relations between components which deal with communications in parallel programs

Résumé

Notre objectif dans cette thèse est de vérifier la validité d'une *formule équationnelle* dans une spécification algébrique axiomatisée par un ensemble d'équations.

Une formule équationnelle est une formule universellement quantifiée d'un langage du premier ordre dont les formules atomiques sont des équations simples.

Nous proposons de généraliser et d'adapter les méthodes utilisées dans le cas des équations simples pour prouver la validité des formules équationnelles. Ainsi, dans le cas de la validité inductive nous avons utilisé l'induction structurelle basée sur la récurrence classique. Nous avons proposé une méthode pour la génération automatique des schémas d'induction dans certains cas particuliers. Dans le cas de la validité équationnelle nous avons proposé une méthode basée sur la transformation et la décomposition de la formule initiale pour déterminer un ensemble représentatif d'équations simples. La validité équationnelle de ces équations est équivalente à celle de la formule initiale.

Un autre mécanisme de raisonnement appelé le chaînage transitif est proposé pour tester la validité des équations contenant des relations d'ordre.

L'étude théorique des principes proposés dans cette thèse a abouti à la réalisation du système de preuve *PAUSE* que nous avons utilisé pour vérifier certaines relations d'asynchronisme entre les composantes qui gèrent la communication dans un système parallèle.

Mots-clés :
Preuve automatique, formule équationnelle, induction structurelle, décomposition, réécriture, raisonnement par cas, chaînage transitif, communication.

تصحيح المواصفات الجبرية

و تطبيقها علي بعض خاصيات البرامج المتوازية

أمام تزايد أهمية الوظائف المناطة بالإعلاميات صار من المفروض إيجاد طرق للتأكد من صحة البرامج. لكن شكل هذه البرامج و تعقيدها أدى الى اهتمام أكثر بمواصفاتها لتأكد من صحتها. إذ أن هذه المواصفات قريبة من المستعمل، يسهل عليه تركيبها و فهمها. و هي في نفس الوقت ذات شكل و دقة، يستطيع الحاسوب فهمها و جعلها كأدات للبرهنة. في هذا الميدان، اهتممنا في هذه الرسالة بإثبات صحة بعض خاصيات المواصفات الجبرية.

تتكون هذه المواصفات الجبرية من مجموعة من المعادلات الشرطية أو اللا شرطية. أما خاصياتها فهي عبارة عن صيغات معادلاتية عامية التكميم. ولإثبات صحة هذه الصيغات المعادلاتية بالنسبة لمواصفة جبرية معينة، اعتمدنا على طريقتين تتخصص كل واحدة منهما بمجموعة من الصيغات.
ففيما يخص البرهنة التراجعية، اعتمدنا على التراجع البنيوي وقدمنا طريقة لتركيب المخططات التراجعية أوتوماتكيا. اما فيما يخص البرهنة المعادلاتية، ولكي نتمكن من استعمال أساليب البرهنة الخاصة بالمعادلات السهلة، قدمنا طريقة تعتمد على تحويل و تجزئة الصيغات المعادلاتية الى مجموعة من المعادلات السهلة حيث تكون صحة هذه المجموعة مكافئة لصحة الصيغة الأولى.
ومن جانب اخر وظفنا طريقة سميناها التسلسل المتعدي لإتباث صحة المعادلات التي تتظمن علاقات ترتيبية.
وفي الأخير قدمنا المبرهن الأوتوماتيكي (PAUSE) كنتيجة عملية للدراسات النظرية للمباذئ التي طرحناها مسبقا. وقد تم استعماله لإثبات صحة بعض خاصيات البرامج المتوازية.

مواصفة جبرية	Specification algebrique
معادلة شرطية	Equation conditionnelle
صيغة معادلاتية	Formule equationnelle
عامية التكميم	Universellement quantifiee
البرهنة التراجعية	Preuve par induction
التراجع البنيوي	Induction structurelle
المخططات التراجعية	Schema d'induction
البرهنة المعادلاتية	Preuve equationnelle
التسلسل المتعدي	Chainage transitif
علاقة ترتيبية	Relation d'ordre
برامج متوازية	Programmes paralleles

Table des matières

introduction

1 Contexte de notre travail

Devant l'importance accrue des taches confiées à l'Informatique, disposer de programmes prouvés est désormais une nécessité. De nombreux travaux d'Informatique sont consacrés à cette question. Mais le problème tel qu'il est posé paraît insurmontable: Comment en effet assurer la correction de programmes écrits dans des langages de programmation qui ne sont pas eux même bâtis sur des principes logiques rendant le comportement du programme transparent? Les solutions à ce problème, proposées par le monde de la recherche, peuvent être classées en deux catégories:

- La première consiste à choisir les mathématiques comme des langages de programmation; L'écriture d'un programme satisfaisant certaines spécifications, revient donc à écrire une démonstration d'un énoncé exprimant ces spécifications. La correction d'un programme ne serait plus un problème et pourrait être vérifiée automatiquement. Cependant pour construire un langage de programmation il faut caractériser les démonstrations qui possèdent un contenu algorithmique puis savoir les transformer en de véritables programmes. Ces questions ont été déjà traitées avec succès dans les travaux concernant la logique intuitioniste [Lau70] et le λ-calcul [HS86].

- L'autre catégorie de solutions, la plus fréquente, consiste à construire une "logique des programmes" sur des principes ad hoc décrivant de manière externe leur comportement. L'étude des propriétés des programmes reviendrait donc à l'étude de ces propriétés sur la description de leur comportement. Cependant Le langage de description de comportement doit être suffisament proche du programmeur pour lui faciliter la construction de ses spécifications; il doit être assez formel pour pouvoir établir et vérifier des mécanismes de raisonnement pour étudier les propriétés des spécifications. Ce langage doit être assez rigoureux pour être compris par la machine et permettre l'automatisation des mécanismes de raisonnement.
  Pour répondre à ces exigences, les types abstraits algébriques sont connus par leur facilité d'expression [Fin79], par leur formalismes [GTW78] et par leur adaptation à la mécanisation des raisonnements [Gut78].

Nous avons adopté, nous aussi, cette deuxième solution à côté des innombrables recherches dans ce domaine; et nous allons utiliser ces qualités des types abstraits pour contribuer à la vérification des programmes.

Un type abstrait est une collection d'objets appelée domaine, plus un ensemble d'opérations sur ces domaines. Ces opérations et ces domaines sont indépendants de la représentation. L'algèbre universelle constitue le fondement théorique des types abstraits. En effet, les algèbres et en particulier l'algèbre initiale et l'algèbre libre, sont les domaines d'interprétation des

termes d'un types abstrait.
Une spécification algébrique d'un type abstrait est une algèbre décrite par un ensemble de sortes et un ensemble d'opérations, plus un ensemble d'équations qui décrivent le comportement de ces opérations.
Ces équations sont utilisées pour définir une congruence sur les termes. Ces congruences sont le fondement théorique du remplacement de termes par leurs égaux qui constitue le principal mécanisme de raisonnement dans la logique équationnelle. En plus si ces équations sont orientables, alors elle peuvent définir un système de réécriture qui avec quelques propriétés constitue la pierre de base pour l'automatisation du mécanisme de remplacement d'égaux par des égaux.

Les propriétés à vérifier sur une spécification peuvent être décomposées en deux catégories :

- Des propriétés globales comme l'existence de l'algèbre initiale associée à cette spécification, la complétude ou la consistance de celle ci, ou la terminaison ou la confluence du système de réécriture qui lui est associé.

- Des propriétés élémentaires qui concernent le calcul ou le comportement de certaines opérations.

2 Objectif

Notre objectif dans cette thèse est de répondre à la question : étant données

- un ensemble d'équations qui constituent une axiomatisation d'une spécification algébrique et

- une propriété élémentaire,

est ce que cette propriété élémentaire décrit un comportement correct vis a vis de cette spécification. Autrement dit, est ce que nous pouvons prouver que cette propriété est une conséquence logique de cette spécification; et cela indépendemment des propriétés globales que peut vérifier ou non cette dernière.

Pour répondre à cette question beaucoup de travaux ont été faits dans des cas particuliers de spécifications et de propriétés élémentaires.

Notre contribution consiste à utiliser et à étendre ces travaux pour pouvoir prouver une classe plus grande de propriétés élémentaires dans des spécifications moins restreintes.

2.1 Contexte de la preuve

L'ensemble des équations associées à une spécification est composé de deux catégories :

- des équations inconditionnelles dites également des équations simples,

- des équations conditionnelles de la forme *si-alors* ou de la forme *si-alors-sinon*. Le fait d'introduire cette deuxième forme n'a pas d'influence directe sur la sémantique d'une spécification, mais il permet plus de contrôle au moment de la spécification en définissant les opérations d'une manière exhaustive, il présente aussi un avantage pour l'automatisation de la preuve.

Pour pouvoir modéliser la classe de propriétés élémentaires susceptibles d'être prouvées, nous allons représenter ces propriété par des *formules équationnelles*. Une formule équationnelle est une formule universellement quantifiée d'un langage du premier ordre dont la formule atomique est l'équation simple.

2.2 La preuve de validité

Dans le cadre de la logique équationnelle, une propriété élémentaire est une conséquence logique d'une spécification si et seulement si elle est valide dans l'algèbre initiale associée à cette spécification. Deux catégories de propriétés élémentaires sont à distinguer :

- Les propriétés qui nécessitent une récurrence sur certaines de leurs variables pour être prouvées; nous parlerons alors de la *validité inductive*. Pour prouver cette validité nous utiliserons le principe de l'induction structurelle.

- Les autres propriétés qui peuvent être prouvées par les mécanismes de déduction de base; nous parlerons alors de la *validité équationnelle*.

Or dans le cas des propriétés élémentaires exprimées avec des équations simples, le remplacement d'égaux par des égaux suffisait pour prouver leur validité équationnelle; ce n'est plus le cas pour les formules équationnelles.

Dans [LPF89], nous avons développé les principes de base d'une solution à ce problème.
Notre approche consiste à appliquer sur ces formules une série de transformations jusqu'à obtenir un ensemble d'équations représentatives sur lesquelles nous pouvons appliquer les mêmes mécanismes de raisonnement que dans le cas des équations simples. Pour cela nous proposons de décomposer de telles formules en un ensemble de *a_équations* (équation avec assertions) $\langle\{a_1, \ldots, a_n\}, e\rangle$ tels que la validité équationnelle d'une formule soit équivalente à celle des a_équations associées. Et pour passer de la validité d'une a_équation à celle d'une équation, nous utilisons le théorème suivant qui dit que la validité équationnelle d'une a_équation $\langle\{a_1, \ldots, a_n\}, e\rangle$ dans un ensemble d'équations E est équivalente à celle de e dans $E \cup \{a_1, \ldots, a_n\}$.

Pour automatiser cette approche nous définissons le mécanisme de réécriture sur les a_équations plus le raisonnement par cas en présence des règles de réécriture de la forme *si-alors-sinon*. Mais avant d'ajouter les assertions à l'ensemble des équations initiales, nous essayons de les réduire pour détecter celles qui introduisent des inconsistances, sinon dans le cas contraire utiliser aussi leurs formes normales pour prouver la validité de la partie conclusion e.

Dans cette approche nous avons utilisé un autre mécanisme de déduction, il s'agit du chaînage transitif appliqué en présence d'équations contenant des relations d'ordre. L'utilisation de ce mécanisme est nécessaire puisque la propriété de transitivité perd toute sa puissance quand elle est exprimée sous forme d'équation.

Les études théoriques de ces principes proposés dans cette thèse ont abouti à la réalisation du système de preuve *PAUSE*.

2.3 Application

Ce travail a été réalisé dans le cadre du projet *COMETE* (*COM*munication *Et TE*mps) développé au CRIN à Nancy et au LIB à Besançon et soutenu par le projet CNRS C^3.
Ce projet se préoccupe de méthodologie de construction de programmes parallèles. La notion

de *communication* proposée dans [Per85] constitue l'outil de base de la démarche proposée. Cette démarche consiste à concevoir un système parallèle comme un ensemble de processus indépendants qui communiquent entre eux par des modules intermédiaires appelés *"Types de communication"*. Cette notion de type de communication caractérise la relation qui existe entre les suites des valeurs échangées entre ces processus.
Cet aspect modulaire favorise la modification du système obtenu pour qu'il s'adapte à certaines contraintes. En effet, dans un système et pour un point de communication donné, il est possible de remplacer un type de communication par un autre pour obtenir un système plus "asynchrone" dans le sens où les valeurs produites ne sont pas toutes consommées et les processus consommateurs sont de moins en moins dans des situations d'attente.
Ces modifications sont réalisables chaque fois qu'il est possible de montrer qu'il existe une relation d'ordre "d'asynchronisme" entre le type de communication de départ et celui avec lequel il sera remplacé.
Notre contribution dans ce projet consiste à vérifier certaines propriétés des types de communication et plus particulièrement à prouver ces relations d'asynchronisme.

3 Plan de cette thèse

Après ce bref exposé informel de nos propositions, nous allons donner le plan de cette thèse :

- Dans le premier chapitre nous définissons le cadre formel sur lequel est fondé notre travail, ainsi que les bases théoriques et les notations qui vont servir dans les autres chapitres. Nous rappelons tout d'abord les principes de base des algèbres à plusieurs sortes et l'ensemble des termes associés à une signature. Ensuite, nous traitons le problème de validité dans le cas des équations simples et des équations conditionnelles. Nous terminons ce chapitre par quelques rappels sur le calcul des prédicats du premier ordre, ce qui nous permettra de définir la notion de formule équationnelle, puis de définir sa validité.

- Dans le deuxième chapitre nous décrivons le processus de passage de la validité équationnelle d'une formule à la validité de certaines équations qui la composent. Pour cela nous commençons par établir le principe utilisé souvent dans la logique du premier ordre et qui dit que pour prouver une formule universellement quantifiée, il suffit de fixer ses variables puis d'effectuer la preuve sur la formule résultante. Nous définissons ensuite, la notion de a_équation, puis nous présentons notre stratégie de décomposition en utilisant des règles d'inférence. Ces règles d'inférence nous faciliteront la tâche de preuve de correction de cette stratégie. Ensuite, nous déterminons la classe de formules qui peuvent être traitées par cette stratégie, et par suite qui pourront être prouvées. Nous terminons par le passage de la validité d'une a_équation à celle de sa partie conclusion. Et nous terminons par donner les algorithmes de décomposition.

- Le troisième chapitre traite la validité inductive. Après un rappel sur le principe de preuve par induction sans induction, ses avantages et ses inconvénients, nous introduisons les principes de la méthode que nous avons utilisée et qui est fondée sur l'induction structurelle. Nous introduisons aussi les principes d'une méthode pour la génération automatique des schémas d'induction.

- Le quatrième chapitre est une mise en œuvre, à l'aide des techniques de la réécriture, des résultats obtenus dans le deuxième chapitre. Nous commençons par définir la réécriture

conditionnelle sur les a_équations, puis la stratégie pour tester leur validité. Nous établissons ensuite, la correction de cette stratégie. Nous donnons ensuite, la méthode utilisée pour calculer l'ensemble des formes normales des composantes d'une a_équation. Et nous terminons par donner les algorithmes qui décrivent ces stratégies.

- Le cinquième chapitre concerne le chaînage transitif. Nous commençons par donner les définitions du chaînage transitif simple et conditionnel puis du chaînage transitif en présence de règles de simplification. Nous donnons ensuite, la stratégie de chaînage, puis comment tester la validité des équations contenant des relations d'ordre.

- Le sixième chapitre est consacré au système de preuve $PAUSE$. Nous commençons par une description sommaire de ce système avec ces différentes fonctionnalités et nous terminons par une étude comparative avec les systèmes de preuve existants.

- Le dernier chapitre est une application des stratégies décrites dans les chapitres précédents pour prouver certaines propriétés des systèmes parallèles. Nous commençons par définir les types de communication qui sont l'outil de base du langage d'expression du parallélisme proposé dans [Per85]. Nous donnons ensuite, les autres principes de ce langage. Nous essayons ensuite de définir et de formaliser les relations d'asynchronisme entre les types de communication. Le paragraphe qui suit montre comment appliquer les différentes stratégies vues dans les chapitres précédents pour prouver ces relations. Enfin, nous terminons par une application de ces relations d'asynchronisme à la modification des programmes parallèles.

Partie I

Etude formelle

Chapitre 1

Étude formelle du problème de validité

1 Introduction

L'objectif de ce chapitre est de définir le cadre formel sur lequel est fondé notre travail, ainsi que les bases théoriques et les notations qui vont servir dans la suite de cette thèse. Nous commencerons par décrire la notion d'algèbre qui est la représentation formelle des spécifications algébriques. Ensuite, pour formaliser notre objectif, qui est de prouver qu'une propriété est une conséquence logique d'une spécification, nous verrons que cela veut dire qu'il faut prouver que cette propriété est valide dans des classes particulières d'algèbres qui représentent cette spécification. Nous serons donc obligé de définir la notion de validité d'une propriété dans une algèbre. Nous commencerons par la validité des équations simples. Nous verrons aussi que les algèbres qui représentent une spécification sont toutes les algèbres qui valident toutes les équations de la spécification. Nous supposerons au début que tous ces équations sont des équations simples. Nous donnerons ensuite deux représentants importants, avec leurs conditions d'existence, de cette classe d'algèbres qui sont l'algèbre libre et l'algèbre initiale. Ensuite partant du fait que la définition de validité est une définition formelle qui ne peut être automatisable, nous essaierons de donner une définition syntaxique équivalente en utilisant la notion de E-égalité, où E est l'ensemble des équations de la spécification donnée. Pour cela, nous définirons la notion de congruence et d'algèbre quotient. Le théorème de Birkhoff permettra de passer de la validité formelle à la validité syntaxique. Nous essaierons ensuite de suivre la même démarche pour les spécifications contenant des équations conditionnelles. Un rappel sur la logique du premier ordre permettera de définir la notion de formule équationnelle utilisée pour exprimer les propriétés élémentaires des spécifications. Nous terminerons par définir la validité de ces formules équationnelles.

Les sections 1, 2, 3 et 4 de ce chapitre sont largement inspirées de [EM85], [Gal86], [Hul80], [HO80], [Mar86], [Rem82] et [Kap84].

2 Algèbre à plusieurs sortes

La notion d'algèbre constitue un cadre formel pour étudier plusieurs domaines de l'informatique et en particulier les types abstraits de données. La formalisation de ces types abstraits nécessite l'utilisation des algèbres à domaines et opérations dans différents types (ou sortes). Ceci nous amène à introduire la notion d'algèbre multi-sortes (ou hétérogènes). Chaque domaine est nommé par un symbole appelé *sorte*, de même que chaque opération est une application à zéro, un ou plusieurs arguments de sortes différentes.

Dans la suite le symbole S sera considéré comme un ensemble non vide de sortes.

Définition 1.1 Alphabet profilé

Un alphabet profilé *par S est un couple (F, r) formé d'un ensemble de symboles F et une fonction $r : F \longrightarrow S^* \times S$, qui associe à chaque symbole f de F un profil (u, s) noté aussi par $u \rightarrow s$.*

Soit f un symbole de profil (u, s). La longueur de la suite u est appelé *arité* de f. Si $u = s_1 \dots s_n$, $(n \geq 1)$ le symbole f est interprété comme une opération dont le $i^{ème}$ argument est de sorte s_i et qui a un résultat de sorte s. Dans ce cas, nous disons que f est de sorte s. Si u est égale à la liste vide alors le symbole f est un symbole de *constante*. Pour la simplicité de notation un alphabet (F, r) profilé par S sera noté par F.

Définition 1.2 Signature

Une signature $\Sigma = (S, F)$ est la donnée d'un ensemble de sortes S et d'un alphabet F profilé par S.

Exemple 1.1

Soit ENTIER une signature définie par :
$S = \{entier\}$
$F = \{0, succ, add\}$ *avec les profils suivants :*
$0 : \longrightarrow entier$
$succ : entier \longrightarrow entier$
$add : entier, entier \longrightarrow entier$

Exemple 1.2

Soit ENS_ORD une signature définie par :
$S = \{ens\_ord, entier\}$,
$F = \{ens\_vide, union, max, min\}$ *avec les profils suivants :*
$ens\_vide : \longrightarrow ens\_ord$,
$union : ens\_ord \times entier \longrightarrow ens\_ord$,
$max : ens\_ord \longrightarrow entier$,
$min : ens\_ord \longrightarrow entier$,

Définition 1.3 Σ_algèbre

Soit $\Sigma = (S, F)$ une signature. Une Σ_algèbre A est un couple (D_A, F_A) avec $D_A = (D_{A_s})_{s \in S}$ une famille indexée par S d'ensembles non vides; chaque D_s est appelé domaine de sorte s et $F_A = (F_s)_{s \in S}$ une famille indexée par S d'opérations sur D_A. Il existe une fonction I appelé

interprétation qui à chaque élément f de F de profil $u \rightarrow s$ avec $u = s_1 \dots s_n$, associe une opération de $F_A : I(f) = f_A : D_{A_{s_1}} \times \dots \times D_{A_{s_n}} \longrightarrow D_{A_s}$. Les constantes de F de profil $\rightarrow s$ sont interprétés comme des éléments de D_{A_s}.

Nous parlerons de classe d'algèbres lorsqu'il s'agit d'un ensemble d'algèbres sur une même signature.

Exemple 1.3

Soit ENTIER la signature de l'exemple 1.2. L'ensemble des entiers naturels et l'ensemble des entiers relatifs peuvent être munis d'une structure de ENTIER_algèbre tel que :
$entier\_naturel = (\{\mathbb{N}\}, \{0, +1, +\})$; *et*
$entier\_relatif = (\{Z\}, \{0, +1, +\})$.
avec $+1$ comme la fonction successeur et $+$ l'addition habituelle sur les entiers

Exemple 1.4

Soit ENS_ORD la signature de l'exemple 1.2. L'ensemble des parties finies et ordonnées d'entiers peut être muni d'une structure de ENS_ORD_algèbre :
$(\{ens\_ord_N, N\}, \{ens\_vide_N, union_N, max_N, min_N\})$. *$ens\_vide_N$ définit l'ensemble vide habituel, $union_N$ définit l'adjonction d'un entier dans un ensemble, max_N définit le maximum d'un ensemble et min_N définit le minimum d'un ensemble.*

Notation 1.1
Soit $u = s_1 \dots s_n$ une suite de sortes. D_A^u désignera le produit cartésien $D_{A_{s_1}} \times \dots \times D_{A_{s_n}}$
Soit $h = (h_s)_{s \in S}$ une famille indexée par S de fonctions $h_s : D_{A_s} \longrightarrow D_{B_s}$, la fonction $h^u : D_A^u \longrightarrow D_B^u$ est définie par : pour tout $(a_1, \dots a_n)$ appartenant à D_A^u, $h^u(a_1, \dots, a_n) = (h_{s_1}(a_1), \dots, h_{s_n}(a_n))$.

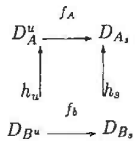
2.1 Opérations sur les algèbres

Définition 1.4 Morphisme

Soient A et B deux Σ_algèbre. Une famille $h = (h_s)_{s \in S}$ de fonctions $h_s : D_{A_s} \longrightarrow D_{B_s}$ est un morphisme si et seulement si elle vérifie la propriété suivante :
Pour tout symbole de fonction f de profil (u, s) avec $u = s_1, \dots, s_n$ et pour tout $(a_1, \dots, a_n)$ de D_A^u

$$h_s(f_A(a_1, \dots, a_n)) = f_B(h_{s_1}(a_1), \dots, h_{s_n}(a_n)) = f_B(h^u(a_1, \dots, a_n))$$

Cette condition peut être représentée par le diagramme suivant :



Si, de plus, h est bijectif, alors h est un isomorphisme de A vers B.

Exemple 1.5

Soient entier_naturel $= (\{IN\}, \{0, +1, +\})$ *et entier_relatif* $= (\{Z\}, \{0, +1, +\})$ *les deux ENTIER_algèbres définies dans l'exemple 1.3.*
L'application f : *entier_naturel* $\longrightarrow$ *entier_relatif tel que* $f(0) = 0$, $f(+1(x)) = +1(f(x))$ *et* $f(x + y) = f(x) + f(y)$ *est un morphisme d'algèbres.*

Définition 1.5 Sous algèbre

Soient $A = (D_A, F_A)$ *et* $B = (D_B, F_B)$ *deux* Σ*_algèbres. B est une sous algèbre de A si et seulement si*

- *Chaque* D_{B_s} *est un sous ensemble de* D_{A_s} *pour toute sorte s de S*
  et
- *pour tout symbole de fonction f de profil* $u \rightarrow s$ *avec* $u = s_1, \ldots, s_n$, $f_B : D_B^u \longrightarrow D_{B_s}$ *est une restriction de* $f_A : D_A^u \longrightarrow D_{A_s}$.

Exemple 1.6

Soient entier_naturel et entier_relatif les deux ENTIER_algèbre de l'exemple 1.3. Il est clair que entier_naturel et une sous-algèbre de entier_relatif.

Définition 1.6 Produit d'algèbre

Soit $A_i = (D_{A_i}, F_{A_i})_{i \in I}$ *une famille de* Σ*_algèbres indexées par un ensemble I. Le produit de ces algèbres noté* $A = (D_A, F_A)$ *est une* Σ*_algèbre définie par :*

- $D_{A_s} = \prod_{i \in I} D_{A_{i_s}}$ *: produit cartésien,*
- *pour tout symbole de fonction de profil* $u \rightarrow s$ *avec* $u = s_1, \ldots, s_n$ *et pour tout* $(x_1, \ldots, x_n)$ *de* $(\prod_{i \in I} D_{A_i})^u$, $\prod_i(f_{A_i}(x_1, \ldots, x_n)) = f_{A_i}(\prod_i(x_1), \ldots, \prod_i(x_n))$, *où* $\prod_i$ *est la* $i^{ème}$ *projection.*

2.2 Algèbre libre et algèbre initiale

Définition 1.7 Algèbre libre

Soit C une classe de Σ*_algèbres et* $X = (X_s)_{s \in S}$ *une famille indexée d'ensembles. Nous appelons* Σ*_algèbre C_libre engendrée par X (ou sur X) toute* Σ*_algèbre* $A = (D_A, F_A)$ *telle que :*

- $A \in C$,
- $X_s \subseteq D_{A_s}$,
- *pour tout* Σ*_algèbre* $B = (D_B, F_B)$ *de C et toute famille* $h : X \longrightarrow D_B$ *d'application* $h_s : X_s \longrightarrow D_{B_s}$, *il existe un unique homomorphisme de A vers B qui prolonge h sur* D_A.

Les définitions suivantes nous permettrons de donner une définition constructive de l'algèbre libre.

Définition 1.8 Plus petite sous-algèbre

Soient A une Σ*_algèbre, et* $X = (X_s)_{s \in S}$ *une famille indexée, avec* $X_s \subset D_{A_s}$. *La plus petite sous-algèbre de A contenant X et notée* $[X]$ *est caractérisée par son domaine* $D_{[X]} = (D_{[X_s]})_{s \in S}$ *qui est défini récursivement par :*

- $D_{[X_s]_0} = X_s \cup \{c_A \mid c$ *est une constante de sorte* $s\}$,
- $D_{[X_s]_{i+1}} = D_{[X_s]_i} \cup \{f_A(x_1, \ldots, x_n) \mid r(f) = u \rightarrow s$ *et* $(x_1, \ldots, x_n) \in (D_{[X]_i})^u\}$, *avec* $u = s_1, \ldots, s_n$.

Le domaine de sorte s de l'algèbre $[X]$ *est* $\cup_{i \geq 0} D_{[X_s]_i}$.

Hypothèse 1

Dans tout ce qui suit, nous supposons que pour chaque sorte s, $X_s \cup F_s$ *est non vide. Cela nous permettra de garantir que* $D_{[X_s]_0}$ *est non vide, et par suite que* $D_{[X]_s}$ *est non vide aussi.*

Définition 1.9 Sous algèbre libre

La sous algèbre $[X]$ *définie précédemment est un sous algèbre libre générée par X dans A si et seulement si les conditions suivantes sont vérifiées :*

- *Pour toute fonction f de profil* $u \rightarrow s$ *(non constante), la restriction de* $f_A : D_A^u \longrightarrow D_{A_s}$ *à* $[X]^u$ *est une fonction injective.*
- *Pour tout* $f_A : D_A^u \longrightarrow D_{A_s}$ *et* $g_A : D_A^v \longrightarrow D_{A_{s'}}$, *si* $f \neq g$ *alors* $f([X]^u) \neq g([X]^v)$.
- *Pour tout* $f_A : D_A^u \longrightarrow D_{A_s}$ *et pour tout* $(x_1, \ldots, x_n) \in D_{[X]}^u$, $f_A(x_1, \ldots, x_n) \notin X_s$.

Théorème 1.1 Extension homomorphe unique *[Gal86]*

Soient A et B deux Σ*_algèbres,* $X = (X_s)_{s \in S}$ *une famille indexée de sous ensembles de* A_s *et* $[X]$ *l'algèbre libre engendrée par X sur A.*
Pour toute famille indexée par s, $h : X \longrightarrow D_B$ *de fonction* $h_s : X_s \longrightarrow D_{B_s}$ *il existe un morphisme unique* $\overline{h} : [X] \longrightarrow B$ *tel que :*

- *pour tout x de* X_s, $\overline{h_s}(x) = h_s(x)$ *pour toute sorte s de S*
  et
- *pour chaque symbole de fonction f de profil* $u \rightarrow s$ *avec* $u = s_1, \ldots, s_n$ *et pour tout* $(x_1, \ldots, x_n)$ *de* $D_{[X]}^u$
  $\overline{h}_s(f_A(x_1, \ldots, x_n)) = f_B(\overline{h}_{s_1}(x_1), \ldots, \overline{h}_{s_n}(x_n))$.

$$X \longrightarrow [X]$$

$$B$$

En utilisant le théorème précédent nous pouvons démontrer que l'algèbre libre engendrée par X dans une classe d'algèbres C, si elle existe, est unique à un isomorphisme près. Le théorème suivant nous donne les cas où cette algèbre libre existe.

Théorème 1.2 *[Bir35]*

Soient C une classe de Σ*_algèbres stables par produit et par sous algèbre et* $X = (X_s)_{s \in S}$ *une famille indexée d'ensembles. La* Σ*_algèbre C_libre sur X existe.*

Convention 1

Nous parlerons d'algèbre libre sur X à la place d'algèbre C_libre sur X quand il n'y aura pas d'ambiguïté sur la classe C.

Définition 1.10 Algèbre initiale

L'algèbre initiale associée à une classe d'algèbres C est l'algèbre C_libre sur $\emptyset$.

2.3 Termes, arbres étiquetés

Soient $\Sigma = (S, F)$ une signature et $X = (X_s)_{s \in S}$ une famille indexée d'ensembles. Les éléments de X_s sont appelés des variables de sorte s. Soit C la classe de toutes les Σ_algèbres. La Σ_algèbre libre sur X existe et elle sera notée par $T(F, X)$. L'algèbre initiale sera notée par $T(F)$

Définition 1.11 Terme

Nous appelons termes les éléments de $T(F, X)$.

Les termes dont la racine est une fonction ou une variable de sorte s sont dits des termes de sorte s. L'ensemble des termes de sorte s sera noté $T(F, X)_s$. Les éléments de l'algèbre initiale sont appelés des termes *clos*.

Exemple 1.7

Soient ENS_ORD la signature de l'exemple 1.2, et $X = (X_{ens\_ord}, X_{entier})$,
avec $X_{ens\_ord} = (\{E_1, \ldots, E_n, \ldots\}$ et $X_{entier} = \{x_1, \ldots, x_n, \ldots\}$.
Les expressions suivantes sont des termes, éléments de $T(ENS\_ORD, X)$: E_1, x_4, ens_vide, $union(E_1, x_4)$, $max(union(union(ens\_vide, x_1), min(E_1)), x_4)$.

Cette forme fonctionnelle des termes n'est pas facile à manipuler, c'est pourquoi nous allons définir une image isomorphe à celle-ci. Il s'agit de l'arbre étiqueté représentant un terme.

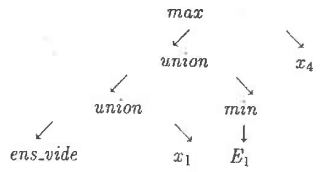
Définition 1.12 Arbre étiqueté

Soient une application t de N_+^* dans $F \cup X$ et $O(t)$ son domaine de définition. Alors t est un arbre étiqueté sur $F \cup X$ si et seulement si :

- *Le mot vide ϵ appartient à $O(t)$,*
- *si u appartient à $O(t)$ alors $u.i$ appartient à $O(t)$ si et seulement si i est compris entre 1 et l'arité de $t(u)$. (On suppose que le point désigne la concaténation entre une suite et un élément. Il sera indifféremment utilisé à gauche ou à droite).*

Exemple 1.8

Soit le terme $t = max(union(union(ens\_vide, x_1), min(E_1)), x_4)$,
$O(t) = \{\epsilon, 1, 11, 111, 112, 12, 121, 2\}$. Sa représentation arborescente est :



Lemme 1.1

L'ensemble des arbres étiquetés sur $F \cup X$ est une Σ_algèbre libre engendrée par X

Comme toutes les Σ_algèbres libres sont isomorphes, alors $T(F, X)$ et l'ensemble des arbres étiquetés le sont aussi. Terme et arbre correspondent donc à la même notion. Dans la suite nous utiliserons indifféremment ces deux notions.
Les éléments de $O(t)$ sont appelés des *occurrences* de t. Elles représentent les différents chemins d'accès dans l'arborescence de t.

Définition 1.13 Sous terme

Soient t un terme et u une occurrence de t. Le sous terme de t à l'occurrence u noté $t_{/u}$ est défini par :

- $t_{/\epsilon} = t$,
- $f(t_1, \ldots, t_n)_{/i.u} = t_{i/u}$

Définition 1.14 Remplacement dans un terme

Soient t et t' deux termes et u une occurrence de t. Le terme issu de t par remplacement de son sous terme à l'occurrence u par t' noté $t[u \leftarrow t']$ est défini par :

- $t[\epsilon \leftarrow t'] = t'$,
- $f(t_1, \ldots, t_n)[i.u \leftarrow t'] = f(t_1, \ldots, t_i[u \leftarrow t'], \ldots, t_n)$.

Lorsque nous avons une famille d'occurrence $(v_i)_{i \in I}$ à remplacer par le même terme t', nous notons $t[(v_i)_{i \in I} \leftarrow t']$.

Définition 1.15 Variables d'un terme

Soit t un terme de $T(F, X)$. L'ensemble des variables de t noté $var(t)$ est défini par $var(t) = \{x \in X \mid \exists u \in O(t)$ et $t_{/u} = x\}$.

Exemple 1.9

Soient $t = max(union(union(ens\_vide, x_1), min(E_1))$ et $t' = min(E_1)$.
t' est un sous terme de t à l'occurrence 12.
$t'' = t[112 \leftarrow t'] = max(union(union(ens\_vide, min(E_1)), min(E_1))$.
$var(t) = \{x_1, E_1\}$

Définition 1.16 Substitution

Une substitution σ est une application de X dans $T(F, X)$. Son domaine noté $dom(\sigma)$ est égal à l'ensemble des variables telle que $\sigma(x) \neq x$. Le prolongement de σ à $T(F, X)$ est un endomorphisme. Il vérifie donc pour toute fonction f de profil $u \rightarrow s$ avec $u = s_1, \ldots, s_n$, et pour tout $(t_1 \ \ldots \ t_n)$ de $T(F, X)^u$, $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$. La composition des substitutions est la composition classique des applications $(\sigma\beta)x = \sigma(\beta(x))$.
Une substitution σ est appelée substitution close lorsqu'elle est une application de X vers $T(F)$.

Définition 1.17 Filtrage

Soient t et t' deux termes de $T(F, X)$, nous disons que t est une instance de t' (ou t' filtre t) s'il existe une substitution σ telle que $\sigma(t') = t$. La substitution σ s'appelle filtre de t' vers t .

Définition 1.18 Unification

Deux termes s et t sont unifiables s'il existe une substitution σ, appelé unificateur, telle que $\sigma(s) = \sigma(t)$.

Proposition 1.1

Si deux termes s et t sont unifiables, il existe un unificateur σ, tel que pour tout autre unificateur β, nous puissions trouver une substitution α vérifiant $\beta = \alpha o \sigma$.

σ est appelé unificateur principal de s et t, il est unique à un renommage près des variables. Plusieurs algorithmes ont été proposés pour trouver un unificateur principal et notamment dans [Rob65] et [Hue76].

2.4 Type abstrait et spécification équationnelle

La notion de type abstrait, [GTW78], [Fin79], permet de décrire les objets en séparant complètement la description des aspects apparents du comportement (définition d'un type et de son comportement) de la description des aspects qui peuvent être cachés (détail de l'implantation). Cette propriété parmi d'autres fait des types abstraits un bon outil de spécification. En effet, la description de leurs aspects apparents peut se faire d'une manière syntaxique à l'aide de spécifications équationnelles.

Définition 1.19 Équation simple

Une équation simple est un couple de termes (t_1, t_2) de $T(F,V)^2_s$, où s est une sorte appartenant à S, elle sera noté $t_1 == t_2$.

Définition 1.20 Spécification équationnelle

Une spécification équationnelle (S, F, E) est la donnée d'une signature $\Sigma = (S, F)$ et d'une famille $E = (E_s)_{s \in S}$ d'ensembles d'équations. Nous la noterons aussi (Σ, E) quand il n'y a pas d'ambiguité sur la signature.

Dans une spécification équationnelle la signature représente la description des objets manipulés par le type abstrait, tandis que les équations décrivent son comportement.

Exemple 1.10

Dans cet exemple nous allons donner la spécification des booléens avec seulement la négation $\neg$ et la conjonction et comme opérations de base. Cette spécification sera utilisée dans la suite pour définir les spécifications conditionnelles.

Sorte *bool.*
opérations :
$vrai : \longrightarrow bool$
$faux : \longrightarrow bool$
$\neg : bool \longrightarrow bool$
$et : bool, bool \longrightarrow bool$
axiomes :
$\neg vrai == faux$
$\neg faux == vrai$

$\neg(\neg b) == b$
$b \; et \; vrai == b$
$b \; et \; faux == faux$
f-axiomes

L'algèbre des booléens A_BOOL sera définie par :
$A\_BOOL = (\{vrai, \; faux\}, \{vrai, \; faux, \; et, \; \neg\})$

Exemple 1.11
On considère la spécification algébrique des entiers :

sorte *entier.*
opérations:
$0 : \longrightarrow entier$
$s : \; entier \longrightarrow entier$
$entier \; + \; entier \longrightarrow entier$
axiomes:
$x \; + \; 0 \; == \; x$
$x \; + \; s(y) \; == \; s(x + y)$
f-axiomes

Le comportement d'un type abstrait peut être décrit en précisant les propriétés des opérations associées. Parmi ces opérations se distinguent des opérations que nous appelons des *constructeurs:* ce sont les opérations à partir des quelles tous les objets du types abstrait peuvent être exprimés ou engendrés. Les autres opérations sont appelés des *opérations définies.*
Dans l'exemple précédent 0 et s sont les constructeurs et + est une opération définie.

Quand nous aurons besoin de distinguer ces deux types d'opérations dans une spécification nous la noterons par $(S, C \cup D, E)$. C désignera l'ensemble des constructeurs . D désignera celui des opérations définies.

Définition 1.21 Sémantique d'une équation simple

Soient A une Σ_algèbre. L'équation $t_1 == t_2$ est valide dans A, ou A est un modèle de $t_1 == t_2$, et nous notons $A \models_\Sigma t_1 == t_2$, si et seulement si pour toute application $v : var(t_1) \cup var(t_2) \longrightarrow D_A$, $v(t_1) = v(t_2)$.

L'application v peut être étendue à un morphisme $\bar{v}$ entre $T(F,X)$ et D_A. La condition de validité s'écrira alors $\bar{v}(t_1) = \bar{v}(t_2)$.

Une Σ_algèbre A est un modèle d'un ensemble d'équations E si et seulement si A est un modèle de chaque élément de E.
Une classe de Σ_algèbres C valide une équation si et seulement si chaque algèbre de C valide cette équation.

La sémantique d'un type abstrait associé à une spécification (S, F, E) est donnée par une sous classe de l'ensemble des (S, F)_algèbres validant E et qui sont isomorphes. Chaque algèbre de cette sous classe est une représentation des données du type.

La construction d'une spécification peut se faire avec des enrichissements et des extensions successives. Beaucoup de travaux ont été effectués dans ce domaine. Pour plus de détails voir [GTW78], [Fin79] [Rem82] et [TW82] dans lesquelles on peut trouver les différentes sémantiques associées aux spécifications. Par contre nous allons nous attaquer au problème de validité d'une équation dans une spécification équationnelle.

3 Problème de validité

Maintenant que nous avons défini les pierres de base, nous allons nous intéresser au problèmes de validité. Il existe deux types de validité d'une équation dans une spécification: La validité équationnelle et la validité inductive. La première consiste à utiliser des remplacements de termes par leurs égaux jusqu'à trouver une égalité entre les termes. La deuxième consiste à utiliser une récurrence sur certaines variables. Pour ces deux types de validité nous allons commencer par leur aspect formel pour retrouver à la fin leur aspect syntaxique.

Exemple 1.12

On considère la spécification des entiers de l'exemple 1.11. L'équation $x + s(0) == s(x)$ est équationnellement valide, tandis que les équations $x + y == y + x$ et l'équation $x+(y+z) == (x+y)+z$ nécessitent une récurrence. Elles sont inductivement valides.

3.1 Validité équationnelle

Une équation est équationnellement valide dans une spécification si est seulement si elle est valide dans tous les modèles de celle ci. La notion de variété équationnelle regroupe l'ensemble de ces modèles.

Définition 1.22 Variété équationnelle d'algèbres

Soit E un ensemble d'équations d'une spécification (S, F, E). Une variété équationnelle d'algèbres engendrées par E, noté par $M(E)$ est la classe de toutes les algèbres validant E.

Définition 1.23 Validité équationnelle d'une équation simple

Soient $SP = (\Sigma, E)$ une spécification équationnelle et t_1, t_2 deux termes de $T(F, X)$. L'équation $t_1 == t_2$ est équationnellement valide dans SP si et seulement si elle est valide dans chaque algèbre de $M(E)$. Nous la notons par $SP \models_\Sigma t_1 == t_2$ ou $E \models_\Sigma t_1 == t_2$.

Pour pouvoir automatiser cette définition sémantique de la validité, nous voudrions lui donner un aspect syntaxique. Pour cela nous allons nous intéresser à la congruence engendrée par l'ensemble d'équations de la spécification. Et le problème de validité d'une équation se résumera à tester si celle-ci appartient à une classe d'équivalence de cette congruence.

Définition 1.24 Congruence

Soit $A = (D_A, F_A)$ une Σ-algèbre. Une congruence $\simeq$ sur A est une famille indexée $(\simeq_s)_{s\in S}$ de relations telle que chaque relation $\simeq_s$ est une relation d'équivalence sur D_A qui vérifie la condition suivante :

- *Pour chaque symbole de fonction f de profil $u \rightarrow s$ avec $u = s_1, \ldots, s_n$ et pour tout $(x_1, \ldots, x_n)$ et $(y_1, \ldots, y_n)$ de D^u_A*
  si $x_i \simeq_{s_i} y_i$ pour tout i de $[1..n]$ alors $f_A(x_1, \ldots, x_n) \simeq_s f_A(y_1, \ldots, y_n)$

La classe d'équivalence de x modulo $\simeq_s$ sera notée par $[x]_s$.

Définition 1.25 Algèbre quotient

Soient A une Σ-algèbre et $\simeq$ une congruence sur A. L'algèbre quotient $A_{/\simeq}$ est définie par $D_{A_{/\simeq}} = (D_{A_s/\simeq_s})_{s\in S}$ comme domaine, et $F_{A_{/\simeq}}$ comme ensemble d'opérations. Ces opérations vérifient la condition suivante :

- *Pour tout symbole de fonction f de profil $u \rightarrow s$ avec $u = s_1, \ldots, s_n$, pour tout $([a_1]_{s_1}, \ldots, [a_n]_{s_n})$ élément de $D^u_{A/\simeq}$*
  $f_{A/\simeq}([a_1]_{s_1}, \ldots, [a_n]_{s_n}) = [f_A(a_1, \ldots, a_n)]_s$.

La famille indexée $h_\simeq = (h_{\simeq_s})_{s\in S}$ avec $h_{\simeq_s} : D_{A_s} \longrightarrow D_{A/\simeq_s}$, et tel que $h_{\simeq_s}(x) = [x]_s$, est un morphisme de A vers $A_{/\simeq}$.

Appliquons maintenant cette notion de congruence sur les équations.

Définition 1.26 Congruence engendrée par un ensemble d'équations simples

Soit E un ensemble d'équations sur $T(F, X)$. La plus petite congruence sur $T(F, X)$ contenant tous les couples $(\sigma(t_1), \sigma(t_2))$ où $t_1 == t_2$ est une équation de E et σ une substitution, est appelée la congruence engendrée par E ou la E-égalité et notée $=_E$.

Définition 1.27 Théorie équationnelle

La théorie équationnelle engendrée par un ensemble d'équations E est égale à l'ensemble des équations qui sont valides dans tous les algèbres de $M(E)$.

Lorsque E est l'ensemble vide nous parlons de la théorie équationnelle vide qui correspond à $T(F, X)$.

Théorème 1.3 *[EM85]*

L'algèbre libre associée à la variété équationnelle $M(E)$ d'algèbres engendrées par un ensemble d'équations E est égale à $(T(F, X)_{/=_E}, F)$.

Ce théorème a pour conséquence de transformer la preuve de la validité d'une équation $t_1 == t_2$ dans $M(E)$ en une preuve dans $(T(F, X)_{/=_E}, F)$. Cela est suffisant car d'après la définition de l'algèbre libre il existe toujours un morphisme de $(T(F, X)_{/=_E}, F)$ vers chaque algèbre de $M(E)$. Mieux encore la preuve de la validité dans $(T(F, X)_{/=_E}, F)$ peut se faire d'une manière syntaxique en utilisant la relation $=_E$. Cela est une conséquence du Théorème de Birkhoff qui est la base de la logique équationnelle.

Théorème 1.4 *[Bir35]*

Soit E un ensemble d'équations. Une équation $t_1 == t_2$ est valide dans la variété $M(E)$ si et seulement si $t_1 =_E t_2$.

La relation $=_E$ peut être définie comme la fermeture reflexive symétrique et transitive de la relation de *E-égalité en une étape* noté par $\vdash\dashv_E$.

Définition 1.28 E-égalité en une étape

Soient E un ensemble d'équations et t_1 et t_2 deux termes. t_1 est E-égale à t_2 en une étape, noté par $t_1 \vdash\dashv_E t_2$, si et seulement si :
il existe une équation $g == d$ de E, une occurrence u de t_1 et une substitution σ tels que $t_{1/u} = \sigma(g)$ (ou resp $t_{1/u} = \sigma(d)$) et $t_2 = t_1[u \leftarrow \sigma(d)]$ (ou resp $t_2 = t_1[u \leftarrow \sigma(g)]$).

Ainsi pour deux termes t et t', $t =_E t'$ si et seulement si il existe $(t_1, \ldots, t_n)$ de $T(F, X)^n$ tels que

- $t = t_1$

- $t_i \vdash\dashv_E t_{i+1}$ pour tout i *de* $[1..n]$
- $t_n = t'$

Maintenant pour prouver qu'une équation $t_1 == t_2$ est valide dans la variété $M(E)$ d'algèbres engendrées par un ensemble d'équations E, il faut et il suffit d'utiliser un nombre fini de fois la relation $\vdash\dashv_E$ qui consiste à faire des remplacements des termes par leurs égaux. Nous disons dans ce cas que $t_1 == t_2$ est une conséquence équationnelle de E et nous la notons par $E \vdash t_1 == t_2$.

3.2 Validité inductive

L'autre aspect de validité des équations dans une spécification est celui de la validité inductive. Il s'agit des équations qui nécessitent une récurrence pour être prouvées. Cette récurrence utilise comme support l'ensemble des constructeurs de la spécification dans laquelle va se dérouler la preuve. Il s'agit donc d'une validité inductive relative à cet ensemble de constructeurs [Zha88,GG88]. Intuitivement, la validité inductive d'une équation est définie comme la validité de celle ci dans les algèbres associées à la spécification donnée qui sont image homomorphe de l'algèbre des termes clos sur les constructeurs.

Convention 2

Dans la suite, nous supposons donnée une spécification $(S, C \cup D, E)$ avec $F = C \cup D$ et $C \cap D = \emptyset$. Nous supposons aussi que l'ensemble des constructeurs est non vide.

Définition 1.29 Algèbre C_finiment engendrée

Une algèbre $A = (D_A, F_A)$ est une algèbre finiment engendrée en respectant les constructeurs si la restriction du morphisme h_A de $T(F)$ à A sur $T(C)$ est surjective. Nous dirons aussi que A est une algèbre C_finiment engendrée.

Intuitivement, une algèbre C_finiment engendrée est une algèbre qui est image homomorphe de $T(C)$ qui est l'algèbre des termes clos sur les constructeurs.

Définition 1.30 Validié inductive d'une équation

Une équation $t_1 == t_2$ est inductivement valide dans une spécification (S, F, E) en respectant les constructeurs si est seulement si elle est valide dans toute algèbre C_finiment engendrée de $M(E)$. Nous la notons $SP \models_\Sigma t_1 == t_2$ ou $E \models_\Sigma t_1 == t_2$. Avec $\Sigma = (S, F)$.

Comme dans le cas de la validité équationnelle, nous allons donner un aspect syntaxique à cette définition formelle de validité inductive. Pour cela nous allons définir la congruence engendrée par l'ensemble d'équations E sur l'algèbre $T(C)$.

Proposition 1.2

Soit (S, F, E) une spécification équationnelle. La plus petite congruence contenant tous les couple $(\sigma(t_1), \sigma(t_2))$ où $t_1 == t_2$ est une équation de E et σ une substitution de X vers $T(C)$ est égale à la restriction de la congruence $=_E$ sur $T(C)$.

Proposition 1.3

L'algèbre quotient $(T(C)_{/=_E}, F)$ appelée algèbre des constructeurs est égale à l'algèbre initiale de la classe d'algèbres de $M(E)$ qui sont C_finiment engendrées.

Preuve :

Nous pouvons facilement constater que $(T(C)_{/=_E}, F)$ est C_finiment engendrée. Mais il reste à montrer que pour toute algèbre A C_finiment engendrée il existe un homomorphisme $h_{A/E} : T(C)_{/=_E} \longrightarrow D_A$.
Soit A une Σ_algèbre C_finiment engendrée, alors il existe un homomorphisme surjectif $h_A : T(C) \longrightarrow D_A$. Cet homomorphisme définit une congruence $=_A$ sur $T(C)$ telle que $t =_A t'$ si et seulement si $h_A(t) = h_A(t')$.
Or, $=_E$ est la plus petite congruence sur $T(C)$, donc $=_E \subset =_A$ et par conséquent $[t]_E \subset [t]_A$.
En d'autres termes pour tout terme $t' \in [t]_{=_E}$, $t' \in [t]_{=_A}$ et par conséquent $h_A(t) = h_A(t')$. L'homomorphisme $h_{A/E}$ est alors défini par $h_{A/E}([t]_{=_E}) = h_A(t)$.

Cette proposition a pour conséquence de ramener la preuve de validité inductive de la preuve dans toutes les algèbres C_finiment engendrées de $M(E)$ à la preuve de validité dans $(T(C)_{/=_E}, F)$ qui est l'algèbre initiale de cette classe. Or pour prouver la validité d'une équation dans cette algèbre il faut et il suffit de prouver que toute instance close dans les constructeurs de cette équation est équationnellement valide :

Théorème 1.5

Une équation $t_1 == t_2$ est inductivement valide en respectant les constructeurs si et seulement si pour toute substitution close sur les constructeurs $\sigma : X \longrightarrow T(C)$, $\sigma(t_1) =_E \sigma(t_2)$.

En résumé, pour prouver qu'une équation est inductivement valide dans une spécification, il suffit prouver que les résultats du remplacement de ses variables par toutes les instances closes sur les constructeurs possibles sont équationnellement valides. Il est clair que le recencement de toutes les instances closes possibles construites à partir des constructeurs ne peut pas être automatisable. Dans le chapitre 3 nous allons utiliser une méthode pour contourner ce problème.

3.3 Spécification conditionnelle

Dans le paragraphe précédent nous avons défini les spécifications équationnelles avec seulement des équations simples. Or, comme nous pouvons le voir plus formellement dans [TW82], ces spécifications ne sont pas assez puissantes pour exprimer certains problèmes informatiques. D'où l'introduction des équations avec des préconditions ou autrement dit des équations conditionnelles.

Dans cette étude, nous allons nous intéresser à une catégorie particulière d'équations conditionnelles dont les préconditions sont de sorte booléenne. Ainsi, dans tout ce qui suit, nous considérons une signature $\Sigma = (S, F)$ avec une sorte spéciale *bool*. A cette sorte correspond un ensemble de symboles de fonction F_{bool} contenant seulement deux constantes *vrai et faux*. F_{bool} peut contenir d'autres fonctions d'arité supérieure à 0. Les Σ_algèbres contiennent donc une sous-algèbre isomorphe à l'algèbre des booléens définie dans l'exemple 1.10. Rappelons aussi, qu'un terme booléen est un élément de $T(F, X)_{bool}$; autrement dit c'est un terme dont la racine est une fonction ou une variable de sorte *bool*.

Définition 1.31 Équation booléenne

Une équation $t == p$ est une équation booléenne si et seulement si t est un terme booléen de $T(F, X)_{bool}$ ne contenant pas de connecteur logique et p une constante booléenne vrai ou faux.

Définition 1.32 Équation conditionnelle

Une équation conditionnelle peut avoir l'une des deux formes suivantes :

- $\wedge_{i\in[1..n]} p_i == p'_i \Rightarrow g == d$,
- $\wedge_{i\in[1..n]} p_i == p'_i \Rightarrow g == (d_1, d_2)$.

avec $p_i == p'_i$ sont des équations booléennes.

Une spécification (S, F, E) est dite conditionnelle si E contient des équations conditionnelles

Intuitivement, la première forme signifie que si $p_1 = p'_1$ *et* ... *et* $p_n = p'_n$ *alors* $g = d$.
La deuxième signifie que si $p_1 = p'_1$ *et* ... *et* $p_n = p'_n$ *alors* $g = d_1$ *sinon* $g = d_2$.

Exemple 1.13

La spécification du type ensemble ordonné peut être définie par (ENS_ORD, E), où ENS_ORD est la signature définie par : ({ens_ord, entier, bool}, {ens_vide, union, difference, estvide, min, max, =, <}).
Avec le profil suivant des opérations :
ens_vide : $\longrightarrow$ *ens_ord*,
union : *ens_ord* $\times$ *entier* $\longrightarrow$ *ens_ord*,
difference : *ens_ord* $\times$ *entier* $\longrightarrow$ *ens_ord*,
estvide : *ens_ord* $\longrightarrow$ *bool*,
max : *ens_ord* $\longrightarrow$ *entier*,
min : *ens_ord* $\longrightarrow$ *entier*,
$\_ = \_$: *entier* $\times$ *entier* $\longrightarrow$ *bool*
$\_ < \_$: *entier* $\times$ *entier* $\longrightarrow$ *bool*

axiomes :
difference(ensvide,xi) == ensvide
(xi = yi) == vrai $\Rightarrow$ *difference(union(xE,xi),yi) ==*
(difference(xE,yi), union(differeince(xE,yi),xi))
appartienta(ensvide,xi) == faux
(xi = yi) == vrai $\Rightarrow$ *appartienta(union(xE,xi),yi) == (vrai, appartienta(xE,yi))*
estvide(ensvide) == vrai
estvide(union(xE,xi)) == faux
min(ensvide) == INFINI
min(union(xE,xi)) == inf(xi , min(xE))
max(ensvide) == 0
max(union(xE,xi)) == sup(xi , max(xE))
(xi < yi) == vrai $\Rightarrow$ *inf(xi , yi) == (xi, yi)*
(yi < xi) == vrai $\Rightarrow$ *sup(xi,yi) == (xi, yi)*
faxiomes

theoremes
estvide(E) == vrai $\Rightarrow$ *(card(E) = 0) == vrai*
estvide(E) == faux $\Rightarrow$ *min(E)* $\leq$ *max(E) == vrai*
ftheo.

Avant de définir formellement la sémantique de ces équations conditionnelles, nous allons donner un aperçu sur les différents travaux qui abordent les spécifications conditionnelles.

Différentes approches des spécifications conditionnelles

Les travaux sur les spécifications conditionnelles et leurs propriétés globales ont connu ces dernières années une expansion considérable. Leurs objectifs est de pouvoir utiliser les mêmes résultats obtenus dans le cas des spécifications avec des équations simples. Une diversité d'approches qui se distinguent par des formes syntaxiques différentes des équations conditionnelles, a été alors proposée. Parmi cette diversité nous pouvons en distinguer trois.

La première approche utilise des équations conditionnelles qui ne contiennent pas à priori d'équations booléennes. En effet, les conditions sont des conjonctions d'équations quelconques avec des termes de même sortes. Cette approche a été utilisée dans [Kap84]. L'auteur montre que les résultats obtenus dans le cas des équations simples peuvent être prolongés au cas des équations conditionnelles. Il garantit l'existence de l'algèbre initiale et l'algèbre libre pour de nombreuses spécifications. Ces idées ont été ensuite améliorées dans [Kap87] et [Gan87]. Nous pouvons retrouver cette même forme d'équations conditionnelles, sous le nom de clause de Horn équationnelle, dans [KR87b] qui utilisent d'autres méthodes pour prouver la complétude des systèmes de réécriture conditionnelle.

Une deuxième approche utilisée dans [Nav87], considère les spécifications conditionnelles comme des extensions de la spécification des booléens. Les préconditions sont des formules propositionnelles. L'auteur fournit un système de déduction qui permet de compléter de telles spécifications.

La troisième approche que nous pouvons trouver dans [BR88] utilise des équations conditionnelles avec des équations booléennes dans les conditions. Les auteurs ont défini le comportement de ces spécifications par rapport aux booléens (consistance et complétude par rapport aux booléens). Dans ces conditions l'existence de l'algèbre initiale est garantie dans le cas des spécifications hiérarchiques.

L'approche que nous avons adoptée utilise la même forme de préconditions que dans [BR88]. Mais comme nous l'avons signalé dans l'introductions nous nous sommes penchés sur la preuve de propriétés élémentaires plutôt que sur la preuve des propriétés globales des spécifications. Toutefois, il faut signaler aussi que nous avons utilisé deux formes différentes d'équations conditionnelles : la forme *si-alors* et la forme *si-alors-sinon*. Les équations qui sont de la première forme seront réservées pour exprimer les opérations partiellement définies ainsi que les préconditions sur les opérations. Les équations de la deuxième forme seront utilisées pour exprimer les opérations complètement définies. Cette dernière forme permet de bien contrôler la définition des opérations dans le sens où elle les définit de manières exhaustives. Par ailleurs, nous verrons dans le chapitre 4 que cette forme permet de générer automatiquement des cas dans le but de faciliter la réduction des termes.

Il faut signaler aussi que cette forme de conditions que nous avons choisi n'est pas une restriction sur l'expressivité des spécifications. En effet, toute forme particulière d'équation conditionnelle peut être transformée sous forme d'équations telles qu'elles sont définies dans la définition1.32 en ajoutant éventuellement d'autres équations. Par exemple, une équation conditionnelle de la forme $p_1 \vee \ldots \vee p_n \Rightarrow g == d$ peut se transformer en plusieurs équations $p_1 \Rightarrow g == d, \ldots, p_n \Rightarrow g == d$.

Validité dans une spécification conditionnelle

Définition 1.33 Sémantique d'une équation conditionnelle

Soit A une Σ_algèbre et e une équation conditionnelle, si e est de la forme :

- $\wedge_{i\in[1..n]} p_i == p'_i \Rightarrow g == d$ *alors e est valide dans A et nous notons* $A \models_\Sigma e$ *si et seulement si pour tout application* $v : X \longrightarrow D_A$, *si pour tout i de* $[1..n]$ $v(p_i) = v(p'_i)$ *alors* $v(g) = v(d)$.
- $\wedge_{i\in[1..n]} p_i == p'_i \Rightarrow g == (d_1, d_2)$ *alors e est valide dans A et nous notons* $A \models_\Sigma e$ *si et seulement si pour toute application* $v : X \longrightarrow D_A$
  - *si pour tout i de* $[1..n]$ $v(p_i) = v(p'_i)$ *alors* $v(g) = v(d_1)$,
  - *si il existe un i de* $[1..n]$ *tel que* $v(p_i) = v(\neg p'_i)$ *alors* $v(g) = v(d_2)$.

Soit *e* une équation conditionnelle qui peut avoir une des deux formes précédemment décrites, alors *e* est valide dans une classe d'algèbre si et seulement si elle est valide dans chaque algèbre de celle-ci.

Convention 3

Une équation simple peut être considérée comme une équation conditionnelle de la première forme avec $n = 0$.
Quand nous parlerons dans la suite d'un ensemble d'équations E il sera sous entendu qu'il contiendra aussi bien des équations simples que des équations conditionnelles.

Comme dans le cas des spécifications avec des équations simples, la validité équationnelle d'une équation est définie par la validité dans la variété d'algèbres engendrées par l'ensemble des équations de cette spécification. Afin de proposer une méthode syntaxique pour prouver cette validité, comme ce qui a été fait dans [Rem82] et [Kap84], nous allons étendre les résultats obtenus dans le cas des équations simples. Cela se traduit par le calcul de la congruence engendrée par un ensemble d'équations conditionnelles, puis par une extension du théorème de Birkhoff.

Définition 1.34 Congruence engendrée par un ensemble d'équations conditionnelles

Soit E un ensemble d'équations conditionnelles. La congruence $=_E$ *engendrée par E est définie par approximation successive comme suit :*

- $=_0$ *est la plus petite congruence contenant tout les couples* $(\sigma(t_1), \sigma(t_2))$, *pour toute équation* $t_1 == t_2$ *appartenant à E et pour toute substitution* σ.
- $=_{i+1}$ *est la plus petite congruence contenant les couples* (t_1, t_2) *tels que* $t_1 =_i t_2$ *ou il existe une équation conditionnelle :*
  - $\wedge_{j\in[1..n]} p_j == p'_j \Rightarrow g == d$ *et une substitution* σ *telles que pour tout* $j \in [1..n]$, $\sigma(p_j) =_i p'_j$, $t_1 = \sigma(g)$ *et* $t_2 = \sigma(d)$.
  - $\wedge_{j\in[1..n]} p_j == p'_j \Rightarrow g == (d_1, d_2)$ *et une substitution* σ *telles que :*
    - *pour tout* $j \in [1..n]$, $\sigma(p_j) =_i p'_j$, $t_1 = \sigma(g)$ *et* $t_2 = \sigma(d_1)$
      ou
    - *il existe un* $j \in [1..n]$ *tel que* $\sigma(p_j) =_i \neg p'_j$ *et pour tout* $k \neq j$ *et* $k \in [1..n]$, $\sigma(p_k) =_i \varepsilon p'_k$, $t_1 = \sigma(g)$ *et* $t_2 = \sigma(d_2)$. ε *est égale à* $\neg$ *ou le mot vide.*

$=_E$ *est alors égale à* $\cup_{i\geq 0}\{=_i\}$

L'existence de cette congruence peut être prouvé en utilisant la théorie de point fixe comme dans [Rem82], [Kap84] ou [Bou88a]. Mais pour garantir que la congruence soit non triviale et par suite garantir l'existence d'une algèbre initiale correcte, il faut faire attention au traitement des préconditions booléennes. L'exemple suivant montre que la partie booléenne de l'algèbre initiale peut être non conforme à ce qui est attendu.

Exemple 1.14 *[KR87a]*

Soit $SP = (S, F, E)$ *la spécification suivante :*

1. $S = \{bool\}$,
2. $F = \{true, false, p, q : \rightarrow bool\}$,
3. $E = \{p == false \Rightarrow q == true\}$

$\equiv_E$ *est l'identité et le support de l'algèbre initiale de la spécification SP est l'ensemble* **{p, q, true, false}** *où* **p, q, true, false** *sont les valeurs respectives des constantes p, q, true et false. Cependant, la partie booléenne de cette algèbre initiale n'est pas isomorphe à l'ensemble* **{true, false}**.

Pour cela nous allons supposer que les spécifications conditionnelles utilisées vérifient deux propriétés qui définissent le bon comportement de ces spécifications par rapport aux booléens. Ces propriétés sont la complétude et la consistance par rapport aux booléens.

Définition 1.35 Complétude par rapport aux booléens *[BR88]*

Soit $SP = (\Sigma, E)$ *une spécification conditionnelle,*
SP est complète par rapport aux booléens si et seulement si pour chaque terme booléen clos t, $t =_E vrai$ *ou* $t =_E faux$.

Cette propriété assure que chaque terme booléen clos ne peut se réduire qu' à *vrai* ou $faux$

Définition 1.36 Consistance par rapport aux booléens *[BR88]*

Soit $SP = (\Sigma, E)$ *une spécification conditionnelle,*
SP est consistante par rapport aux booléens si et seulement si $\neg vrai =_{SP} faux$.
Comme conséquence de la consistance par rapport aux booléens, pour chaque terme booléen clos t, si $t =_E vrai$ *alors* $t \neq_E faux$.

Comme dans le cas des équations simples, la théorie équationnelle engendrée par un ensemble d'équations conditionnelles est égale à $(T(F,X)_{/=_E}, F)$, qui est aussi la Σ_algèbre libre associée à la variété d'algèbres $M(E)$ engendrée par E. D'autre part, le théorème de Birkhoff peut être prolongé aux équations conditionnelles. Cela veut dire que la méthode syntaxique qui consiste à remplacer des termes par leur égaux reste valable dans le cas des équations conditionnelles. La validité inductive est définie de la même manière que dans le cas des spécifications avec des équations simples.

Théorème 1.6 *[Rem82]*

Soit E un ensemble d'équations conditionnelles.
$t_1 == t_2$ *est valide dans* $M(E)$ *si et seulement si* $t_1 =_E t_2$.

4 Calcul des prédicats du premier ordre

Dans les paragraphes précédents nous avons défini la validité d'une équation simple dans des spécifications simples puis conditionnelles. Pour étendre la classe de propriétés élémentaires à prouvées nous allons définir la notion de formule équationnelle. Dans ce paragraphe nous rappelons les concepts essentiels de la logique du premier ordre [Sho67]. Ces concepts seront utilisés ensuite pour définir la validité d'une formule équationnelle. Nous les utiliserons aussi dans le chapitre consacré à la décomposition pour pouvoir prouver la correction de nos règles d'inférences. Nous allons surtout parler de la logique du premier ordre multi-sortes puisque nous utilisons des équations dont les membres sont des termes qui peuvent avoir plusieurs sortes.

4.1 Syntaxe de la logique du premier ordre multi-sortes

Définition 1.37 Langage de la logique du premier ordre

Un langage L de la logique du premier ordre est composé :

- *d'une signature $\Sigma = (S \cup bool, FS \cup PS)$ telle que :*
  - *bool est la sorte des booléens;*
  - *L'ensemble des symboles de prédicats PS est caractérisé par la fonction de profil $r : PS \longrightarrow S^* \times \{bool\}$ qui associe à chaque symbole de prédicat P un profil $r(P) = u \rightarrow bool$. La longueur de la suite u est l'arité du prédicat P. Si u est vide P est un symbole de proposition;*
  - *L'ensemble FS est l'ensemble de symboles de fonctions et de constantes non booléennes,*
- *d'une famille $X = (X_s)_{s \in S}$ de variables,*
- *des connecteurs logiques $\wedge$ (et), $\vee$ (ou), $\supset$ (implication), $\Leftrightarrow$ (équivalence) qui ont un profil $(bool, bool \rightarrow bool)$ et $\neg$ (non) qui a un profil $(bool \rightarrow bool)$,*
- *des quantificateurs : pour toute sorte s de S $\forall_s$ (pour tout) et $\exists_s$ (il existe),*
- *et des symboles auxiliaires '(' et ')'.*

X, FS, PS sont supposé être disjoints.

Les termes sont les éléments de $T((S, FS), X)$.

Définition 1.38 Atome

Soient P un symbole de prédicat de PS de profil $u \rightarrow bool$, avec $u = s_1, \ldots, s_n$ et $(t_1, \ldots, t_n)$ un élément de $T((S, FS), X)^u$ (t_i est un terme de sorte s_i). Nous appelons atome tout terme de la forme

$$P(t_1, \ldots t_n).$$

L'ensemble des atomes est noté par $A(SF, SP, X)$

Définition 1.39 Formule du premier ordre

Une formule du premier ordre est définie par :

- *Tout atome est une formule,*

- *soient f_1 et f_2 deux formules alors $(f_1 \wedge f_2)$, $(f_1 \vee f_2)$, $(f_1 \supset f_2)$, $(f_1 \Leftrightarrow f_2)$ et $\neg f_1$ sont des formules.*
- *pour toute variable x de sorte s et pour toute formule f_1, $\forall_s x f_1$ et $\exists_s x f_1$ sont des formules.*

Pour la simplicité de notations, nous écrirons $\forall x f_1$ et $\exists x f_1$ à la place de $\forall_s x f_1$ et $\exists_s x f_1$.

Remarque 1.1

Nous avons noté l'implication par le symbole $\supset$ au lieu du symbole $\Rightarrow$, pour différencier les formules équationnelles des équations conditionnelles. Cela à pour but de différencier les deux concepts, mais il est clair que les équations conditionnelles de la forme si-alors *sont des formules équationnelles.*

Définition 1.40 Variables libres et variables liées

Soit f_1 une formule. L'ensembles des variables libres de f_1 noté $FV(f_1)$ et l'ensemble des variables liées de f noté $BV(f_1)$ sont définis par :

- *Soit f_1 un atome alors $BV(f_1) = \emptyset$ et $FV(f_1) = \emptyset$*
- *Soient f_1 et f_2 deux formules alors $BV(f_1 * f_2) = BV(f_1) \cup BV(f_2)$ et $FV(f_1 * f_2) = FV(f_1) \cup FV(f_2)$ avec $* \in \{\vee, \wedge, \supset, \Leftrightarrow\}$,*
- *$BV(\neg f_1) = BV(f_1)$ et $FV(\neg f_1) = FV(f_1)$,*
- *$BV(\forall x f_1) = BV(f_1) \cup \{x\}$ et $FV(\forall x f_1) = FV(f_1) - \{x\}$,*
- *$BV(\exists x f_1) = BV(f_1) \cup \{x\}$ et $FV(\exists x f_1) = FV(f_1) - \{x\}$.*

Une formule f_1 est dite *close* si $BV(f_1) = var(f_1)$.

4.2 Sémantique de la logique du premier ordre

Définition 1.41 Interprétation

Une interprétation I d'un langage du premier ordre sur une signature $\Sigma = (S \cup bool, SF \cup SP)$ est une Σ-algèbre multi-sortes $I = (D_I, F_I \cup R_I)$:

- *D_I est le domaine d'interprétation avec $D_{bool} = BOOL = \{vrai, faux\}$,*
- *F_I est un ensemble d'applications dans D_I associé à l'ensemble de symboles de fonction FS tel que à chaque symbole f profilé par $u \rightarrow s$, nous associons la fonction $I(f) = f_I : D_I^u \longrightarrow D_s$,*
- *R_I est l'ensemble d'applications de D_I vers $BOOL$ (ou de relations) associés aux prédicats de PS, tel que à chaque prédicat P de PS profilé par $u \rightarrow bool$ nous associons la relation $P_I : D_I^u \longrightarrow BOOL$.*

Une interprétation permet de donner un sens à une formule en donnant aux connecteurs logiques leur signification habituelle.

Définition 1.42 Valuation

Une valuation $v : X \longrightarrow D_I$ d'une interprétation I est une famille indexée $(v_s)_{s \in S}$ d'applications $v_s : X_s \longrightarrow D_{I_s}$
Cette valuation peut être étendue sur $T((S, SF), X)$ et $A(SF, SP, X)$ par morphisme :

- $\forall f \in FS,\ v(f(t_1,\ldots,t_n)) = I(f)(v(t_1),\ldots,v(t_n))$,
- $\forall P \in PS,\ v(P(t_1,\ldots,t_m)) = I(P)(v(t_1),\ldots,v(t_m)))$.

L'ensemble de ces valuations sera noté par $[X \longrightarrow D_I]$.

Une valuation v permet d'associer à un terme t de $T((S,SF),X))$ une valeur $(I,v)(t)$ dans D_I et à chaque atome a une valeur booléenne $(I,v)(a)$ dans $\{vrai, faux\}$. Elle peut être étendue à une formule en appliquant les règles suivantes:
soient f_1, f_2 deux formules, alors

- $(I,v)(f_1 * f_2) = (I,v)(f_1) * (I,v)(f_2)$ avec $* \in \{\vee, \wedge \supset, \Leftrightarrow\}$
- $(I,v)(\neg f_1) = \neg(I,v)(f_1)$.

Définition 1.43 Valeur d'une formule

La valeur d'une formule f_1 dans une interprétation I noté par $val_I(f_1)$ est l'ensemble des valuations v de $[X \longrightarrow D_I]$ telles que $(I,v)(f_1) = vrai$.
$val_I(f_1)$ peut être calculé par récurrence sur la forme de f_1 : soient f_1 et f_2 deux formules alors :

- $val_I(f_1 \vee f_2) = val_I(f_1) \cup val_I(f_2)$
- $val_I(f_1 \wedge f_2) = val_I(f_1) \cap val_I(f_2)$
- $val_I(\neg f_1)$ = *complément de* $val_I(f_1)$ *par rapport à* $[X \longrightarrow D_I]$.
- $val_I(\forall_s x f_1) = \{v \in [X \longrightarrow D_I] \mid \forall v' \in [X \longrightarrow D_I]$ *si* $v_{|X-\{x\}} = v'_{|X-\{x\}}$ *alors* $v' \in val_I(f_1)\}$
- $val_I(\exists_s x f_1) = \{v \in [X \longrightarrow D_I] \mid \exists v' \in [X \longrightarrow D_I]$ *telle que* $v_{|X-\{x\}} = v'_{|X-\{x\}}$ *et* $v' \in val_I(f_1)\}$

Définition 1.44 Validité d'une formule

Une formule f_1 est valide dans une interprétation I si et seulement si $val_I(f_1) = [X \longrightarrow D_I]$. Nous la notons par $I \models f_1$.

Théorème 1.7

Soit f_1 une formule avec un ensemble de variables libres $FV(f_1) = \{x_1,\ldots,x_n\}$, la clôture universelle de A est une formule $A' = \forall x_1,\ldots \forall x_n A$, alors A est valide dans une interprétation I si et seulement si A' est valide dans I.

5 Application aux formules équationnelles

Etant données une signature $\Sigma = (S,F)$ et une famille indexée d'ensembles de variables $X = (X_s)_{s\in S}$, notre objectif est de définir une formule équationnelle comme une formule d'un langage du premier ordre LE dont l'ensemble des atomes $A(F,PS,X)$ est égale à l'ensemble des équations simples. Or, une équation simple est un couple de termes reliés par le symbole d'équation '=='. L'ensemble des prédicats PS sera donc égal à $\{==\}$. Cette appellation n'est pas nouvelle parce que nous avons trouvé chez plusieurs auteurs la notion de clause de Horn équationnelle, notamment chez [Kap84,Rus87,Com88].

Nous allons nous intéresser à un ensemble particulier de formules de ce langage du premier ordre LE.

Définition 1.45 Formule équationnelle

Une formule équationnelle est une formule du langage du premier ordre LE, telle que toutes ses variables sont supposées universellement quantifiées.

Puisque les termes des formules équationnelles sont des éléments de $T(F,X)$, nous parlerons de formules équationnelles sur $T(F,X)$.

Exemple 1.15

Un exemple de propriété élémentaire que nous pouvons prouver sur la spécification des ensembles ordonnés de l'exemple 1.13, peut être exprimée par la formule équationnelle

$$estvide(E) == faux \supset min(E) \leq max(E) == vrai$$

5.1 Validité d'une formule équationnelle

Nous allons définir la validité d'une formule équationnelle comme un prolongement de la validité d'une équation dans une algèbre. Pour cela nous allons nous intéresser à des interprétations particulières que nous allons construire à partir d'une algèbre donnée.

Proposition 1.4

Soit A une Σ_algèbre, que nous pouvons considérer aussi comme une interprétation du langage du premier ordre LE. Une équation $t_1 == t_2$ est valide dans la Σ_algèbre A si et seulement si elle valide dans l'interprétation A.

Preuve:
Signalons d'abord que l'interprétation du ==, $A(==)$ est égale à $=$ et que toute application $v : X \longrightarrow D_A$ peut être prolongée à une valuation $v' : X \longrightarrow D_A$ telle que $v'(t_1 == t_2) = (v(t_1) = v(t_2))$. Cette preuve consiste à montrer que les définitions des deux validités sont équivalentes:
$t_1 == t_2$ est valide dans la Σ_algèbre A si et seulement si, pour toute application $v : X \longrightarrow D_A$, $v(t_1) = v(t_2)$. Ce qui est la même chose que $(v(t_1) = v(t_2)) = vrai$. Ce qui est équivalent à $(v'(t_1)\ A(==)v'(t_2))) = vrai$. Qui est la définition de la validité de $t_1 == t_2$ dans A.

Cette proposition nous montre l'équivalence entre la validité d'une équation considérée comme un couple de termes et la validité de celle ci considérée comme un atome du langage du premier ordre LE. Cela nous permettra, en utilisant la définition 1.43, de définir la validité d'une formule équationnelle dans une algèbre comme un prolongement de la validité d'une équation.

Définition 1.46 Validité d'une formule dans une algèbre

Soient f une formule équationnelle et $A = (D_A, F_A)$ une Σ_algèbre. Nous disons que f est valide dans A si et seulement si pour toute valuation $v : X \longrightarrow D_A \cup BOOL$, $v(f) = vrai$. Nous la notons par $A \models_\Sigma f$.

De la même manière qu'une équation, une formule f est valide dans une classe d'algèbres C si et seulement si elle est valide dans chaque algèbre de C.

Définition 1.47 Validité équationnelle d'une formule

Une formule f est équationnellement valide dans une spécification $SP = (S, F, E)$ si et seulement si elle est valide dans chaque algèbre de la variété $M(E)$. Nous pouvons dire aussi que f est une conséquence équationnelle de SP ou de E, et nous la notons respectivement par $SP \models_\Sigma f$ et $E \models_\Sigma f$.

Définition 1.48 Validité inductive d'une formule équationnelle

Une formule f est inductivement valide dans une spécification $SP = (S, C \cup D, E)$ en respectant les constructeurs C si et seulement si elle est valide dans chaque algèbre C_finiment engendrée de $M(E)$. Nous la notons $SP \models\!\!\models_\Sigma f$ ou $E \models\!\!\models_\Sigma f$.

Comme dans le cas d'une équation simple pour prouver la validité inductive d'une formule il suffit de prouver la validité de celle ci dans l'algèbre des constructeurs $(T(C)_{/=_E}, F)$ qui est l'algèbre initiale de la classe des alg'bres C_finiment engendrées. Pour prouver cette validité il faut et il suffit de prouver que toutes instance closes sur les constructeurs de cette formule soit équationnellement valide:

Proposition 1.5

Une formule équationnelle f est inductivement valide dans une spécification $SP = (S, C \cup D, E)$ si et seulement si pour toute substitution close sur les constructeurs $\sigma : X \longrightarrow T(C)$, $\sigma(f)$ est équationnellement valide dans SP.

6 Conclusion

Dans ce chapitre nous avons essayé de donner progressivement une sémantique formelle au problème de validité d'une propriété élémentaire dans une spécification. Ainsi, nous avons défini la validité équationnelle et inductive des équations simples dans des spécifications simples puis conditionnelles. Nous avons vu que cette validité formelle pourrait se transformer à une validité syntaxique. Nous avons terminé par les définitions formelles de la validité équationnelle et inductive d'une formule équationnelle. La question qui se pose maintenant est: est ce que ces systèmes de déduction que nous avons définies pour les équations simples peuvent se prolonger aux formules équationnelles? Les chapitre 2 et 3 auront comme objectif respectif de répondre à cette question dans le cas de la validité équationnelle et de la validité inductive.

Chapitre 2

Décomposition et validité équationnelle

1 Introduction

Dans le chapitre 1 nous avons défini une méthode syntaxique pour tester la validité d'une équation dans une variété d'algèbres engendrées par un ensemble d'équations. La question que nous nous posons ici, est de savoir si il existe une méthode similaire pour tester la validité d'une formule équationnelle. Les travaux qui existent actuellement répondent à cette question dans des cas particuliers de formules sous forme d'implication $e_1 \supset e_2$, comme dans, par exemple, LP [GG89]. Des formes plus générales de formules équationnelles sont traitées dans [Gog88].

Ce que nous allons proposer dans ce chapitre, c'est une méthode plus générale pour traiter une classe plus grande de formules. Cette classe sera déterminée par la possibilité ou non d'appliquer cette méthode à une formule équationnelle donnée et par la possibilité d'utiliser des règles de déduction simple pour prouver sa validité. Cette méthode consiste à essayer de ramener la preuve de validité d'une formule à la preuve de validité de certaines équations qui la composent. Pour cela nous allons essayer de transformer la formule à prouver sous une forme la moins complexe et la plus maniable possible, sur laquelle nous pourrons appliquer la même méthode de preuve que dans le cas d'une équation simple. La validité équationnelle d'une formule va donc se ramener à la validité équationnelle de certaines équations qui la composent. Les autres équations vont servir comme hypothèses pour cette preuve.

L'objectif de ce chapitre est de décrire la stratégie qui nous permettra de passer d'une formule donnée à un ensemble d'équations tel que la preuve de validité de ces équations soit suffisante pour réaliser la preuve de validité de la formule. Cette stratégie est basée sur deux principes. Le premier, souvent utilisé dans la logique du premier ordre, qui dit que pour prouver une formule universellement quantifiée il suffit de fixer ses variables et les considérer comme de nouvelles constantes, puis de prouver la formule résultante. Le deuxième que chacun d'entre nous emploie chaque fois qu'il est confronté à un problème complexe. Il s'agit de la *décomposition*. Cela consiste à diviser le problème initial en plusieurs sous problèmes avec des contextes différents, tels que le regroupement des solutions de ces sous problèmes soit équivalent à la solution du problème initial.

Dans ce cadre nous allons définir une structure qui va être le support de cette stratégie de décomposition. Nous donnerons ensuite une sémantique à cette structure. Puis nous décrirons cette stratégie sous forme de règles d'inférences et de tactiques. La preuve de correction de ces règles d'inférences nous permettra de prouver la consistance de cette méthode. Nous donnerons ensuite la classe de formules sur lesquelles nous pourrons appliquer cette méthode. Enfin nous verrons comment se ramener à la validité équationnelle de quelques équations seulement.

2 Passage d'une formule avec variables à une formule close

Nous avons signalé qu'une formule équationnelle est une formule supposée être universellement quantifiée. Or en logique du premier ordre, nous savons que la clôture universelle d'une formule est valide si et seulement si cette formule est valide. Cela veut dire que les variables libres de cette formule sont considérées comme de nouvelles constantes. L'objectif de ce paragraphe est de pouvoir étendre ce raisonnement aux formules équationnelles. Autrement dit, nous voulons prouver la validité équationnelle d'une formule équationnelle nouvelle en considérant ses variables comme de nouvelles constantes.

Convention 4

Soit $\Sigma = (S, F)$ une signature et V un ensemble de constantes de sorte dans S, alors $\Sigma \cup V$ désignera la signature $(S, F \cup V)$.

Commençons d'abord par établir ce résultat dans le cas des équations simples.

Théorème 2.1

Soient $SP = (S, F, E)$ une spécification équationnelle, $t_1 == t_2$ une équation simple et V l'ensemble des variables de $t_1 == t_2$. Alors

$$E \models_\Sigma t_1 == t_2 \text{ si et seulement si } E \models_{\Sigma \cup V} \bar{t_1} == \bar{t_2}.$$

Où $\bar{t_1} == \bar{t_2}$ est l'équation close obtenue à partir de $t_1 == t_2$ en considérant ses variables comme de nouvelles constantes.

Ce théorème a été initialement établi dans le cas de la logique équationnelle dans [Gog88]. Pour faciliter sa preuve nous avons besoin de quelques remarques.

Remarque 2.1

- *Les termes de $T(\Sigma, V)$ peuvent être considérés comme des termes clos de $T(\Sigma \cup V)$.*
- *Soient $A = (D_A, F_A)$ une Σ_algèbre et $h : V \longrightarrow D_A$ une interprétation des variables de V dans D_A, alors*
  - *A peut être considérée comme une $(\Sigma \cup V)$_algèbre en utilisant l'application h pour prolonger la fonction d'interprétation I de A,*
  - *il existe une extension unique de h à un $(\Sigma \cup V)$_homomorphisme $\bar{h} : T(\Sigma \cup V) \longrightarrow D_A$.*

Preuve :
En prenant compte les remarques précédentes, les deux conditions du théorèmes sont équivalentes à la condition suivante :
Pour toute $(\Sigma \cup V)$_algèbre A validant E et pour toute fonction $h : V \longrightarrow D_A$, $\bar{h}(t_1) = \bar{h}(t_2)$, où $\bar{h}$ est l'unique homomorphisme prolongeant h.

Ce théorème peut facilement être prolongé aux formules équationnelles en utilisant la proposition 1.4 et la définition 1.46.

Théorème 2.2

Soient $SP = (S, F, E)$ une spécification équationnelle, f une formule équationnelle et V l'ensemble des variables de f. Alors
$E \models_\Sigma f$ si et seulement si $E \models_{\Sigma \cup V} \bar{f}$. Où $\bar{f}$ est la formule équationnelle close obtenue à partir de f en considérant ses variables comme de nouvelles constantes et $\Sigma = (S, F)$.

Remarque 2.2

Ce passage des formules équationnelle avec variables aux formules closes n'est utilisable, comme il est signalé dans le théorème précédent, que dans le cas de la validité équationnelle. Cette technique va dans le sens contraire de la signification de la validité inductive. En effet cette dernière consiste à tester la validité pour toutes les instances des variables et non pas les considérer comme des constantes.

3 Formule avec assertion

Dans ce paragraphe, il s'agit de définir la structure qui sera le support de la stratégie de décomposition.

Définition 2.1 Formule avec assertion

Soient $a_1, \ldots, a_n$ et f des formules équationnelles **closes**. *Une formule avec assertion, appelée dans la suite de cette thèse* a_formule *et notée $(\{a_1, \ldots, a_n\}, f)$, est constituée d'une partie assertion $\{a_1, \ldots, a_n\}$ et d'une partie conclusion f.*
Nous parlerons de a_équation *dans le cas où $a_1, \ldots, a_n$ et f sont des équations simples.*

Exemple 2.1

On considère la spécification du type ensemble ordonné de l'exemple 1.13.
$(\{estvide(E) == faux\}, min(E) \leq max(E) == vrai)$ est une a_équation.

La sémantique intuitive d'une a_formule $(\{a_1, \ldots, a_n\}, f)$ est que $\{a_1, \ldots, a_n\}$ sont des hypothèses et que f est la conclusion à prouver en utilisant ces hypothèses.

Définition 2.2 Sémantique d'une a_formule

Soient A une Σ_algèbre et $(\{a_1, \ldots, a_n\}, f)$ une a_formule. $(\{a_1, \ldots, a_n\}, f)$ est valide dans A et nous notons $A \models_\Sigma (\{a_1, \ldots, a_n\}, f)$, si et seulement si la formule équationnelle $(\wedge_{i \in [1..n]} a_i)$ est valide dans A implique *f est valide dans A.*
$A \models_\Sigma (\{a_1, \ldots, a_n\}, f)$ si et seulement si $A \models (\wedge_{i \in [1..n]} a_i)$ implique $A \models_\Sigma f$.

La proposition suivante nous permet de passer d'une formule close à la a_formule associée.

Proposition 2.1

Soit E un ensemble d'équations et f une formule équationnelle. f est valide dans la variété d'algèbres $M(E)$ engendrées par E si et seulement si la a_formule $(\{\}, f)$ est valide dans $M(E)$.

Preuve :
$(\{\}, f)$ est valide dans une algèbre A si et seulement si *vrai* est valide dans A implique f est valide dans A. Ce qui est équivalent à f est valide dans A.

Maintenant que nous avons donné la définition et la sémantique d'une a_formule, nous allons décrire la stratégie de décomposition qui permet de passer d'une formule close à un ensemble de a_équations équivalentes. La proposition précédente amorce ce processus en transformant une formule donnée en une a_formule avec un ensemble d'assertions vide.

4 Décomposition

La décomposition des formules logiques dans un but de les prouver ou de les falsifier a été utilisée depuis longtemps. Ainsi, Gentzen [Gen55] avait proposé un système de déduction basé sur des règles d'inférences qui permettent de décomposer les formules logiques et de construire leur arbre de preuve ou leur arbre de contre exemple. Nous pouvons retrouver ces idées aussi chez plusieurs auteurs qui ont adopté la démarche de Gentzen, comme par exemple [Gal86]. Cette démarche est appelée aussi la méthode des tableaux ou des séquents.

Pour décrire notre stratégie de décomposition, nous allons nous aussi utiliser des règles d'inférences similaires mais adaptées à notre structure de a_formule et à notre contexte qui est la preuve dans le cadre équationnel. La description de cette stratégie devient ainsi très facile parce que ces règles reflètent clairement la sémantique des connecteurs logiques. Elles permettent aussi la visualisation de la décomposition étape par étape. La preuve de correction de la stratégie se réduit à la preuve de correction de chacune d'elles.

Ces règles d'inférences qui manipulent des a_formules, peuvent être classées en deux catégories. Les premières opèrent sur la partie conclusion des a_formules. Les autres opèrent sur la partie assertion. Pour chaque règle, le dénominateur ou la *conclusion de la règle* est une conséquence logique du numérateur. Le dénominateur contient une seule a_formule, tandis que le numérateur peut contenir plusieurs a_formules. La définition suivante permet de donner une sémantique à ces règles d'inférences. En plus elle définit la notion de correction qui sera utilisée dans le paragraphe suivant.

Définition 2.3 Sémantique d'une règle d'inférence de décomposition

Soit $\{(\{a_{1_1}, \ldots, a_{1_{m_1}}\}, f_1), \ldots, (\{a_{n+1_1}, \ldots, a_{n+1_{m_{n+1}}}\}, f_{n+1})\}$ un ensemble de a_formules.

- *La règle d'inférence*

$$\frac{\begin{array}{c}\langle\{a_{1_1}, \ldots, a_{1_{m_1}}\}, f_1\rangle \\ \vdots \\ \langle\{a_{n_1}, \ldots, a_{n_{m_n}}\}, f_n\rangle\end{array}}{\langle\{a_{n+1_1}, \ldots, a_{n+1_{m_{n+1}}}\}, f_{n+1}\rangle}$$

est correcte si et seulement si pour toute Σ_algèbre A,
$\langle\{a_{n+1_1}, \ldots, a_{n+1_{m_{n+1}}}\}, f_{n+1}\rangle$ est valide dans A si et seulement si $\langle\{a_{1_1}, \ldots, a_{1_{m_1}}\}, f_1\rangle$ et ... et $\langle\{a_{n_1}, \ldots, a_{n_{m_n}}\}, f_n\rangle$ sont toutes valides dans A.

- *La règle d'inférence*

$$\frac{\langle\{a_{1_1}, \ldots, a_{n_{m_1}}\}, f_1\rangle}{\langle\{a_{n+1_1}, \ldots, a_{n+1_{m_{n+1}}}\}, f_{n+1}\rangle} \quad \vee \ldots \vee \quad \frac{\langle\{a_{n_1}, \ldots, a_{n_{m_n}}\}, f_n\rangle}{\langle\{a_{n+1_1}, \ldots, a_{n+1_{m_{n+1}}}\}, f_{n+1}\rangle}$$

est correcte si et seulement si pour toute Σ_algèbre A,
$\langle\{a_{n+1_1}, \ldots, a_{n+1_{m_{n+1}}}\}, f_{n+1}\rangle$ est valide dans A si et seulement si il existe un i tel que $\langle\{a_{i_1}, \ldots, a_{n_{m_i}}\}, f_i\rangle$ est valide dans A.

Le "si et seulement si" permet d'assurer que si la formule initiale est valide alors les a_équations résultats le seront aussi.

4.1 Décomposition de la partie conclusion

Comme nous l'avons signalé dans l'introduction de ce chapitre, notre objectif est de pouvoir réduire la preuve d'une formule équationnelle à la preuve de certaines équations qui la composent. Pour cela nous allons utiliser les règles d'inférences de la figure 1 pour décomposer les formules de la partie conclusion jusqu'à ce que nous obtenons des équations simples.

1. $$\frac{\begin{array}{c}\langle\{a_1, \ldots, a_n\}, f_1\rangle \\ \langle\{a_1, \ldots, a_n\}, f_2\rangle\end{array}}{\langle\{a_1, \ldots, a_n\},\ f_1 \wedge f_2\rangle}$$

2. $$\frac{\langle\{a_1, \ldots, a_n\}, f_1\rangle}{\langle\{a_1, \ldots, a_n\}, f_1 \vee f_2\rangle} \quad \vee \quad \frac{\langle\{a_1, \ldots, a_n\}, f_2\rangle}{\langle\{a_1, \ldots, a_n\}, f_1 \vee f_2\rangle}$$

3. $$\frac{\langle\{a_1, \ldots, a_n, f_1\}, f_2\rangle}{\langle\{a_1, \ldots, a_n\}, f_1 \supset f_2\rangle}$$

Figure 1 : Décomposition de la partie conclusion

Ces règles de décomposition peuvent être lu de la manière suivante :

1. Pour prouver la formule close $f_1 \wedge f_2$ sous les hypothèses closes $\{a_1, \ldots, a_n\}$ il faut et il suffit de prouver la formule f_1 sous les hypothèses $\{a_1, \ldots, a_n\}$ et de prouver f_2 sous les mêmes hypothèses.

2. Pour prouver la formule close $f_1 \vee f_2$ sous les hypothèses closes $\{a_1, \ldots, a_n\}$ il faut et il suffit de prouver la formule f_1 sous les hypothèses $\{a_1, \ldots, a_n\}$ ou de prouver f_2 sous les mêmes hypothèses. Cela n'est valable bien sûr que pour les formules closes.

3. Pour prouver la formule close $f_1 \supset f_2$ sous les hypothèses closes $\{a_1, \ldots, a_n\}$ il faut et il suffit de prouver la formule f_2 sous les hypothèses $\{a_1, \ldots, a_n\}$ plus la formule f_1.

4.2 Décomposition de la partie assertion

Au moment de la décomposition d'une formule de la partie conclusion il y a certaines parties de celle ci qui passent dans la partie assertions. C'est le cas quand nous utilisons la troisième règle de la figure 1. Ces parties vont donc être utilisées comme des hypothèses pour prouver la partie conclusion. Or, comme nous allons le voir dans le paragraphe 7 et dans les chapitres suivants, pour pouvoir exploiter ces hypothèses il faut les transformer sous forme d'équations simples. Cette transformation est illustrée par les règles de décomposition de la figure 2.

$$\frac{\langle \{a_1, \ldots, a_n, a, b\}, f\rangle}{\langle \{a_1, \ldots, a_n, a \wedge b\}, f\rangle}$$

$$\frac{\langle \{a_1, \ldots, a_n, a\}, f\rangle \quad \langle \{a_1, \ldots, a_n, b\}, f\rangle}{\langle \{a_1, \ldots, a_n, a \vee b\}, f\rangle}$$

$$\frac{\langle \{a_1, \ldots, a_n, \neg a\}, f\rangle \quad \langle \{a_1, \ldots, a_n, a, b\}, f\rangle}{\langle \{a_1, \ldots, a_n, a \supset b\}, f\rangle}$$

Figure 2: Décomposition de la partie assertion

Ces règles de décomposition de la partie assertion peuvent être lues :

1. Pour prouver la validité d'une formule équationnelle close f sous les hypothèses closes $\{a_1, \ldots, a_n, a \wedge b\}$ il faut et il suffit de la prouver sous les hypothèses $\{a_1, \ldots, a_n, a, b\}$.

2. Pour prouver la validité d'une formule équationnelle close f sous les hypothèses closes $\{a_1, \ldots, a_n, a \vee b\}$ il faut et il suffit de la prouver sous les hypothèses $\{a_1, \ldots, a_n, a\}$, puis de la prouver sous les hypothèses $\{a_1, \ldots, a_n, b\}$.

3. Pour prouver la validité d'une formule équationnelle close f sous les hypothèses closes $\{a_1, \ldots, a_n, a \supset b\}$ il faut et il suffit de la prouver sous les hypothèses $\{a_1, \ldots, a_n, \neg a\}$, puis de la prouver sous les hypothèses $\{a_1, \ldots, a_n, a, b\}$.

Remarque 2.3

Prouver une formule f sous une hypothèse $a \supset b$ est équivalent à la prouver sous l'hypothèse "$\neg a$" puis sous l'hypothèse "b". Mais notre objectif est d'avoir le maximum d'hypothèses possible. C'est pourquoi sur la troisième règle de décomposition de la partie assertion, nous avons ajouter l'hypothèse "a" à côté de l'hypothèse "b" dans la partie premisse de la première a_formule de cette règle. Cela n'a pas d'influence sur la correction de la règle comme nous allons le voir dans le paragraphe suivant.

4.3 Forme normale négative

D'après la définition d'une formule équationnelle, celle-ci peut contenir l'opérateur de négation $\neg$. Donc, nous serons amené à décomposer de telles formules. Et cela peut concerner indifféremment la partie assertion ou la partie conclusion d'une a_formule. Ainsi, au lieu de faire des règles de décomposition pour les assertions et d'autres pour les conclusions qui contiennent des opérateurs de négation en racine, il est préférable de regrouper cela sous forme de règles qui seront utilisées pour les deux parties. Le but est de propager l'opérateur de négation de la racine vers les feuilles de telle sorte qu'il ne porte à la fin que sur des équations simples. La formule résultat est dite sous forme normale négative. Ce processus sera appliqué chaque fois que nous serons amené à décomposer une a_formule contenant une formule avec le symbole de négation comme symbole de racine. Cela peut concerner la formule initiale ou une formule résultat de l'étape de décomposition précédente.

Définition 2.4 Forme normale négative

La forme normale négative (FNN) d'une formule peut être définie par :

- *Si E est une équation, alors E et $\neg E$ sont sous forme normale négative.*

- *Si A et B sont deux formules sous forme normale négative alors $(A \vee B)$ et $(A \wedge B)$ sont sous forme normale négative.*

Pour calculer cette forme normale négative nous utilisons récursivement les règles d'inférence appropriées de la figure 3. Ces règles d'inférence qui manipulent des formules sont la traduction naturelle des règles de Morgan auxquelles nous ajoutons deux règles pour traiter les équations

booléennes.

1. $\dfrac{(\neg f_1 \wedge \neg f_2)}{\neg(f_1 \vee f_2)}$

2. $\dfrac{(\neg f_1 \vee \neg f_2)}{\neg(f_1 \wedge f_2)}$

3. $\dfrac{(f_1 \wedge \neg f_2)}{\neg(f_1 \Rightarrow f_2)}$

4. $\dfrac{f_1}{\neg\neg f_1}$

5. $\dfrac{t == vrai}{\neg(t == faux)}$

6. $\dfrac{t == faux}{\neg(t == vrai)}$

Figure 3 :
Forme normale négative

4.4 Algorithme de décomposition

Procédure **decomp_conc**$(\langle\{a_1, \ldots, a_n\}, f\rangle)$

cas $f = (t_1 == t_2)$
  alors $\langle\{a_1, \ldots, a_n\}, f\rangle$
$f = \neg f_1$
  alors *decomp_conc*$(\langle\{a_1, \ldots, a_n\}, FNN(f)\rangle)$,
$f = f_1 \wedge f_2$
  alors noeud_et(decomp_conc$(\langle\{a_1, \ldots, a_n\}, f_1\rangle)$, decomp_conc$(\langle\{a_1, \ldots, a_n\}, f_2\rangle)$),
$f = f_1 \vee f_2$
  alors noeud_ou(decomp_conc$(\langle\{a_1, \ldots, a_n\}, f_1\rangle)$, decomp_conc$(\langle\{a_1, \ldots, a_n\}, f_2\rangle)$),
$f = f_1 \supset f_2$
  alors *decomposer_a*$(f_1, $*decomp_conc*$(\langle\{a_1, \ldots, a_n\}, f_2\rangle))$.
fin_cas
fin

Procédure **decomposer_a**(f_1, l)
f_1 est une formule équationnelle et l est un arbre et – ou de a_formules

cas $l = \langle\{a_1, \ldots, a_n\}, f\rangle$
  alors *decomp_assert*$(\langle\{a_1, \ldots, a_n, f_1\}, f\rangle)$
$l = $ *noeud_et*(l_1, l_2)
  alors *noeud_et*(*decomposer_a*(f_1, l_1), *decompser_*(f_1, l_2))
$l = $ *noeud_ou*(l_1, l_2)
  alors *noeud_ou*(*decomposer_a*(f_1, l_1), *decompser_*(f_1, l_2))
fin_cas
fin

Procédure **decomp_assert**$(\langle\{a_1, \ldots, a_n, f\}, f'\rangle)$

cas $f = (t_1 == t_2)$
  alors $\langle\{a_1, \ldots, a_n, f\}, f'\rangle$
$f = \neg f_1$
  alors *decomp_assert*$(\langle\{a_1, \ldots, a_n, FNN(f_1)\}, f'\rangle)$
$f = f_1 \wedge f_2$
  alors *decomposer_a*$(f_1, $*decomp_assert*$(\langle\{a_1, \ldots, a_n, f_2\}, f'\rangle))$
$f = f_1 \vee f_2$
  alors *noeud_et*(*decomp_assert*$(\langle\{a_1, \ldots, a_n, f_1\}, f'\rangle)$,
    decomp_assert$(\langle\{a_1, \ldots, a_n, f_2\}, f'\rangle)$)
$f = f_1 \supset f_2$
  alors *noeud_et*(*decomp_assert*$(\langle\{a_1, \ldots, a_n, FNN(f_1)\}, f'\rangle)$,
    decomp_assert$(\langle\{a_1, \ldots, a_n, f_1 \wedge f_2\}, f'\rangle)$)
fin_cas
fin

Procedure **FNN**(f)
f est une formule équationnelle

cas $f = (t == vrai)$
  alors $t == faux$
$f = (t == faux)$
  alors $t == vrai$
$f = f_1 \wedge f_2$
  alors $FNN(f_1) \vee FNN(f_2)$
$f = f_1 \vee f_2$
  alors $FNN(f_1) \wedge FNN(f_2)$
$f = f_1 \supset f_2$
  alors $f_1 \wedge FNN(f_2)$
autres
  alors f
fin_cas
fin

4.5 Exemple de décomposition

Soit a, b, c, d et e quatre équations closes, et $(((b \wedge c) \supset d) \supset (a \wedge e))$ une formule à prouver. Nous commençons la décomposition avec la a_formule $\langle\{\}, (((b \wedge c) \supset d) \supset (a \wedge e))\rangle$.

L'application de la règle d'inférence 3 de la figure 1 donne: $\langle\{((b \wedge c) \supset d)\}, (a \wedge e)\rangle$.
L'application de la règle 3 de la figure 2 donne: $\langle\{\neg(b \wedge c)\}, (a \wedge e)\rangle$ et $\langle\{(b \wedge c), d\}, (a \wedge e)\rangle$
L'application de la règle 1 de la figure 2 sur la première a_formule donne: $\langle\{\neg b\}, (a \wedge e)\rangle$ et $\langle\{\neg c\}, (a \wedge e)\rangle$.
L'application de la règle 2 sur $\langle\{(b \wedge c), d\}, (a \wedge e)\rangle$ donne: $\langle\{b, c, d\}, (a \wedge e)\rangle$.
L'application de la règle 1 de la figure 2 sur les trois a_formules résultats donne:
$\langle\{\neg b\}, a\rangle$, $\langle\{\neg c\}, e\rangle$, $\langle\{\neg b\}, e\rangle$, $\langle\{\neg c\}, a\rangle$, $\langle\{b, c, d\}, a\rangle$ et $\langle\{b, c, d\}, e\rangle$.
Donc prouver la validité de la a_formule initiale revient à prouver indépendamment la validité de ces six a_équations.

5 Correction des règles de décomposition

Cette preuve de correction des règles d'inférence a pour objectif d'assurer la consistance et le bon fondement de notre stratégie de décomposition. Ainsi nous pouvons conclure que si une formule équationnelle close est décomposable en un ensemble de a_équations alors si cette formule est valide dans une algèbre A alors toutes les a_équations associées sont valides dans A et réciproquement si toutes les a_équations sont valides dans A alors la formule initiale l'est aussi. Toutefois il faudrait signaler que cette décomposition n'est applicable que sur la classe de formules que nous allons voir dans le paragraphe 6.
Nous allons commencer par énoncer quelques lemmes du calcul des propositions qui seront utilisés pour prouver la correction des règles de décomposition.

Convention 5

Dans toute la suite de cette thèse, lorsque nous parlerons d'une a_formule ou d'une a_équation $\langle\{a_1, \ldots, a_n\}, f\rangle$, les formules ou les équations $a_1, \ldots, a_n, f$ seront considérées comme des formules closes. La signature Σ contiendra éventuellement leurs variables fixées.

5.1 Lemmes de calcul des propositions

Lemme 2.1 *Soient a, b, c et d quatre propositions.*

$$a \supset (b \wedge c) \Leftrightarrow (a \supset b) \wedge (a \supset c)$$

Lemme 2.2

$$a \supset (b \supset c) \Leftrightarrow (a \wedge b) \supset c$$

Lemme 2.3

$$(a \vee b) \supset c \Leftrightarrow (a \supset c) \wedge (b \supset c)$$

Lemme 2.4

$$(a \wedge (b \supset c)) \Leftrightarrow (a \wedge \neg b) \wedge (a \wedge b \wedge c)$$

Lemme 2.5

$$(a \supset (b \vee c)) \Leftrightarrow ((a \supset b) \vee (a \supset c))$$

5.2 Preuve de correction des règles d'inférences

preuve de correction des règles de décomposition de la partie conclusion

Proposition 2.2 *La règle d'inférence:*

$$\frac{\langle\{a_1, \ldots, a_n\}, f_1\rangle \quad \langle\{a_1, \ldots, a_n\}, f_2\rangle}{\langle\{a_1, \ldots, a_n\},\ f_1 \wedge f_2\rangle}$$

est correcte.

Preuve: Soit A une Σ_algèbre.

$A \models_\Sigma \langle\{a_1, \ldots, a_n\},\ f_1 \wedge f_2\rangle$

$$\begin{array}{l}
\Longleftrightarrow A \models_\Sigma (\wedge_{i\in[1..n]} a_i) \Longrightarrow A \models_\Sigma (f_1 \wedge f_2)\ , \\
\Longleftrightarrow A \models_\Sigma (\wedge_{i\in[1..n]} a_i) \Longrightarrow (A \models_\Sigma f_1 \wedge A \models_\Sigma f_2)\ , \\
\Longleftrightarrow A \models_\Sigma (\wedge_{i\in[1..n]} a_i) \Longrightarrow A \models_\Sigma f_1 \wedge \\
\quad\ A \models_\Sigma (\wedge_{i\in[1..n]} a_i) \Longrightarrow A \models_\Sigma f_2 \text{ (Lemme 2.1)} \\
\Longleftrightarrow A \models_\Sigma \langle\{a_1, \ldots, a_n\}, f_1\rangle \wedge A \models_\Sigma \langle\{a_1, \ldots, a_n\}, f_2\rangle
\end{array}$$

Proposition 2.3 *La règle d'inférence:*

$$\frac{\langle\{a_1, \ldots, a_n\}, f_1\rangle}{\langle\{a_1, \ldots, a_n\}, f_1 \vee f_2\rangle} \quad \vee \quad \frac{\langle\{a_1, \ldots, a_n\}, f_2\rangle}{\langle\{a_1, \ldots, a_n\}, f_1 \vee f_2\rangle}$$

est correcte.

Preuve:

Soit A une Σ_algèbre.
$A \models_\Sigma \langle\{a_1, \ldots, a_n\}, f_1 \vee f_2\rangle$

$$\begin{array}{l}
\Longleftrightarrow A \models_\Sigma (\wedge_{i\in[1..n]} a_i) \Longrightarrow A \models_\Sigma (f_1 \vee f_2), \\
\Longleftrightarrow A \models_\Sigma (\wedge_{i\in[1..n]} a_i) \Longrightarrow (A \models_\Sigma f_1 \ \vee A \models_\Sigma f_2) \\
\Longleftrightarrow (A \models_\Sigma (\wedge_{i\in[1..n]} a_i) \Longrightarrow A \models_\Sigma f_1) \text{ ou} \\
\quad (A \models_\Sigma (\wedge_{i\in[1..n]} a_i) \Longrightarrow A \models_\Sigma f_1) \text{ (Lemme 2.5)} \\
\Longleftrightarrow A \models_\Sigma \langle\{a_1, \ldots, a_n\}, f_1\rangle \text{ ou } A \models_\Sigma \langle\{a_1, \ldots, a_n\}, f_2\rangle.
\end{array}$$

Proposition 2.4 *La règle d'inférence:*

$$\frac{\langle\{a_1, \ldots, a_n, f_1\}, f_2\rangle}{\langle\{a_1, \ldots, a_n\}, f_1 \supset f_2\rangle}$$

est correcte.

Preuve:

Soit A une Σ_algèbre. $A \models_\Sigma \langle\{a_1, \ldots, a_n\}, f_1 \supset f_2\rangle$

$\iff A \models_\Sigma (\wedge_{i\in[1..n]} a_i) \Longrightarrow A \models_\Sigma (f_1 \supset f_2)$,
$\iff A \models_\Sigma (\wedge_{i\in[1..n]} a_i) \Longrightarrow (A \models_\Sigma f_1 \Longrightarrow A \models_\Sigma f_2)$,
$\iff (A \models_\Sigma (\wedge_{i\in[1..n]} a_i) \wedge A \models_\Sigma f_1) \Longrightarrow A \models_\Sigma f_2$ (Lemme 2.2),
$\iff (A \models_\Sigma (\wedge_{i\in[1..n]} a_i \wedge f_1)) \Longrightarrow A \models_\Sigma f_2$
$\iff A \models_\Sigma \langle\{a_1, \ldots, a_n, f_1\}, f_2\rangle$

Correction des règles de décomposition de la partie assertion

Proposition 2.5 *La règle d'inférence :*

$$\frac{\langle\{a_1, \ldots, a_n, a, b\}, f\rangle}{\langle\{a_1, \ldots, a_n, a \wedge b\}, f\rangle}$$

est correcte.

Preuve : évidente

Proposition 2.6 *La règle d'inférence :*

$$\frac{\langle\{a_1, \ldots, a_n, a\}, f\rangle \quad \langle\{a_1, \ldots, a_n, b\}, f\rangle}{\langle\{a_1, \ldots, a_n, a \vee b\}, f\rangle}$$

est correcte.

Preuve :

$A \models_\Sigma \langle\{a_1, \ldots, a_n, a \vee b\}, f\rangle$

$\iff A \models_\Sigma (\wedge_{i\in[1..n]} a_i \wedge (a \vee b)) \Longrightarrow A \models_\Sigma f$,
$\iff A \models_\Sigma ((\wedge_{i\in[1..n]} a_i \wedge a) \vee (\wedge_{i\in[1..n]} a_i \wedge b))$
$\Longrightarrow A \models_\Sigma f$ (Distributivité du $\wedge$ par rapport à $\vee$),
$\iff (A \models_\Sigma (\wedge_{i\in[1..n]} a_i \wedge a) \vee A \models_\Sigma (\wedge_{i\in[1..n]} a_i \wedge b)) \Longrightarrow A \models_\Sigma f$,
$\iff (A \models_\Sigma (\wedge_{i\in[1..n]} a_i \wedge a) \Longrightarrow A \models_\Sigma f) \wedge$
$(A \models_\Sigma (\wedge_{i\in[1..n]} a_i \wedge b) \Longrightarrow A \models_\Sigma f)$ (Lemme 2.3),
$\iff A \models_\Sigma \langle\{a_1, \ldots, a_n, a\}, f\rangle$ et $A \models_\Sigma \langle\{a_1, \ldots, a_n, b\}, f\rangle$

Proposition 2.7 *La règle d'inférence :*

$$\frac{\langle\{a_1, \ldots, a_n, \neg a\}, f\rangle \quad \langle\{a_1, \ldots, a_n, a, b\}, f\rangle}{\langle\{a_1, \ldots, a_n, a \supset b\}, f\rangle}$$

est correcte.

Preuve :

Soit A une Σ_algèbre. $A \models_\Sigma \langle\{a_1, \ldots, a_n, a \supset b\}, f\rangle$

$\iff A \models_\Sigma \wedge_{i\in[1..n]} a_i \wedge (a \supset b) \Longrightarrow A \models_\Sigma f$
$\iff A \models_\Sigma ((\wedge_{i\in[1..n]} a_i \wedge \neg a) \vee (\wedge_{i\in[1..n]} a_i \wedge a \wedge b)) \Longrightarrow A \models_\Sigma f$ (Lemme 2.4)
$\iff (A \models_\Sigma (\wedge_{i\in[1..n]} a_i \wedge \neg a) \vee A \models_\Sigma (\wedge_{i\in[1..n]} a_i \wedge a \wedge b)) \Longrightarrow A \models_\Sigma f$
$\iff (A \models_\Sigma (\wedge_{i\in[1..n]} a_i \wedge \neg a) \Longrightarrow A \models_\Sigma f) \vee$
$(A \models_\Sigma (\wedge_{i\in[1..n]} a_i \wedge a \wedge b) \Longrightarrow A \models_\Sigma f)$ (Lemme 2.3)
$\iff A \models_\Sigma \langle\{a_1, \ldots, a_n, \neg a\}, f\rangle$ et $A \models_\Sigma \langle\{a_1, \ldots, a_n, a, b\}, f\rangle$.

6 Classe de formules décomposables

Dans le paragraphe précédent nous avons établi la correction des règles d'inférence, ce qui nous a permis d'assurer la consistance de cette méthode de décomposition. Maintenant, nous allons essayer de déterminer la classe de formules décomposables et par conséquent susceptibles d'être prouvées. Le premier critère pour déterminer cette classe concerne la possibilité d'appliquer les règles de décompositions que nous avons définies précédemment. Pour répondre à ce critère il suffit de respecter les formes des formules utilisées dans les règles de décomposition. Le deuxième critère concerne le résultat de cette décomposition et la possibilité d'utiliser des règles de déduction simples pour pouvoir prouver la validité de ce résultat. En effet, le résultat que nous voulons obtenir est un ensemble de a_équations, car, comme nous allons le voir dans le paragraphe suivant, la validité de ces a_équations est équivalente à celle de leur partie conclusion. Mais il se peut qu'à la fin de la décomposition, nous obtenions des a_formules qui peuvent contenir une négation sur une équation, comme par exemple les a_formules $\langle\{a_1, \ldots, a_n, \neg(t_1 == t_2)\}, e\rangle$ ou $\langle\{a_1, \ldots, a_n\}, \neg(t_1 == t_2)\rangle$. Or, les méthodes existantes de résolution de diséquation (négation d'équation) dans un ensemble d'équations, [Com88,CL88,Bur88], se restreignent à des classes particulières d'ensembles d'équations. Pour remédier à ce problème nous allons restreindre les formules qui peuvent donner de tels résultats aux formules booléennes. Cela vient du fait que la négation d'une équation booléenne est bien définie (cf figure 3).

Nous allons tout d'abord définir des ensembles intermédiaires de formules qui seront utilisés pour définir la classe de formules décomposables.

Définition 2.5 Formule conjonctive

L'ensemble des formules conjonctives $\bar{C}$ est défini par :

1. *Soient t_1 et t_2 deux termes de $T(\Sigma, X)$ alors $t_1 == t_2$ appartient à $\bar{C}$,*
2. *Soient f_1 et f_2 deux formules de $\bar{C}$ alors $f_1 \wedge f_2$ appartient à $\bar{C}$.*

Définition 2.6 Formule disjonctive

L'ensemble des formules disjonctives $\bar{D}$ est défini par :

1. *Soit f une formule conjonctive de $\bar{C}$ alors f appartient à $\bar{D}$,*
2. *Soient f_1 et f_2 deux formules de $\bar{D}$ alors $f_1 \vee f_2$ appartient à $\bar{D}$.*

Définition 2.7 Formule booléenne

L'ensemble des formules booléennes $\bar{B}$ est défini par :

1. *Soient t un terme booléen de $T(\Sigma, X)$ et p une constante booléenne vrai ou faux, alors $t == p$ appartient à $\bar{B}$,*
2. *Soient f_1 et f_2 deux formules de $\bar{B}$ alors les formules $\neg f_1$, $f_1 \wedge f_2$, $f_1 \vee f_2$ et $f_1 \supset f_2$ appartiennent à $\bar{B}$.*

Définition 2.8 Formule implicative

L'ensemble des formules implicatives IMP est défini par:

1. *Soient f_1 une formule booléenne de $\bar{B}$ et f_2 une formule conjonctive de $\bar{C}$ alors $f_1 \supset f_2$ appartient à IMP,*
2. *Soient f_1 une formule booléenne de $\bar{B}$ et f_2 une formule disjonctive de $\bar{D}$ alors $f_1 \supset f_2$ appartient à IMP,*
3. *Soient $(f_1, f_2) \in IMP^2$ alors $f_1 \wedge f_2$ et $f_1 \vee f_2$ appartiennent à IMP.*

Définition 2.9 Formule décomposable

L'ensemble des formules décomposables $DECOMP$ est défini par:

1. *Soient t_1 et t_2 deux termes de $T(\Sigma, X)$ alors $t_1 == t_2$ appartient à $DECOMP$,*
2. *Soit f une formule booléenne de $\bar{B}$ alors f appartient à $DECOMP$,*
3. *Soient f_1 et f_2 deux formules de $DECOMP$ alors $f_1 \wedge f_2$ appartient à $DECOMP$,*
4. *Soient f_1 et f_2 deux formules de $DECOMP$ alors $f_1 \vee f_2$ appartient à $DECOMP$,*
5. *Soient f_1 une formule conjonctive de $\bar{C}$ et f_2 une formule de $DECOMP$ alors $f_1 \supset f_2$ appartient à $DECOMP$,*
6. *Soient f_1 une formule disjonctive de $\bar{D}$ et f_2 une formule de $DECOMP$ alors $f_1 \supset f_2$ appartient à $DECOMP$,*
7. *Soient f_1 une formule booléenne de $\bar{B}$ et f_2 une formule de $DECOMP$ alors $f_1 \supset f_2$ appartient à $DECOMP$.*
8. *Soient f_1 une formule implicative de IMP et f_2 une formule de $DECOMP$ alors $f_1 \supset f_2$ appartient à $DECOMP$.*

Remarque 2.4

Soit f une formule de $DECOMP$ alors f peut être de l'une des forme suivante:

- $t_1 == t_2$ *avec* $(t_1, t_2) \in T(\Sigma, X)$,
- $\neg f_1$ *avec* $f_1 \in \bar{B}$,
- $f_1 \wedge f_2$ *avec* $(f_1, f_2) \in DECOMP^2$,
- $f_1 \vee f_2$ *avec* $(f_1, f_2) \in DECOMP^2$,
- $f_1 \supset f_2$ *avec* $(f_1, f_2) \in \bar{B}^2$,
- $f_1 \supset f_2$ *avec* $f_1 \in \bar{B}$ *et* $f_2 \in DECOMP$,
- $f_1 \supset f_2$ *avec* $f_1 \in \bar{D}$ *et* $f_2 \in DECOMP$.

- $f_1 \supset f_2$ *avec* $f_1 \in IMP$ *et* $f_2 \in DECOMP$.

Remarque 2.5

Soit $(f_1 \supset f_2) \supset f_3$ la formule close associée à une formule de $DECOMP$. La décomposition superficielle de cette formule nous donne les deux a_formules suivantes: $\langle\{\neg f_1\}, f_3\rangle$ et $\langle\{f_1, f_2\}, f_3\rangle$. Supposons que f_1 ne soit pas une formule de $\bar{B}$. Dans ce cas sa forme normale négative contient des formules du type $\neg(t_1 == t_2)$. Ce qui implique qu'à la fin de la décomposition nous allons obtenir des a_formules du type $\langle\{a_1, \ldots, a_n, \neg(t_1 == t_2)\}, e\rangle$ et qui ne sont pas des a_équations. C'est pour cela que nous avons exigé dans la définition de $DECOMP$ que la partie gauche d'une implication soit ou bien une conjonction, une disjonction ou bien une formule booléenne de $\bar{B}$ ou bien une formule implicative de IMP.

Définition 2.10 Taille d'une formule

La taille d'une formule est égale au nombre d'équations qui la composent. Nous pouvons la définir récursivement par:

- $taille(t_1 == t_2) = 1$,
- $taille(\neg f) = taille(f)$,
- $taille(f_1 * f_2) = taille(f_1) + taille(f_2)$, *avec* $* \in \{\wedge, \vee, \supset\}$

Théorème 2.3

Toute formule équationnelle close associée à une formule qui appartient à $DECOMP$ est décomposable en un ensemble de a_équations.

La preuve de ce théorème nécessite une récurrence sur la taille de la formule à décomposer.

Preuve:

Soit f la formule close associée à une formule de $DECOMP$. Rappelons que la proposition 2.1, permet de transformer f en une a_formule $\langle\{\}, f\rangle$.
Maintenant nous allons prouver par récurrence sur la taille de f_1 que la a_formule $\langle\{\}, f\rangle$ peut se décomposer en un ensemble de a_équations.

- Supposons que la taille de f soit égale à 1, donc f est une équation simple que nous pouvons transformer en une a_équation.
- Supposons maintenant que toute formule de taille inférieure à n soit décomposable en un ensemble de a_équations. Et soit f une formule contenant n opérateurs.
  f peut être de la forme:
  - $f_1 \wedge f_2$. La a_formule correspondante $\langle\{\}, f_1 \wedge f_2\rangle$ peut se décomposer en appliquant la première règle de la figure 1 aux deux a_formules $\langle\{\}, f_1\rangle$ et $\langle\{\}, f_2\rangle$. Or la tailles de f_1 et de f_2 sont chacune inférieure à n. Nous pouvons donc appliquer l'hypothèse de récurrence et conclure que f est décomposable.
  - $f_1 \vee f_2$. La a_formule correspondante $\langle\{\}, f_1 \vee f_2\rangle$ peut se décomposer en utilisant la deuxième règle de la figure 1 aux deux a_formules $\langle\{\}, f_1\rangle$ ou $\langle\{\}, f_2\rangle$. Or la taille respective de f_1 et de f_2 est inférieure à n. Nous pouvons donc appliquer l'hypothèse de récurrence et conclure que f est décomposable.

- $\neg f_1$. Avant de décomposer cette formule, il faut calculer sa forme normale négative en utilisant une des règles de la figure 3. Nous obtenons une formule de type $f' \wedge f''$ ou $f' \vee f''$ qui nous fera revenir sur l'un des cas précédents,
- $f_1 \supset f_2$. La a_formule correspondante $\langle\{\}, f_1 \supset f_2\rangle$ peut se décomposer en utilisant la troisième règle de la figure 1 à la a_formule $\langle\{f_1\}, f_2\rangle$. Or la taille de f_2 est inférieure à n, ce qui implique que nous pouvons appliquer l'hypothèse de récurrence. Mais maintenant il faut faire une récurrence sur f_1.
  * Supposons que f_1 soit de taille 1, donc f_1 est une équation. Et nous pouvons conclure que $\langle\{f_1\}, f_2\rangle$ est décomposable.
  * Supposons maintenant que toute a_formule $\langle\{f_1'\}, f_2\rangle$ telle que f_1' est de taille inférieure à m, soit décomposable en un ensemble de a_équations. Et soit $\langle\{f_1\}, f_2\rangle$ une a_formule telle que la taille de f_1 soit égale à m. f_1 peut être de la forme:
    - $f_{1_1} \wedge f_{1_2}$, nous appliquons alors la première règle de décomposition de la figure 2 pour obtenir la a_formule $\langle\{f_{1_1}, f_{1_2}\}, f_2\rangle$. Or puisque la taille respective de f_{1_1} et de f_{1_2} est inférieure à m, alors nous pouvons appliquer l'hypothèse de récurrence et conclure que $\langle\{f_{1_1} \wedge f_{1_2}\}, f_2\rangle$ est décomposable.
    - $f_{1_1} \vee f_{1_2}$, nous appliquons alors la deuxième règle de décomposition de la figure 2 pour obtenir les deux a_formules $\langle\{f_{1_1}\}, f_2\rangle$ et $\langle\{f_{1_2}\}, f_2\rangle$. Or puisque la taille respective de f_{1_1} et de f_{1_2} est inférieure à m, alors nous pouvons appliquer l'hypothèse de récurrence et conclure que $\langle\{f_{1_1} \vee f_{1_2}\}, f_2\rangle$ est décomposable.
    - $\neg f_{1_1}$, nous pouvons alors appliquer l'une des règles de transformations en forme normale négative pour obtenir une formule du type $f' \wedge f''$ ou $f' \vee f''$ qui nous fera revenir sur l'un des deux cas précédents.
    - $f_{1_1} \supset f_{1_2}$ avec f_{1_1} et f_{1_2} deux éléments de $\bar{\mathrm{B}}$. Nous appliquons alors la troisième règle de décomposition de la figure 2 pour obtenir les deux a_formules $\langle\{\neg f_{1_1}\}, f_2\rangle$ et $\langle\{f_{1_1}, f_{1_2}\}, f_2\rangle$. $\neg f_{1_1}$ est décomposable puisque f_{1_1} est une formule booléene. Or puisque la taille respective de f_{1_1} et de f_{1_2} est inférieure à m, nous pouvons appliquer l'hypothèse de récurrence et conclure que $\langle\{f_{1_1} \supset f_{1_2}\}, f_2\rangle$ est décomposable.
    - $f_{1_1} \supset f_{1_2} \in IMP$. Nous appliquons alors la troisième règle de décomposition de la figure 2 pour obtenir les deux a_formules $\langle\{\neg f_{1_1}\}, f_2\rangle$ et $\langle\{f_{1_1}, f_{1_2}\}, f_2\rangle$. $\neg f_{1_1}$ est décomposable puisque f_{1_1} est une formule booléenee. Or puisque la taille respective de f_{1_1} et de f_{1_2} est inférieure à m, nous pouvons appliquer l'hypothèse de récurrence et conclure que $\langle\{f_{1_1} \supset f_{1_2}\}, f_2\rangle$ est décomposable.

Remarque 2.6

L'idée de tester la validité des formules équationnelles a été proposée dans [Gog88], mais sur une classe de formule plus restreinte. En effet, cette classe peut être déterminée par les règles 1, 2 et 4 de la définition 2.9 et peut se résumer aux formules conjonctives et aux implications ne contenant que des conjonctions dans leur partie gauche.

7 Conséquence de la décomposition sur la validité d'une formule

Dans l'introduction de ce chapitre nous avons prétendu que nous allons ramener la validité équationnelle d'une formule à la validité équationnelle de certaines équations qui la composent. L'objectif de ce paragraphe est d'éclaircir ce point en précisant quelles sont ces équations, puis de prouver le bon fondement de cette affirmation.

7.1 Conséquence sur la validité équationnelle

La décomposition d'une formule a permis d'aboutir à un ensemble de a_équations. Donc, la validité équationnelle de celle ci doit passer par la validité de chacune des a_équations correspondantes.

Définition 2.11 Validité équationnelle d'une a_équation

Soit E un ensemble d'équations. Une a_formule $\langle\{a_1, \ldots, a_n\}, f\rangle$ est valide dans la variété d'algèbres $M(E)$ si et seulement si elle est valide dans chaque algèbre de $M(E)$. Nous dirons aussi que $\langle\{a_1, \ldots, a_n\}, f\rangle$ est équationnellement valide dans E.

Nous pouvons remarquer que la définition de validité d'une a_équation est aussi difficile que celle d'une formule. Heureusement le théorème suivant nous permettra de passer de la validité équationnelle d'une a_équation à la validité équationnelle de l'équation de la partie conclusion. En effet, pour qu'une a_équation soit valide dans la variété d'algèbres $M(E)$, il faut et il suffit que la conclusion soit valide dans la variété d'algèbres engendrées par l'ensemble d'équations E augmenté de l'ensemble des assertions.

Théorème 2.4

Soient E un ensemble d'équations associé à une spécification (S, F, E).
Une a_équation $\langle\{a_1, \ldots, a_n\}, e\rangle$ est valide dans la variété d'algèbres $M(E)$ engendrée par E si et seulement si

- $\forall\ A \in M(E)$ $(\exists i \in [1..n]$ *tel que a_i n'est pas valide dans A),*
  ou
- *e est valide dans la variété d'algèbre $M(E \cup \{a_1, \ldots, a_n\})$.*

Preuve :

Rappelons que $A \in M(E) \Longleftrightarrow \forall e \in E, A \models_\Sigma e$

- $\Longrightarrow$ par l'absurde.
  Supposons qu'il existe une algèbre non triviale A telle que pour tout $i \in [1..n]$, a_i est valide dans A. Cela implique que $M(E \cup \{a_1, \ldots, a_n\})$ contient au moins une algèbre non triviale, car elle contient au moins cette algèbre A.
  Supposons aussi que e n'est pas valide dans $M(E \cup \{a_1, \ldots, a_n\})$.
  Cela veut dire qu'il existe une algèbre A telle que $A \in M(E \cup \{a_1, \ldots, a_n\})$ et $A \not\models_\Sigma e$. Cela veut dire que A valide chaque équation de E, valide $a_1, \ldots, a_n$ et ne valide pas e. Autrement dit, $\wedge_{i \in [1..n]} a_i$ est valide dans A et e n'est pas valide dans A. Ce qui contredit le fait que $\langle\{a_1, \ldots, a_n\}, e\rangle$ est valide dans $M(E)$.

- $\Leftarrow$

  Nous devons prouver ici que les deux tranches impliquent séparément que $\langle\{a_1,\ldots,a_n\},e\rangle$ est valide dans $M(E)$.

  - supposons que $\forall\ A\ \in M(E)\ (\exists i \in [1..n]$ tel que a_i n'est pas valide dans $A)$. Cela implique $(\forall\ A\ \in M(E)(\wedge_{i\in[1..n]}a_i$ est non valide dans $A)$. Cela implique $(\forall\ A\ \in M(E)(\wedge_{i\in[1..n]}a_i$ est valide dans $A)$ implique e est valide dans $A)$ est toujours vrai. Ce qui implique que $\langle\{a_1,\ldots,a_n\},e\rangle$ est valide dans $M(E)$.
  - Supposons que e est valide dans $M(E\cup\{a_1,\ldots,a_n\})$. Ce qui est équivalent à dire que pour toute algèbre A, Si A valide toutes les équations de E et A valide tous les $a_i\ i \in [1..n]$ alors A valide e.
    Soit A une algèbre de $M(E)$ deux cas peuvent se présenter :
    * il existe un $i \in [1..n]$ tel que A ne valide pas a_i. Dans ce cas A ne valide pas $\wedge_{i\in[1..n]}a_i$ et donc A valide $\langle\{a_1,\ldots,a_n\},e\rangle$,
    * A valide tous les a_i pour $i \in [1..n]$. Et puisque A valide aussi toutes les équations de E alors elle valide e. Dans ce cas aussi A valide $\langle\{a_1,\ldots,a_n\},e\rangle$.

Ce théorème nous a permis de passer de la preuve de validité équationnelle d'une a_équation à la preuve de validité équationnelle d'une équation. Or dans le chapitre précédent nous avons vu que la validité d'une équation peut se faire d'une manière syntaxique en utilisant des remplacements des termes par des égaux en utilisant la relation de E_égalité en une étape. Ainsi cette méthode syntaxique sera prolongée aux cas des a_équations.

Le processus de preuve de validité équationnelle d'une formule peut se résumer ainsi : Nous commençons par décomposer la formule close associée en un ensemble de a_équations. Puis pour chaque a_équation s'il n'existe pas d'assertion qui n'est pas valide, alors nous entamons la preuve de validité de la partie conclusion dans l'ensemble des équations initiales plus les équations de la partie assertion.

8 conclusion

Dans ce chapitre, nous avons vu comment prolonger la validité syntaxique définie dans le cas des équations simples au cas des formules équationnelles. Il s'agissait seulement de la validité équationnelle. En effet, le passage d'une formule équationnelle à la formule close associée n'est valable que dans ce cas et les règles de décomposition ne sont correctes que dans le cas des formules closes. Par conséquence, la classe de formules que nous avons donnée ne concerne que la validité équationnelle. Cela constitue une restriction sur les classes de propriétés que nous pouvons prouver dans une spécification, car il peut exister des formules qui nécessitent la preuve pour toutes les valeurs possibles affectées à leurs variables. Nous ne pouvons donc pas passer à la validité des formules closes associées. Par contre il existe des méthodes de preuve par induction pour éviter de recenser toutes ces valeurs possibles des variables et qui ramène la preuve à la validité équationnelle. Le chapitre suivant sera consacré à ce sujet.

Chapitre 3

La preuve par induction

1 Introduction

Dans le chapitre 1 nous avons vu qu'il existe deux aspects de preuve de validité d'une propriété élémentaire. Le premier aspect qui est la validité équationnelle concerne une partie limitée de ces propriétés car la preuve se restreint aux formules closes. Le deuxième aspect qui est la validité inductive concerne la majeur partie de ces propriétés et nécessite la preuve de validité pour toutes les valeurs possibles affectées à leurs différentes variables. La preuve par induction consiste à exprimer d'une manière finie ce nombre infini de calculs.

Exemple 3.1
*On considère la spécification algébrique des entiers : ({N},{0, s, 1, +, *}, E). E est un enrichissement des axiomes de Peano :*

1. $x + 0 == x$
2. $x + s(y) == s(x+y)$
3. $1 == s(0)$
4. $x * 0 == 0$
5. $x * s(y) == (x * y) + x$

L'équation $x + s(0) == s(x)$ est un théorème équationnel que nous pouvons prouver à partir des équation 2 puis 1. Tandis que l'équation $x + y == y + x$ et l'équation $x + (y + z) == (x + y) + z$ nécessitent une preuve par induction.

Dans ce chapitre nous allons nous intéresser aux méthodes d'automatisation des preuves par induction. Deux grandes tendances se distinguent par leur système formel et par leurs stratégies de preuve. La première utilise la récurrence classique, est appelé l'*induction structurelle* [Aub79] [BM79]. La deuxième utilise la notion de consistance, est appelée l'*induction sans induction* ou *la preuve par consistance* [HH82].

La méthode que nous avons adopté utilise l'induction structurelle. Mais pour être complet, nous allons donner un aperçu sur la méthode de preuve par consistance, son domaine d'application ses avantages et ses inconvénients.

2 L'induction sans induction

L'induction sans induction est basée sur la notion de preuve par consistance utilisée dans la logique de premier ordre. Dans la logique équationnelle cette consistance est difficile à définir. Par conséquent, la définition de consistance de la logique de premier ordre devient donc inutilisable. Algébriquement un ensemble d'équations est consistant si est seulement si il admet un modèle non trivial; autrement dit son modèle initial a plus d'un élément. Cette notion de consistance est difficile à tester. C'est pour cela que plusieurs méthodes de preuve par induction utilisent en plus de cette notion de consistance d'autres contraintes sur les spécifications. Ce qui réduit le nombre de théorèmes inductifs qu'ils peuvent prouver.

L'origine de cette méthode de preuve par consistance est le travail de D. Musser [Mus80]. Son but était d'utiliser les techniques développées autour de la réécriture pour proposer une méthode totalement automatique. Cette méthode consiste à utiliser un raisonnement équationnel pour démontrer la validité d'une équation dans l'algèbre initiale. En d'autres termes, il faut ajouter l'équation à démontrer à l'ensemble d'équations initiales et regarder si le résultat reste consistant. Le sens de la consistance dépend des restrictions faites sur l'ensemble des équations initiales. La stratégie de preuve repose sur l'algorithme de complétion de Knuth et Bendix [KB70], adapté à la notion de consistance choisie.

Plusieurs améliorations de cette méthode ont été proposées depuis :

- G. Huet et J-M. Hullot [HH82], sous les hypothèses de complétude suffisante des opérateurs définis par rapport aux constructeurs, et de constructeurs libres (pas de relation entre les constructeurs), ont proposé un algorithme basé sur une extension de la procédure de complétion de Knuth et Bendix. Le système d'équations finales est inconsistant s'il contient une équation qui met en relation des constructeurs. Cet algorithme avait subi plusieurs améliorations notamment celle de L. Fribourg [Fri86], qui avait proposé une méthode pour ne calculer que les paires critiques utiles à la démonstration. (La notion de paire critique est la base de l'algorithme de complétion de Knuth et Bendix. Elle consiste à superposer deux équations)

- E. Paul [Pau84] avait proposé une autre approche dans le cas des spécifications contenant des relations entre les constructeurs, mais en plus de la complétude suffisante il faut que l'ensemble des équations exprimant ces relations soit inductivement complet. (Une spécification (S, F, E) est inductivement complète si et seulement si tout théorème inductif est un théorème équationnel). L'inconsistance se manifeste ici dans le cas de la génération d'une nouvelle relation entre les constructeurs qui n'était pas valide avant.

- Une troisième approche proposé par Jouannaud et Kounalis [JK86], présume ne pas supposer d'hypothèse sur l'ensemble d'équations initiales. Ils utilisent la propriété de quasi_réductibilité des membres gauches des équations. Un terme t est quasi_réductible (inductivement_R_ réductible) par rapport à un système de réécriture R si et seulement si toute instance close de t est réductible par rapport à R. En d'autres termes ces instances closes peuvent être réécrites au moins une fois. Le système est inconsistant s'il génère une équation qui a le membre gauche non quasi_réductible.

La stratégie de preuve de ces trois méthodes se compose en deux étapes. La première consiste à prouver que le système de réécriture de base possède la propriété de confluence close. Autrement dit, qu'il termine et que l'ordre de la réécriture des termes clos n'a pas d'influence sur le résultats.

Cette preuve est réalisable à l'aide de la procédure de complétion de Knuth et Bendix, Elle prend comme donnée l'ensemble des équations de base. Si cette procédure termine alors on peut dire que le système est convergent. La deuxième étape consiste à ajouter l'équation à démontrer au système résultat puis de rappeler la procédure de complétion. Si cette procédure échoue alors l'équation n'est pas un théorème inductif. Si elle réussit alors c'est un théorème inductif. Dans le cas ou elle ne termine pas, on ne peut pas décider si l'équation est un théorème inductif ou non.

2.1 Inconvénients de la méthode

La difficulté commune aux deux premières méthodes réside dans la preuve de la complétude suffisante qui est indécidable en général. T. Nipkow et G. Weikum [NW83] ont montré sa décidabilité dans le cas des systèmes de réécriture linéaires gauches. Plusieurs méthodes plus ou moins efficaces ont été proposées, chacune d'entre elle impose des restrictions sur la spécification initiale; citons celles de G. Huet et J. M. Hullot, Bidoit, et celle de J. J. Thiel améliorée et prouvée dans [LLT86].
Pour la deuxième méthode le test de la complétude inductive reste un problème ouvert. Aucune méthode automatique n'a été proposée. J. Heering [Hee86], E. Paul ont étudié cette propriété sur des exemples de spécifications. A. Lazrek [Laz88] a relevé les problèmes qu'il faut étudier pour aboutir à un système automatique qui peut tester cette propriété, appelé la ω_complétude. Le problème majeur de la méthode proposée par Jouannaud et Kounalis est de trouver un algorithme de décidabilité de la réductibilité inductive. Plusieurs algorithmes ont été proposés pour tester si un terme vérifie cette propriété dans le cas des systèmes de réécriture linéaires à gauche. Dans le cas non linéaire à gauche plusieurs algorithmes ont été proposés dans [KNZ87, Com88].

L'extension de ces méthodes aux spécifications conditionnelles n'a pas été suffisamment étudiée. Ainsi, il n'existe pas de méthode automatique pour prouver la complétude suffisante des spécifications conditionnelles. En ce qui concerne la confluence close, on ne peut la décider que dans un cas particulier des systèmes de réécriture conditionnelle [Rem82] [RZ84]. L'extension au cas général a été récemment abordée par [BR87a] et [Gan87].

2.2 Avantage de la méthode

La méthode de preuve par consistance est une méthode totalement automatique. Elle est applicable sur des équations simples chaque fois qu'on peut supposer la complétude suffisante du système de réécriture de base, ou bien supposer les autres hypothèses faites par les autres approches. Elle donne des résultats satisfaisants dans le cas des systèmes de réécriture simples qui ne contiennent ni règle conditionnelle ni équation non orientable [Laz88].

3 L'induction structurelle

Rappelons d'abord que nos objectifs n'appartiennent pas au domaine d'application de l'induction sans induction. En effet, nous voudrions utiliser des spécifications avec des équations conditionnelles, alors qu'il n'existe pas actuellement de système basé sur l'induction sans iduction qui puisse travailler dans un tel environnement sans faire de restrictions. Nous voudrions ne pas supposer de propriétés globales sur les spécifications, telle que la complétude suffisante ou la complétude close, alors que ces propriétés sont la base de cette méthode. Nous voudrions enfin

prouver des propriétés élémentaires non seulement sous forme d'équations simples, mais sous une forme plus générale représentée par la notion de formule équationnelle que nous avons définie dans le chapitre précédent. Ces exigences nous ont amené à choisir l'utilisation de la méthode classique de la preuve par induction. Il s'agit de faire une récurrence en utilisant les opérateurs avec lesquels tout objet du type abstrait peut être généré; ce sont les constructeurs. Avant de développer en détail le principe de l'induction structurelle, nous allons rappeler la sémantique algébrique de la validité inductive d'une formule équationnelle.

3.1 Validité Inductive

Définition 3.1 Validité inductive d'une formule équationnelle

Soit $SP = (S, C \cup D, E)$ une spécification. Une formule équationnelle f est inductivement valide dans SP en respectant les constructeurs si et seulement si, pour toute substitution close sur les constructeurs $\sigma : X \longrightarrow T(C)$, $\sigma(f)$ est équationnellement valide. Nous dirons aussi que f est un théorème inductif de SP ou de E, ou une conséquence inductive de E, et nous la notons $E \models\!\!\!= f$.

Nous rappellerons aussi deux propriétés intéressantes que nous allons utiliser dans la suite et qui mettent en évidence la relation entre la validité équationnelle et la validité inductive.

Proposition 3.1

Si une formule est équationnellement valide dans une spécification $(S, C \cup D, E)$ alors elle est inductivement valide en respectant les constructeurs C.

Preuve : trivial.

Proposition 3.2

Une formule équationnelle close est équationnellement valide dans une spécification $(S, C \cup D, E)$ si et seulement si elle est inductivement valide en respectant les constructeurs C.

Preuve : trivial.

3.2 Les principes de l'induction structurelle

L'induction structurelle a été initialement utilisée dans [Bur69] et dans [BM79] pour prouver les programmes écrits en Lisp. Nous la retrouvons aussi dans [ZKK88] et [GG88], qui l'utilisent pour prouver certaines propriétés élémentaires sous forme d'équations simples dans le cadre des spécifications équationnelles. Elle est basée sur la notion de récurrence classique utilisée souvent pour prouver des propriétés relatives aux entiers naturels, ou aux ensembles inductifs. En effet, pour réaliser la condition de la définition 3.1, il faut tester la validité de la propriété pour tous les termes de $T(C)$. Ce calcul infini peut être remplacé par une récurrence sur les termes de $T(C)$.

Exemple

Nous reprenons la spécification des entiers de l'exemple 3.1. Pour prouver la validité inductive d'une propriété comme l'associativité de $+$, il suffit de la prouver pour 0. Puis supposer qu'elle

valide pour un entier n et montrer qu'elle reste valide pour le successeur de n. Cela vient du fait que tout entier élément de $T(\{0, s\})$ est une formule bien formée à l'aide des opérateurs 0 et s.

Dans ce cas on peut exprimer la récurrence sur les entiers avec la formule de second ordre :

$$R = \forall P[(P(0) \text{ et } \forall x[P(x) \Rightarrow P(s(x))]) \Rightarrow \forall x P(x)]$$

Cependant nous voudrions écrire cette formule R sous une forme plus opérationnelle. Et puisqu'il s'agit d'une méta règle de déduction, la manière la plus adéquate serait de la représenter sous forme de règle d'inférence. La règle la plus évidente nous permettra de déduire $P(x)$ à partir de cette infinité de propositions : $P(0), P(s(0)), \ldots, \ldots$
Cependant en plus du fait que cette règle n'est pas efficace car elle a une infinité de prémisses, on ne peut pas l'exprimer d'une manière formelle. D'ou la règle suivante connue sous le nom de *schéma d'induction* [Bur69], qui nous permet de prouver une classe importante de propriétés inductives sur les entiers.

$$\frac{E \vdash_i P(0) \text{ et } E \cup \{P(c)\} \vdash_i P(s(c))}{E \vdash_i \forall x\ P(x)}$$

c est une variable qui n'apparaît ni dans E ni dans P.

Cas général

Le problème qui se pose dans le cas général est l'existence d'une méthode qui permettrait de calculer automatiquement le schéma d'induction à partir d'une spécification et de la formule équationnelle à prouver. Cependant il faut signaler qu'il n'existe aucune méthode applicable à tous les théorèmes inductifs, et qui permettrait ainsi de les prouver. Il faut donc se contenter de méthodes applicables à des classes de théorèmes plus ou moins grandes.

La méthode que nous allons exposer est basée sur les travaux de R. Aubin [Aub79]. Elle permet de générer automatiquement les schémas d'induction pour prouver une classe importante de théorèmes inductifs qui sont des propriétés relatives à des fonctions qui ont été définis à partir des constructeurs.

Avant d'exposer cette méthode, rappelons quelques définitions utiles.
On considère une spécification $(S, C \cup D, E)$ d'un type abstrait quelconque. A chaque fonction f de $F = C \cup D$ est associé un profil $u \rightarrow s$, où u est une suite de sortes et s une sorte quelconque. C est l'ensemble des *constructeurs*: ce sont les opérateurs à partir desquels tous les objets du types abstrait de données peuvent être exprimés ou engendrés. Parmi ces constructeurs se distinguent ceux qui n'ont aucune sorte dans leur domaine de départ (u est une liste vide). On les appelle *les constructeurs de base* ou les *constantes*. D est l'ensemble des *opérateurs définis*.

Exemple 3.2

Soit la spécification des entiers précédemment définie par :
($\{N\}$, $\{0, s, 1, +, *\}$, A)

1. *Le constructeur de base est* 0
2. *Les constructeurs sont* 0, s
3. *Les fonctions définies sont* 1, $+$, $*$

Pour généraliser cette notion de schéma d'induction, nous allons construire les termes sur lesquels sera basée la récurrence. Ces termes seront appelés des *termes générateurs*. Il s'agit d'un ensemble fini de termes avec lesquels nous pouvons exprimer tous les termes clos sur les constructeurs. Ces termes générateurs nous permettront de transformer la preuve par induction d'une preuve sur une infinité de terme en une preuve sur un ensemble fini de termes. Ces derniers dépendent de l'ordonnencement de la suite des variables sur lesquelles sera appliquée la récurrence. Nous allons ensuite construire pour chaque terme générateur la liste des termes qui le précédent (dans l'exemple le terme c précède le terme $s(c)$). Ces prédécesseurs seront utilisés pour définir les hypothèses de récurrence.

Définition 3.2 Variable d'induction

Soit P la propriété à prouver par induction. Nous appelons variables d'induction la suite des variables x^* de P sur lesquelles sera appliquée la récurrence. Dans la suite nous noterons $P(x^*)$ pour dire qu'il faut prouver la propriété P avec une récurrence sur les variables x^*.

Le choix de ces variables d'induction n'est généralement pas automatisable. Nous allons en discuter dans les paragraphes suivants.

Définition 3.3 Ensemble de termes générateurs

Soient $P(x^*)$ une propriété à prouver par induction dans une spécification (S, F, E), $s_1, \ldots, s_n$ la suite des sortes respectives des variables de la suite x^* et EG un ensemble de n_uplets de $T(C,X)_{s_1} \times \ldots \times T(C,X)_{s_n}$. La fonction Ψ de EG vers l'ensemble des parties de $T(C)_{s_1} \times \ldots \times T(C)_{s_n}$ est une fonction génératrice si et seulement si :

1. *Pour tout n_uplet $(t_1, \ldots, t_n)$ de $T(C)_{s_1} \times \ldots \times T(C)_{s_n}$ il existe un n_uplet $(t'_1, \ldots, t'_n)$ de EG et une substitution σ tel que :*
   - $(t_1, \ldots, t_n) \in \Psi(t'_1, \ldots, t'_n)$ *et*
   - *pour tout $i \in [1..n]$, $\sigma(t'_i) = t_i$,*
2. $\Psi(t'_1, \ldots, t'_n) \neq \emptyset, \forall (t'_1, \ldots, t'_n) \in EG$,
3. $(t'_1, \ldots, t'_n)$ *dans (1) est unique.*

Si l'ensemble EG possède une fonction génératrice alors il est appelé ensemble de termes générateurs.

Il existe plusieurs ensembles de termes générateurs possibles pour une suite de sortes donnée. Nous en avons choisi un qui est le plus général et qui est facilement calculable. Il s'agit de l'ensemble des n_uplets de termes obtenus en combinant les constructeurs des sortes $s_1, \ldots, s_n$ dans l'ordre dans lequel ils occurrent. Plus formellement :

Proposition 3.3

Soit $s_1, \ldots, s_n$ une suite de sorte. L'ensemble EG défini par :
$EG = \{(f_1(x_{1_1}, \ldots, x_{m_1}), \ldots, f_n(x_{1_n}, \ldots, x_{m_n}))$ *tel que* $f_i \in C$ *et* $f_i : s_{1_i} \ldots s_{m_i} \longrightarrow s_i$ *et* x_{i_j} *une variable de sorte* $s_{i_j}\}$
est un ensemble de termes générateurs pour la suite de sortes $s_1, \ldots, s_n$.

Preuve :
La fonction Ψ entre EG et $T(C)_{s_1} \times \ldots \times T(C)_{s_n}$ est définie pour tout n-uplet $(f_1(x_{1_1}, \ldots, x_{m_1}), \ldots, f_n(x_{1_n}, \ldots, x_{m_n}))$ par :
$\Psi(f_1(x_{1_1}, \ldots, x_{m_1}), \ldots, f_n(x_{1_n}, \ldots, x_{m_n})) = (f_1(t_{1_1}, \ldots, t_{m_1}), \ldots, f_n(t_{1_n}, \ldots, t_{m_n}))$
avec $t_{i_j} \in T(C)_{s_{i_j}}$.
Les autres conditions sur Ψ sont faciles à vérifier.

Exemple 3.3

Les termes générateurs pour une propriété $P(x)$ sur les entiers, avec une seule variable d'induction x, sont les termes 0 et $s(c)$ avec c une variable quelconque.
Pour une propriété $P(x,y)$ qui utilise deux variables d'induction, les termes générateurs sont les suites de termes $0, 0$; $0, s(n)$; $s(m), 0$; $s(m), s(n)$. Avec m et n des variables de sorte entier quelconques.

Nous allons maintenant calculer les prédécesseurs de ces termes générateurs pour les utiliser dans la définition des hypothèses de récurrence. Pour cela nous aurons besoin de définir un ordre sur les termes.

Définition 3.4 Ordre de réduction

Un ordre $<$ sur $T(F,X)$ est un ordre de réduction, si et seulement si il possède les propriétés suivantes :

- *$<$ est bien fondé (il n'existe pas de chaîne infinie de termes $\ldots, t_2 < t_1$),*
- *(propriété de compatibilité): $t_1 < t_2 \Rightarrow f(\ldots, t_1, \ldots) < f(\ldots, t_2, \ldots)$*
- *(propriété de stabilité par instanciation): Pour toute substitution σ, $t_1 < t_2 \Rightarrow \sigma(t_1) < \sigma(t_2)$*

Définition 3.5 Terme à argument récursif

Un terme $f(t_1, \ldots, t_n)$ de $T(F,X)_s$ est dit à argument récursif si il existe un $i \in [1..n]$ tel que t_i est de sorte s. Le terme t_i est alors appelé un argument de récursivité.

Exemple 3.4

Le terme $s(m)$ est un terme à argument récursif. Le terme m est l'argument de récursivité.

Convention 6

Tout les termes de $T(C)$ qui ne sont pas des constantes sont des termes à argument récursif.

Définition 3.6 Prédécesseur

Un prédécesseur d'un terme générateur $(t_1, \ldots, t_n)$ est un n_uplet de termes $(t'_1, \ldots, t'_n)$ tel qu'il existe un $i \in [1..n]$ tel que :

- *$t_j = t'_j$ pour tout $j \in [1..i-1]$ et*
- *t'_i est un sous terme de t_i avec $t'_i < t_i$ et t'_i est un argument de récursivité dans t_i. ($<$ est l'ordre de réduction sur les termes).*

Un prédécesseur de la liste des termes $c_1(x_1^*), \ldots, c_n(x_n^*)$ est une liste de termes:
$c_1(x_1^*), \ldots, c_{i-1}(x_{i-1}^*), x_{ij}, t_{i+1}, \ldots, t_n$.
avec x_{ij} un argument de récursivité de $c_i(x_i^*)$ et t_k $(i+1 \leq k \leq n)$ des termes quelconques.

Exemple 3.5

La liste des termes 0, 0 n'a pas de prédécesseur.
La liste 0, s(n) a comme prédécesseur la liste 0, n.
La liste s(n), 0 a comme prédécesseur la liste n, t. t est un terme quelconque.
La liste s(m), s(n) a deux prédécesseurs: la liste m, t et la liste s(m), n.

Définition 3.7 Théorème inductif

Soit $P(x^*)$ la propriété à prouver par induction à partir d'un ensemble d'axiomes E. Cette propriété est exprimée sous forme d'une formule équationnelle.
Soit $c_{i1}(x_{i1}^*), \ldots, c_{in}(x_{in}^*)$ avec $i \in [1..r]$, les termes générateurs associés à cette propriété. Soient p_i $i \in [1..r]$ des implications de la forme

$$P(t_{i1}^*) \wedge \ldots \wedge P(t_{im_i}^*) \supset P(c_{i1}(x_{i1}^*), \ldots, c_{in}(x_{in}^*))$$

avec t_{ij}^* $j \in [1..m_i]$ sont les prédécesseurs de $c_{i1}(x_{i1}^*), \ldots, c_{in}(x_{in}^*)$.
Alors $P(x^*)$ est un théorème inductif si on peut le déduire à partir de la règle d'inférence suivante:

$$\frac{E \vdash_i p_1 \wedge \ldots \wedge p_r}{E \vdash_i P(x^*)}$$

Exemple 3.6

Pour exprimer une double induction sur les entiers on utilise la règle d'inférence suivante:

$$\frac{\begin{array}{l} E \vdash_i P(0,0) \\ \quad (P(0,n) \supset P(0,s(n))) \\ \quad (P(m,t) \supset P(s(m),0)) \\ \quad ((P(m,t) \wedge P(s(m),n)) \supset P(s(m),s(n))) \end{array}}{E \vdash_i P(x,y)}$$

En effet, les constructeurs sont 0 et s. Ils donnent quatre combinaisons de deux (deux variables d'induction): $(0,0),(0,s),(s,0),(s,s)$. Les prédécesseurs de ces couples sont donnés dans l'exemple précédent. D'où la règle d'inférence ci-dessus. Les prémisses de cette règle peuvent être calculée à la main en faisant une induction sur x puis sur y.

Proposition 3.4

Soient $(S, C \cup D, E)$ une spécification équationnelle et f une formule équationnelle.
Si f est un théorème inductif alors f est inductivement valide.

Preuve :

Il s'agit de prouver que si la formule f a été prouvé en utilisant un schéma d'induction de la définition 3.7 alors f vérifie les conditions de la définition 3.1. Autrement dit toute instance close sur les constructeurs de f est équationnellement valide.
Nous allons faire la preuve pour le cas d'une seule variable d'induction x. Le cas de

plusieurs variables d'induction sera une généralisation de celle ci.
Nous allons faire une preuve par l'absurde, pour cela supposons qu'il existe une substitution close sur les constructeurs $\sigma : X \longrightarrow T(C)$ telle que $E \not\models_\Sigma \sigma(f)$, avec $\Sigma = (S, C \cup D)$.
Soit σ telle que $t = \sigma(x)$ soit minimal par l'ordre de réduction $<$ sur les termes.
t est un élément de $T(C)$ donc il existe un terme générateur t_1 et une substitution θ tel que $t = \theta(t_1)$.
Soit t_2 un prédécesseur de t_1. Puisque f est un théorème inductif alors d'après la définition 3.7 nous pouvons dire que:
$E \vdash_i f[x \leftarrow t_2] \supset f[x \leftarrow t_1]$
$\Longrightarrow$
$E \vdash_i f[x \leftarrow \theta(t_2)] \supset f[x \leftarrow \theta(t_1)]$
et puisque $f[x \leftarrow \theta(t_2)] \supset f[x \leftarrow \theta(t_1)]$ est une formule close, nous pouvons dire que (proposition 3.2)
$E \models_\Sigma f[x \leftarrow \theta(t_2)] \supset f[x \leftarrow \theta(t_1)]$
$\Longrightarrow$
$E \models_\Sigma f[x \leftarrow \theta(t_2)] \Longrightarrow E \models_\Sigma f[x \leftarrow \theta(t_1)]$
Or $E \not\models_\Sigma f[x \leftarrow \theta(t_1)]$ car $t = \theta(t_1)$ donc
$E \not\models_\Sigma f[x \leftarrow \theta(t_2)]$
et puisque $\theta(t_2) < \theta(t_1) = t$ cela contredit le fait que t est minimal.

Exemple 3.7

Soit la propriété $P(y,x) = x + y == y + x$. L'ordre des variables x et y dans la propriété P à une influence sur la preuve, comme nous allons le voir dans le paragraphe suivant.
Le schéma d'induction suivant permet de prouver la commutativité de l'addition

$$\frac{\begin{array}{l} E \vdash_i 0+0 == 0+0 \\ \quad n+0 == 0+n \supset s(n)+0 == 0+s(n) \\ \quad t+m == m+t \supset 0+s(m) == s(m)+0 \\ \quad (t+m == m+t \wedge n+s(m) == s(m)+n) \supset s(n)+s(m) == s(m)+s(n) \end{array}}{E \vdash_i x+y == y+x}$$

Les quatres prémisses peuvent être prouvées seulement par un raisonnement équationnel.

3.3 Application aux formules équationnelles

Décomposition et validité inductive

La question qui se pose dans ce cadre consiste à se demander pourquoi ne pas appliquer des règles de décomposition, comme dans le cas de la validité équationnelle, pour faciliter la preuve par induction. Malheureusement la plupart des règels de décomposition du chapitre précédent ne sont correctes que parce que nous avons considéré les variables des formules commes des constantes. Ce n'est plus le cas pour les formules à prouver par induction, car nous avons besoin de tester leurs validité pour chaque instance de leurs variables. La seule règle de décomposition correcte et qui a un intéret pratique est celle qui transforme la validité inductive d'une conjonction de formules en la conjonction des validités de chaque formule: $E \models\!\models_\Sigma f_1 \wedge f_2 \Longleftrightarrow E \models\!\models_\Sigma f_1 \wedge E \models\!\models_\Sigma f_2$. La correction de cette règle vient du fait que pour toute Σ-algèbre A, $A \models_\Sigma f_1 \wedge f_2 \Longleftrightarrow A \models_\Sigma f_1 \wedge A \models_\Sigma f_2$.

Déroulement de la preuve par induction

Soient f une formule équationnelle et x^* la liste des variables sur lesquelles va être appliquée l'induction. La définition 3.7 nous permet de générer un schéma d'induction adapté à notre formule et à nos variables d'induction. En appliquant ce schéma sur f nous obtenons une conjonction de formules $f_1 \wedge \ldots \wedge f_r$ avec chaque f_i de la forme $f(t^*_{i1}) \wedge \ldots \wedge f(t^*_{im_i}) \supset f(c_{i1}(x^*_{i1}), \ldots, c_{in}(x^*_{in}))$. Or d'après la propriété 3.1, si nous arrivons à prouver la validité équationnelle d'une formule alors nous pouvons déduire qu'elle est inductivement valide. Et d'après le théorème 2.2 pour prouver la validité équationnelle de cette formule il suffit de prouver la validité de la formule close associée. Or, pour prouver une formule d'une telle complexité nous aurons besoin de faire appel à la décomposition; ce qui nous permettra de la transformer en un ensemble de a_équations et de réduire ainsi sa preuve à la preuve de ces a_équations. Si cette preuve réussit nous pouvons conclure que la formule initiale est inductivement valide, sinon si cela est nécessaire, une autre induction sur d'autres variables pourra être proposée.

Exemple 3.8

On considère la formule équationnelle $estvide(e) == faux \supset min(e) \leq max(e) == vrai$. *Pour la prouver nous avons besoin de faire une induction. Rappelons que la spécification dans laquelle nous voulons faire cette preuve est celle des ensembles ordonnés de l'exemple 1.13. Le constructeur de base est "ensvide". L'autre constructeur est l'opérateur "union".*

La variable d'induction est e. *Le schéma d'induction associé est :*

$$\frac{E \vdash_i P(ensvide) \wedge (P(X) \supset P(union(X,a)))}{E \vdash_i P(X)}$$

En appliquant ce schéma sur notre formule nous obtenons la formule suivante :

$$\begin{array}{l}(estvide\ (ensvide) == faux \supset min(ensvide) \leq max(ensvide) == vrai) \wedge \\ \quad (\ (estvide(X) == faux \supset min(X) \leq max(X) == vrai) \\ \qquad\qquad \supset (estvide(union(X,a)) == faux \\ \qquad\qquad\quad \supset min(union(X,a)) \leq max(union(X,a)) == vrai)\ ).\end{array}$$

La décomposition de cette formule nous donne l'ensemble de a_équations suivant :

$$\begin{array}{l}\{\{estvide(ensvide) == faux\},\ min(ensvide) \leq max(ensvide) == vrai\}, \\ \{\{estvide(X) == vrai,\ estvide(union(X,a)) == faux\}, \\ \qquad min(union(X,a)) \leq max(union(X,a)) == vrai\}, \\ \{\{estvide(X) == faux,\ min(X) \leq max(X) == vrai,\ estvide(union(X,a)) == faux\}, \\ \qquad min(union(X,a)) \leq max(union(X,a)) == vrai\}\}.\end{array}$$

Il reste à prouver séparément la validité équationnelle de chaque a_équation

Classe de formules prouvable par induction

Lorsque nous avons déterminé la classe de formules sur lesquelles nous pouvons appliquer notre méthode de preuve équationnelle, le critère de choix était de savoir si ces formules sont décomposables ou non. Dans le cas de la validité inductive, la décomposition n'intervient qu'après avoir appliqué le schéma d'induction. Or, nous ne pouvons savoir à priori ni la forme du schéma d'induction ni la forme de la formule résultat. Mais si nous supposons que la partie premisse d'un schéma est généralement de la forme $E \vdash_i p_1 \wedge \ldots \wedge p_r$ décrite dans la définition 3.7

avec $p_i = P(t^*_{i1}) \wedge \ldots \wedge P(t^*_{im_i}) \supset P(c_{i1}(x^*_{i1}), \ldots, c_{in}(x^*_{in}))$, nous pouvons déterminer une classe de formules sur lesquelles l'application de tel schéma d'induction donnerait des formules décomposables. Cette classe sera celle des formules susceptibles d'être prouvées par induction.

Proposition 3.5

Sous l'hypothèse que la forme générale des schémas d'induction est celle de la définition 3.7. La classe de formules susceptibles d'être prouvées par induction C_INT est définie par :

1. *Soit f une formule conjonctive de $\bar{C}$ alors f appartient à C_INT,*
2. *soit f une formule disjonctive de $\bar{D}$ alors f appartient à C_INT,*
3. *soit f une formule booléenne de $\bar{B}$ alors f appartient à C_INT,*
4. *soient f_1 une formule booléenne de $\bar{B}$ et f_2 une formule de $\bar{C} \cup \bar{D}$ alors $f_1 \supset f_2$ appartient à C_INT.*

Preuve :

Il faut prouver que la formule obtenue par l'application d'un schéma d'induction sur une formule de C_INT, est une formule décomposable appartenant à $DECOMP$ (cf définition 2.9).

- Soit f une formule conjonctive. La formule générée par le schéma d'induction est de la forme $p_1 \wedge \ldots \wedge p_r$ avec $p_i = f_{1_i} \wedge \ldots \wedge f_{n_i}$. Cette forme est une forme décomposable d'après la clause 3 de la définition 2.9.
- Même démarche pour les formules disjonctives et booléennes.
- Soit f une formule de la forme $f \supset g$ avec $f \in \bar{B}$ et $g\bar{C} \cup \bar{D}$. La formule générée par le schéma d'induction est de la forme $p_1 \wedge \ldots \wedge p_r$ avec $p_i = ((f_1 \supset g_1) \wedge \ldots \wedge (f_m \supset g_m)) \supset (f_{m+1} \supset g_{m+1})$, et $f_i \in \bar{B}$ et $g_i \in \bar{C} \cup \bar{D}$. Cette forme est décomposable d'après les clauses 8 puis 3 de la définition 2.9.

3.4 Comparaison avec les autres approches

Il existe une autre méthode de génération automatique de schéma d'induction. Elle a été proposée dans [ZKK88] et [Zha88], elle est fondée sur l'idée de chercher un ensemble de termes qui représentent l'ensemble des termes clos sur les constructeurs. Cet ensemble est appelé *ensemble de recouvrement*. La différence entre l'ensemble des termes générateurs est que ce dernier ne peut contenir que des éléments de $T(C,X)$ alors que l'ensemble de recouvrement peut contenir des termes de $T(C \cup D, X)$ avec des opérateurs définis. C'est une méthode plus générale que la nôtre dans le sens où elle permet de prouver des théorèmes inductifs avec des opérateurs qui n'ont pas été définis à partir des constructeurs. Par exemple l'opération *plus grand commun diviseur gcd* est définie par les équations :

$gcd(0,x) == x \quad gcd(x,0) == x$

$gcd(x+y,y) == gcd(x,y) \quad gcd(x,x+y) == gcd(x,y)$

Si nous voulons prouver la commutativité de gcd, l'ensemble de termes générateurs que nous pourrons proposer est $\{(0,0),(0,suc(x)),(suc(x),0),(suc(x),suc(y))\}$. Le schéma d'induction qui en résultera ne permettra pas de prouver cette commutativité. Tandis que les auteurs dans [ZKK88] proposent l'ensemble de recouvrement suivant : $\{(0,y),(x,0),(x+y,y),(x,x+y)\}$. Le schéma d'induction :

$$\frac{E \vdash_i P(0,y) \wedge P(x,0) \wedge (P(x,y) \supset P(x+y,y)) \wedge (P(x,y) \supset P(x,x+y))}{E \vdash_i P(a,b)}$$

permet de montrer que le *gcd* est commutatif.

Cette méthode nécessite des techniques plus compliquées pour la mise en œuvre. Ainsi, pour générer cet ensemble de recouvrement, des tests doivent être effectués sur la complétude suffisante des opérateurs définis. En plus, pour déterminer l'équivalent de nos termes prédécesseurs les auteurs utilisent un ordre plus compliqué appelé *l'inéquivalence inductive.* Ces difficultés compliquent l'implémentation de cette méthode.

Mais il y a des cas où cette méthode ne peut pas générer le bon ensemble de recouvrement, comme le montre l'exemple suivant:

Exemple 3.9

$S = \{entier\}$, $C = \{0, suc\}$, $D = \{+\}$
$E = \{suc(suc(x)) == x$, *Les autres équations pour définir* $+\}$.
Notre objectif est de prouver que l'équation $suc(x) + suc(y) == x + y$ *est un théorème inductif. L'ensemble de recouvrement proposé à l'aide de la méthode de Zhang est* $(suc(x), suc(y))$. *Les couples de termes susceptibles d'être inférieurs à* $(suc(x), suc(y))$ *sont* (x, t) *et* $(suc(x), y)$. *Or ces couples ne sont pas inductivement inéquivalent à* $(suc(x), suc(y))$; *ce qui implique qu'avec la méthode d'ensemble de recouvrement l'auteur ne peut pas générer automatiquement de schéma d'induction.*
Cependant, avec notre méthode, l'ensemble des termes générateurs est celui proposé pour une induction sur deux variables : $\{(0,0), (0, suc(x)), (suc(x), 0), (suc(x), suc(y))\}$. *Ce qui permet de générer le schéma d'induction habituel et de prouver que notre équation est un théorème inductif.*

3.5 Sélection des variables d'induction

L'inconvénient de la méthode de preuve par induction structurelle, est qu'elle n'est totalement automatisable. En effet, il n'existe pas de méthode générale pour désigner les variables d'inductions. R. Aubin et R.S. Boyer ont proposé des heuristiques pour résoudre ce problème dans des cas particuliers, comme le traitement des listes ou bien la preuve de propriétés sur les programmes Lisp. Mais plusieurs questions se posent dans le cas général: sur quels critères doit on faire le choix des variables d'induction? Et dans le cas où nous devons faire une induction sur plusieurs variables, selon quel ordre devons nous l'appliquer?

La réponse à ces questions varie d'un domaine à un autre. Ainsi pour répondre à la première question nous pouvons dire qu'il faut faire une induction sur chaque variable de récursivité. Or cela est parfois inutile. Par exemple pour prouver $(x + y) + z == x + (y + z)$, une récurrence sur une seule variable x ou z suffit, alors que pour prouver $x + y == y + x$ il faut une double récurrence sur x et y.
La réponse à la deuxième question est plus délicate. Sur l'exemple précédent et en utilisant les équations $x + s(y) == s(x + y)$ et $x + 0 == x$, si on commence par une induction sur x puis sur y, la réduction s'arrêtera avec une forme irréductible qui n'est ni vraie ni fausse. Alors que si on commence par une induction sur y puis sur x, le résultat est immédiat. Si on prend les équations $s(x) + y == s(x + y)$ et $0 + x == x$ alors il faut commencer par une induction sur x puis sur y. A partir de cet exemple on peut dire qu'il faut commencer par la variable qui correspond à l'argument avec lequel on définit l'opération $+$.
Ainsi la seule tactique applicable d'une manière générale, est d'appliquer l'induction sur une seule variable, puis sur une autre, dans le cas où la première n'a pas permis de faire aboutir la preuve.

La méthode proposé par [ZKK88] ne permet pas non plus de contourner ce problème de choix de variables d'induction.

3.6 Comparaison entre la preuve par consistance et l'induction structurelle

La méthode de preuve par induction structurelle est une méthode généralement applicable à toutes les spécifications algébriques, dont les sortes sont générées par une famille de constructeurs, et dont les opérateurs sont complètement définis par un ensemble d'axiomes. Elle est applicable à condition de trouver le bon schéma d'induction, soit automatiquement soit par l'intermédiaire de l'utilisateur. Son domaine d'application recouvre celui de la méthode d'induction sans induction. En effet, aucune propriété globale n'est supposée, et elle est utilisable indifféremment dans les spécifications conditionnelles et inconditionnelles.

Ce problème de choix des variables d'induction et de schéma d'induction ne se pose pas dans la méthode d'induction sans induction. C'est le mécanisme de paires critiques qui est la base de l'algorithme de complétion de Knuth et Bendix qui génère les instances nécessaires à l'induction. Cette génération se fait en orientant l'équation à démontrer en une règle de réécriture, et en superposant ces règles entre elles. Cet algorithme génère des paires critiques autres que celles nécessaires à l'induction, car il fait toutes les superpositions possibles. Les risques de cet algorithme c'est qu'il peut générer indéfiniment des paires critiques, ce qui entrainerait la non terminaison de la preuve. Il risque aussi de générer des paires qui sont non orientables, ce qui entrainerait l'échec de la preuve.

Un autre inconvénient de la méthode basée sur l'induction sans induction, est que l'algorithme de Knuth et Bendix n'est pas déductif, ce qui implique une difficulté de suivre et de comprendre la preuve; et par conséquent de voir pourquoi la preuve a échoué, ou quel lemme ajouter pour relancer la preuve. Par contre avec l'induction structurelle chaque étape est déduite de l'étape précédente par un raisonnement simple, tel que calculer la forme normale d'un terme, ou bien faire une décomposition par cas. Ce qui rend la preuve plus lisible et plus compréhensible. Et dans le cas d'un échec, la preuve s'arrête sur une forme irréductible, qui donnerait une idée sur le lemme à ajouter pour faire converger la preuve. Ce lemme est assez souvent une généralisation de la forme irréductible.

Exemple 3.10 *[GG88]*

Soit l'ensemble d'équations suivant:

estvide(ensvide) = vrai
estvide(singleton(e)) = faux
estvide(union(s1, s2)) = estvide(s1) et estvide(s2)
sousens(ensvide ,s) = vrai
sousens(s, ensvide) = estvide(s)
sousens(singleton(e1), singleton(e2)) = (e1 = e2)
sousens(singleton(e), union(s1, s2)) = sousens(singleton(e), s1) ou
sousens(singleton(e), s2)
sousens(union(s1, s2), s3) = sousens(s1, s3) et sousens(s2, s3)

Pour prouver la reflexivité de l'inclusion $sousens(x, x)$ *avec ce système d'équations, on obtient la forme irréductible*
$sousens(x, union(x, y))$ *et* $sousens(y, union(x, y))$

Cette équation suggère le lemme sousens(s, union(s , s1)) = vrai, *qui est une généralisation de la forme irréductible. Ce lemme peut être prouvé par induction et permettra de compléter la preuve.*

4 Conclusion

Dans ce chapitre nous avons présenté les deux tendances actuelles de la preuve par induction qui sont la preuve par consistance ou l'induction sans induction et l'induction structurelle. Une comparaison entre ces deux méthodes puis une confrontation avec nos objectifs nous a permis de choisir la méthode basée sur l'induction structurelle. Ensuite nous avons essayé de déterminer la classe de formules susceptibles d'être prouvées par cette méthode d'induction.
Nous avons vu que l'inconvénient de l'induction structurelle est qu'elle n'est pas totalement automatique. En effet, elle exige la donnée des variables sur lesquelles sera appliquées l'induction pour pouvoir générer le schéma d'induction nécessaire. Nous avons donné une méthode pour générer automatiquement ce schéma d'induction. Mais en général, il existe des schémas qui ne peuvent pas être générés automatiquement. En plus, l'expérience a montré qu'il existe aussi beaucoup de théorèmes inductifs qui nécessitent des schémas d'induction particuliers.

Partie II

Mise en œuvre

Chapitre 4

Réécriture et raisonnement par cas

1 Introduction

Dans le chapitre 1 nous avons vu comment transformer le problème sémantique de validité d'une équation dans la variété d'algèbres engendrées par un ensemble d'équations E, en un problème syntaxique de E_égalité $=_E$. L'idée de base de cette méthode syntaxique, consiste à déterminer pour chaque classe d'équivalence, de la congruence $=_E$, un représentant canonique. Ce qui ramène le problème de E_égalité de deux termes t_1 et t_2 à celui de l'identité de leur représentant canonique. Ces représentants canoniques peuvent être obtenus en utilisant la relation de E_égalité en une étape $\vdash\dashv_E$ autant de fois qu'il est nécessaire.

Ce résultat mathématique du problème de validité n'est pas suffisant pour donner une bonne solution informatique. En effet la E_égalité en une étape ou le remplacement d'égaux par égaux nécessite de nombreux retours en arrière. Ce qui risque d'engendrer des boucles incontrôlables et de ce fait rendre la méthode inefficace. D'où l'idée d'une méthode visant non pas à supprimer les choix, mais à faire en sorte qu'ils fournissent tous le même résultat. C'est la base de la méthode de *réécriture*, qui consiste tout d'abord à choisir un sens d'utilisation des équations pour donner *des règles de réécriture*. L'ensemble de ces équations ainsi orientées forme un *système de réécriture* R. La E_égalité $\vdash\dashv_E$ devient *la relation de réécriture* $\longrightarrow_R$ associée à R. Cette méthode impose l'existence d'une forme normale pour chaque terme, ce qui est exprimé par la propriété de *terminaison*. D'autre propriétés ont été étudiées par de nombreux auteurs et particulièrement [Hue80] pour la réécriture simple et [Rem82] pour la réécriture conditionnelle.

Dans le chapitre 2 nous avons vu que pour pouvoir appliquer cette méthode syntaxique de E_égalité sur des formules équationnelles, il fallait les décomposer sous forme de a_équations. Nous avons vu aussi que la preuve de validité de ces a_équations revient à trouver une inconsistance entre leurs assertions ou bien à prouver la validité de leur partie conclusion dans l'ensemble d'équations initial augmenté avec les assertions. Ce chapitre a pour objectif de mettre en œuvre ces idées théoriques à l'aide des techniques de réécritures afin d'obtenir une solution informatique, et de décider ainsi la validité des a_équations. Nous allons suivre la même démarche que dans le cas des équations simples dans le sens où la preuve de validité d'une a_équation se ramènera à tester l'équivalence des formes normales de chacun de ses a_termes (un a_terme est constitué d'un ensemble d'assertions plus un terme). Nous commen-

cerons par définir la relation de réécriture entre les termes puis entre les a_termes. Ces définitions seront ensuite utilisées pour tester l'inconsistence des assertions puis de définir la validité d'une a_équation. Nous terminerons en donnant les différents algorithmes de calcul de formes normales ainsi que leurs conditions de terminaison.

2 Réécriture conditionnelle

Définition 4.1 Système de réécriture conditionnelle

Soit (S, F, E) une spécification équationnelle. Le système de réécriture conditionnelle associé à cette spécification est un triplet (S, F, R) où R est un ensemble de règles obtenues par orientation des équations de E tel que :

- *à une équation du type $g == d$, nous associons la règle $g \rightarrow d$,*
- *à une équation du type $\wedge_{i\in[1..n]} p_i == p'_i \Rightarrow g == d$, nous associons la règle $\wedge_{i\in[1..n]} p_i == p'_i \Rightarrow g \rightarrow d$,*
- *et à une équation du type $\wedge_{i\in[1..n]} p_i == p'_i \Rightarrow g == (d_1, d_2)$, nous associons la règle $\wedge_{i\in[1..n]} p_i == p'_i \Rightarrow g \rightarrow (d_1, d_2)$*

On suppose que $var(p_i), var(p'_i), var(d), var(d_1)$ et $var(d_2)$ sont inclus dans $var(g)$. Dans la suite un système de réécriture sera représenté seulement par son ensemble de règles R.

L'orientation des équations se fait de telle sorte qu'elle doit assurer la terminaison de la réécriture. Nous allons reprendre ce sujet plus en détail dans le paragraphe 5.

Hypothèse 2

Nous supposons que la spécification (S, F, E) est consistante et complète par rapport aux booléens et que les constantes vrai et faux sont irréductibles dans le système de réécriture R associé à cette spécification.

2.1 Réécriture conditionnelle récursive sur les termes

Définition 4.2 Relation de réécriture conditionnelle récursive entre les termes

Soient R un système de réécriture, t et t' deux termes. Un terme t se réécrit en t' et noté $t \longrightarrow_R t'$, dans l'un des trois cas :

- *il existe une règle $g \rightarrow d$, une occurrence u de t et une substitution σ telle que $t_{/u} = \sigma(g)$ et $t' = t[u \leftarrow \sigma(d)]$*
  ou
- *une règle $\wedge_{i\in[1..n]} p_i == p'_i \Rightarrow g \rightarrow d$, une occurrence u de t et une substitution σ telle que $t_{/u} = \sigma(g)$, pour tout i de $[1..n]$ $\sigma(p_i) \stackrel{*}{\longrightarrow}_R p'_i$, et $t' = t[u \leftarrow \sigma(d)]$*
  ou
- *une règle $\wedge_{i\in[1..n]} p_i == p'_i \Rightarrow g \rightarrow (d_1, d_2)$, une occurrence u de t, une substitution σ telle que*
  $t_{/u} = \sigma(g)$ et

  - *pour tout i de $[1..n]$ $\sigma(p_i) \stackrel{*}{\longrightarrow}_R p'_i$, et $t' = t[u \leftarrow \sigma(d_1)]$*
    ou
  - *il existe un i de $[1..n]$ tel que $\sigma(p_i) \stackrel{*}{\longrightarrow}_R \neg p'_i$ et $t' = t[u \leftarrow \sigma(d_2)]$.*

Les notations $\stackrel{+}{\longrightarrow}_R$ et $\stackrel{*}{\longrightarrow}_R$ représentent respectivement les fermetures transitive et reflexive-transitive de la relation $\longrightarrow_R$.
Il s'agit d'une relation de réécriture conditionnelle recursive dans le sens où les conditions des règles sont évaluées par réécriture avant de réécrire les termes.

Définition 4.3 Forme normale d'un terme

Un terme t est réductible par rapport à un système de réécriture R si et seulement si il existe un terme t' tel que $t \longrightarrow_R t'$. Une forme normale de t par rapport à R, noté $t\downarrow_R$, est un terme irréductible t' tel que $t \stackrel{*}{\longrightarrow}_R t'$.

2.2 Réécriture conditionnelle sur les a_termes

Par analogie à la preuve de validité des équations simples qui consiste à calculer les formes normales de chacune de ses composantes, nous allons suivre la même démarche. Tout d'abord nous allons définir les composantes d'une a_équation. Il s'agit des deux termes de la partie conclusion accompagnés chacun par la partie assertion.

Définition 4.4 Terme avec assertion

Un terme avec assertion ou un a_terme noté $\langle\{a_1, \ldots, a_n\}, t\rangle$ est constitué d'un ensemble d'équations closes sur $T(\Sigma \cup X)$: $\{a_1, \ldots, a_n\}$, appelé partie assertion, et d'un terme t clos de $T(\Sigma \cup X)$.

Rappelons que dans la convention 5, les composantes d'une a_équation sont considérées comme des formules close. Il s'agira de même pour les a_termes.

Cette notion de a_terme est différente de la notion de terme contextuel noté $c :: t$ et qui à été étudié dans [Zha84] et recement dans [Bou88b]. En effet un contexte c est une conjonction de termes booléens, alors que la partie assertion d'un a_terme peut contenir indifféremment des équations booléennes ou des équations quelconques.
Maintenant pour pouvoir calculer les formes normales d'un a_terme nous allons définir la relation de réécriture utilisée.
Cette relation de réécriture entre les a_termes permet, en plus de la relation de réécriture entre les termes, de faire du raisonnement par cas et de réécrire avec une assertion. Le raisonnement par cas est possible quand la condition d'une règle *si-alors-sinon* ne peut être évaluée ni à *vrai* ni à *faux*. Dans ce cas, nous générons deux nouveaux a_termes qui ont en plus des assertions initiales , pour le premier la condition supposée vrai et pour le deuxième la condition supposée fausse. Cela est justifié par la propriété de complétude par rapport aux booléens qui assure que si ces conditions étaient des termes clos de $T(\Sigma)_{bool}$, elle se seraient réduites à *vrai* ou à *faux*.

Définition 4.5 Réécriture de la partie terme dans un a_terme

Soit R un système de réécriture, et soient $\langle\{a_1, \ldots, a_n\}, t\rangle$ et $\langle\{b_1, \ldots, b_m\}, s\rangle$ deux a_termes. $\langle\{a_1, \ldots, a_n\}, t\rangle$ se réécrit en $\langle\{b_1, \ldots, b_m\}, s\rangle$, et nous le notons $\langle\{a_1, \ldots, a_n\}, t\rangle -a\longrightarrow_R \langle\{b_1, \ldots, b_m\}, s\rangle$, si et seulement si

- **réécriture conditionnelle simple**
  $t \longrightarrow_R s$, *et* $\langle\{b_1,\ldots,b_m\},s\rangle = \langle\{a_1,\ldots,a_n\},s\rangle$,
  ou

- **Raisonnement par cas**
  il existe une règle $\wedge_{i\in[1..n]}p_i == p'_i \Rightarrow g \rightarrow (d_1,d_2)$, *une occurrence* u *de* t, *une substitution* σ *telle que*
  $t_{/u} = \sigma(g)$ *et pour tout* i *de* $[1..n]$ $(\sigma(p_i)\downarrow_R = p"_i, (p'_i \neq p"_i)$ *et* $(p'_i \neq \neg p"_i)$
  et $\langle\{b_1,\ldots,b_m\},s\rangle$ *est l'un des a_termes suivant :*
  $\langle\{a_1,\ldots,a_n,(\sigma(p_i) == p'_i)_{i\in[1..n]},(p''_i == p'_i)_{i\in[1..n]},\},t[u \leftarrow \sigma(d_1)]\rangle$

  $\langle\{a_1,\ldots,a_n,\sigma(p_1) == \neg p'_1, p''_1 == p'_1\},t[u \leftarrow \sigma(d_2)]\rangle$,

  $\vdots$

  $\langle\{a_1,\ldots,a_n,\sigma(p_n) == \neg p'_n, p''_n == p'_n\},t[u \leftarrow \sigma(d_2)]\rangle$,
  $p''_i == p'_i$ *peut être omise si* $p''_i = p'_i$
  ou

- **Réduction avec une assertion**
  il existe un i *de* $[1..n]$, *tel que* $a_i = (t_1 == t_2)$
  et il existe une occurrence u *de* t *telle que*
  $t_{/u} = t_1$ *et* $\langle\{b_1,\ldots,b_m\},s\rangle = \langle\{a_1,\ldots,a_n\},t[u \leftarrow t_2]\rangle$.

Normalement lors du raisonnement par cas, il n'y a que deux a_équations générées avec les formules équationnelles $\wedge_{i\in[1..n]}a_i$ et $\neg(\wedge_{i\in[1..n]}a_i)$ qui sont ajoutées aux assertions. Mais après la décomposition de ces deux a_équations nous obtenons les a_équations vues dans la définitions précédentes.

Remarque 4.1

Les variables dans les assertions sont des variables fixées: constantes. Ce qui empêche de faire du filtrage entre les termes de ces assertions et le terme à réduire. C'est pourquoi la réduction avec une assertion consiste seulement à faire un remplacement de termes.

Exemple 4.1

Reprenons l'exemple 3.8.
Et soit $\langle\{estvide(X) == faux, estvide(union(X,a)) == faux, min(X) \leq max(X) == vrai\}$, $min(union(X,a)) \leq max(union(X,a)) == vrai\rangle$ *une a_équation obtenue lors de la décomposition de la formule résultat de l'induction structurelle. Cette a_équation se décompose en deux a_termes :*
$\langle\{estvide(X) == faux, estvide(union(X,a)) == faux, min(X) \leq max(X) == vrai\}$,
$min(union(X,a)) \leq max(union(X,a))\rangle$ *et*
$\langle\{estvide(X) == faux, estvide(union(X,a)) == faux, min(X) \leq max(X) == vrai\}$,
$vrai\rangle$
Une réécriture simple du premier a_terme avec la règle
$max(union(E,x)) \rightarrow sup(x,max(E))$ *a pour résultat le a_terme*
$\langle\{estvide(X) == faux, estvide(union(X,a)) == faux, min(X) \leq max(X) == vrai\}$,
$min(union(X,a)) \leq sup(a,max(X))\rangle$,
En essayant de réduire ce dernier avec la règle conditionnelle $(y < x) == vrai \Rightarrow sup(x,y) \rightarrow$

(x,y). *La condition ne s'évalue ni à vrai ni à faux. Nous allons appliquer un raisonnement par cas, et nous obtenons les deux a_termes :*
$\langle\{estvide(X) == faux, estvide(union(X,a)) == faux, min(X) \leq max(X) == vrai,$
$max(X) < a == vrai\}, min(union(X,a)) \leq a\rangle$. *La dernière assertion étant la condition supposée à vrai.*
et le a_terme
$\langle\{estvide(X) == faux, estvide(union(X,a)) == faux, min(X) \leq max(X) == vrai,$
$max(X) < a == faux\}, min(union(X,a)) \leq max(X)\rangle$ *La dernière assertion étant la condition supposée à faux.*

Contrairement à ce qui se passe pour la relation de réécriture simple, la relation de réécriture entre les a_termes admet plusieurs formes normales pour un même a_terme à cause du raisonnement par cas. Par conséquent, un a_terme n'a pas une forme normale unique, mais un ensemble fini de formes normales.

Définition 4.6 Forme normale d'un a_terme

La forme normale d'un a_terme $\langle\{a_1,\ldots,a_n\},t\rangle$ *est un a_terme* $\langle\{b_1,\ldots,b_m\},t'\rangle$ *tel que* $\langle\{a_1,\ldots,a_n\},t\rangle - a \xrightarrow{*}_R \langle\{b_1,\ldots,b_m\},t'\rangle$ *et* $\langle\{b_1,\ldots,b_m\},t'\rangle$ *est irréductible.*
L'ensemble de formes normales d'un a_terme est noté $EFNA(\langle\{a_1,\ldots,a_n\},t\rangle)$.

2.3 Comparaison avec la réécriture contextuelle

La réécriture contextuelle a été proposée pour la première fois dans [BDJ79]. Elle est définie sur les termes contextuels. Il s'agit d'ajouter la condition de la règle *si-alors*, sans qu'elle soit évaluée, dans le contexte du terme contextuel à réécrire. Elle a ensuite été utilisée dans [Rem82] dans le cadre des spécifications hiérarchiques. Nous la retrouvons aussi dans [Zha84] qui l'utilise comme une base du système *REVEUR*4. Les travaux actuels dans [BR87a], [Bou88b] utilisent aussi la réécriture contextuelle pour étudier la confluence close des systèmes de réécriture conditionnelle. La première différence c'est que nous utilisons les deux formes de règles de réécriture : La forme *si-alors* réservée aux opérations partiellement définies et aux préconditions sur les opérations. Et la forme *si-alors-sinon* utilisé pour définir les opérations dans des contextes exhaustifs. Cette deuxième forme a l'avantage de permettre la génération automatique de cas lors de la réduction d'un a_terme en créant deux nouveaux a_termes qui ont respectivement en plus des anciennes assertions la conditions de la règle et sa négation. Ces nouvelles assertions peuvent être utilisées comme des règles de réécritures pour les prochaines étapes.
Une deuxième différence c'est que nous évaluons les conditions avant de réécrire avec les règles conditionnelles. Cela est obligatoire pour pouvoir réécrire avec les règles de la forme $si-alors$. Pour les règles $si-alors-sinon$ nous évaluons aussi les conditions pour ne pas avoir une explosion combinatoire de cas. Cette évaluation de la condition peut entrainer la non terminaison de la réécriture. Mais nous allons définir dans les paragraphes suivants les conditions pour que cela ne se produise pas.

2.4 Réduction d'une a_équation

Les définitions ci-dessous seront utilisées pour simplifier celles de la section suivante.

Notation 4.1

Soient $\{a_1,\ldots,a_n\}$ et $\{b_1,\ldots,b_m\}$ deux ensembles d'équations closes.
$\{a_1,\ldots,a_n\} \uplus \{b_1,\ldots,b_m\}$ est la réunion entre ces deux ensembles. Sachant que deux équations $t_1 == t'_1$ et $t_2 == t'_2$ sont égales si et seulement si $t_1 = t_2$ et $t'_1 = t'_2$.

Définition 4.7 Réduction de la partie conclusion d'une a_équation

Soit R un système de réécriture. Une a_équation $\langle\{a_1,\ldots,a_n\}, t == s\rangle$ se réduit à la a_équation $\langle\{b_1,\ldots,b_k\}, t' == s'\rangle$, et nous notons $\langle\{a_1,\ldots,a_n\}, t == s\rangle \rightsquigarrow_R \langle\{b_1,\ldots,b_k\}, t' == s'\rangle$ si et seulement si il existe :

- *une forme normale $\langle\{a'_1,\ldots,a'_m\}, t'\rangle$, du a_terme $\langle\{a_1,\ldots,a_n\}, t\rangle$*
  et
- *une forme normale $\langle\{a"_1,\ldots,a"_l\}, s'\rangle$, du a_terme $\langle\{a_1,\ldots,a_n\}, s\rangle$*

telles que $\{b_1,\ldots,b_k\} = \{a'_1,\ldots,a'_m\} \uplus \{a''_1,\ldots,a''_l\}$

3 Validité d'une a_équation

3.1 Réduction de la partie assertion dans une a_équation

Réduire une assertion c'est calculer les formes normales de ses deux composantes en utilisant le système de réécriture initial augmenté avec les autres assertions qui l'accompagnent. Pour accomplir cette tâche nous allons définir la réduction d'une assertion comme la réduction de la a_équation composée de l'assertion à réduire dans la partie conclusion et des autres assertions dans la partie assertion.

Définition 4.8 Réduction d'une assertion

Soient R un système de réécriture. Une a_équation $\langle\{a_1,\ldots,a_n\}, e\rangle$ se réduit en une a_équation $\langle\{b_1,\ldots,b_m\}, e\rangle$ avec une réduction d'une assertion a_i et nous notons :
$\langle\{a_1,\ldots,a_n\}, e\rangle \overset{i}{\rightsquigarrow}_{aR} \langle\{b_1,\ldots,b_m\}, e\rangle$ si et seulement si :

- *$\langle\{a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_n\}, a_i\rangle \rightsquigarrow_R \langle\{c_1,\ldots,c_l\}, b\rangle$*
  et
- *$\{b_1,\ldots,b_m\} = \{a_1,\ldots,a_n\} \uplus \{c_1,\ldots,c_l\} \uplus \{b\}$.*

Les deux a_termes qui correspondent à l'assertion à réduire peuvent avoir plusieurs formes normales. Donc, la réduction d'une assertion peut générer plusieurs a_équations. Or ce travail doit être fait pour chaque assertion d'une a_équation donnée. Nous serons donc amené à composer les résultats de ces différentes réductions pour calculer l'ensemble des dérivées d'une a_équations. Cet ensemble sera calculé par approximations successives comme nous allons le voir dans la définition suivante.

Définition 4.9 Ensemble des dérivées d'une a_équation

Soient R un système de réécriture et $\langle\{a_1,\ldots,a_n\}, e\rangle$ une a_équation. L'ensemble des dérivées de cette a_équation noté par $DERIV(\langle\{a_1,\ldots,a_n\}, e\rangle)$ est défini par :

- *$DERIV_1 = \{\langle\{b_1,\ldots,b_m\}, e\rangle$, telle que $\langle\{a_1,\ldots,a_n\}, e\rangle \overset{1}{\rightsquigarrow}_{aR} \langle\{b_1,\ldots,b_m\}, e\rangle\}$.*

- *$DERIV_{i+1} = \{\langle\{c_1,\ldots,c_l\}, e\rangle$, telle que $\exists\ \langle\{a_1,\ldots,a_n,b_1,\ldots,b_{m_i}\}, e\rangle \in DERIV_i$ et $\langle\{a_1,\ldots,a_n,b_1,\ldots,b_{m_i}\}, e\rangle \overset{i}{\rightsquigarrow}_{aR} \langle\{c_1,\ldots,c_l\}, e\rangle\}$.*
- *$DERIV(\langle\{a_1,\ldots,a_n\}, e\rangle) = DERIV_n$. n est le nombre d'assertions.*

Exemple 4.2

Reprenons la a_équation de l'exemple4.1 :
$\langle\{estvide(X) == faux,\ estvide(union(X,a)) == faux,\ min(X) \leq max(X) == vrai\},$
$min(union(X,a)) \leq max(union(X,a)) == vrai\rangle$.
Avant de commencer la réduction de la partie conclusion il faut réduire les assertions, ou en d'autres termes calculer l'ensemble des dérivées de cette a_équation. Pour cela nous devons réduire les trois a_équations
$\langle\{estvide(union(X,a)) == faux,\ min(X) \leq max(X) == vrai\}, estvide(X) == faux\rangle$,
$\langle\{estvide(X) == faux,\ min(X) \leq max(X) == vrai\}, estvide(union(X,a)) == faux\rangle$
et $\langle\{estvide(X) == faux,\ estvide(union(X,a)) == faux\},\ min(X) \leq max(X) == vrai\rangle$
La réduction de la première et de la troisième a_équation ne change pas le résultat, car estvide(X) et vrai sont irréductibles, ainsi que min(X) et max(X). La réduction de la deuxième donne comme résultat la a_équation $\langle\{estvide(X) == faux,\ min(X) \leq max(X) == vrai\}, faux == faux\rangle$. Et puisque cette a_équation est une a_équation triviale l'ensemble des dérivées
$DERIV(\langle\{estvide(X) == faux,\ estvide(union(X,a)) == faux,\ min(X) \leq max(X) == vrai\}, min(union(X,a)) \leq max(union(X,a)) == vrai\rangle) =$
$\{\langle\{estvide(X) == faux, estvide(union(X,a)) == faux,\ min(X) \leq max(X) == vrai\},$
$min(union(X,a)) \leq max(union(X,a)) == vrai\rangle\}$.

Les motivations de la réduction des assertions sont multiples :

- Elle permet de détecter les inconsistances dans les assertions. Par exemple, il se peut qu'une hypothèse soit toujours vraie, ce qui sera confirmé par la réduction, alors que dans une étape de la décomposition, nous l'avons supposée fausse.
- Elle permet aussi de calculer les formes normales des assertions qui pourront elles aussi être utilisées dans la réduction de la partie conclusion qui les accompagne.
- Le raisonnement par cas au moment de la réduction des assertions permet de générer des assertions supplémentaires. Ce sont les conditions des règles utilisées lors de ce raisonnement par cas. Ces nouvelles assertions pourront alors être utilisées pour réduire la partie conclusion. Cela constitue une anticipation sur la génération de cas pour la réduction de la partie conclusion.

3.2 Validité d'une a_équation

Définition 4.10 Inconsistance d'une a_équation

Une a_équation $\langle\{a_1,\ldots,a_n\}, e\rangle$ est inconsistante si et seulement si il existe un i de $[1..n]$ tel que $a_i = (vrai == faux)$ ou $a_i = (faux == vrai)$.

Exemple 4.3

Reprenons l'exemple 3.8.
Et soit $\langle\{estvide(ensvide) == vrai\}, min(ensvide) \leq max(ensvide) == vrai\rangle$ une a_équation obtenue lors de la décomposition de la formule résultat de l'induction structurelle. L'ensemble

des dérivées de cette a_équation est égal à $\{\langle\{estvide(ensvide) == vrai, faux == vrai\}, min(ensvide) \leq max(ensvide) == vrai\rangle\}$ *qui est une a_équation inconsistante.*

Définition 4.11 Composition de deux ensemble de a_termes

Soient $\{\langle\{a_{i_1}, \ldots, a_{i_{n_i}}\}, t_i\rangle,\ i \in I\}$ *et* $\{\langle\{b_{j_1}, \ldots, b_{j_{m_j}}\},\ s_j\rangle,\ j \in J\}$ *deux ensembles de a_termes. La composition de ces deux ensembles est un ensemble de a_équations défini par*

$$\{\langle\{a_{i_1}, \ldots, a_{i_{n_i}}\} \uplus \{b_{j_1}, \ldots, b_{j_{m_j}}\},\ t_i == s_j\rangle,\ i \in I,\ j \in J\}$$

Définition 4.12 Equivalence de deux ensembles de formes normales de a_termes

Soient $\{\langle\{a_{i_1}, \ldots, a_{i_{n_i}}\}, t_i\rangle,\ i \in I\}$ *et* $\{\langle\{b_{j_1}, \ldots, b_{j_{m_j}}\},\ s_j\rangle,\ j \in J\}$ *deux ensemble de formes normales respectives des a_termes* $\langle\{a_1, \ldots, a_n\}, t\rangle$ *et* $\langle\{b_1, \ldots, b_m\}, s\rangle$, *et* $\{\langle\{c_{k_1}, \ldots, c_{k_{r_k}}\}, t_k == s_k\rangle, k \in K\}$ *leur composition.*
Ces deux ensembles de formes normales sont équivalents si et seulement si pour tout k de K :

- $t_k = s_k$
  ou
- $\langle\{c_{k_1}, \ldots, c_{k_{r_k}}\}, t_k == s_k\rangle$ *est une a_équation inconsistante.*

Définition 4.13 Théorème équationnel

Une a_équation $\langle\{a_1, \ldots, a_n\}, t_1 == t_2\rangle$ *est est un théorème équationnel si et seulement si pour toute a_équation* $\langle\{b_1, \ldots, b_m\}, s_1 == s_2\rangle$ *de* $DERIV(\langle\{a_1, \ldots, a_n\},\ t_1 == t_2\rangle)$, $EFNA(\{\langle\{b_1, \ldots, b_m\}, s_1\rangle\})$ *et* $EFNA(\{\langle\{b_1, \ldots, b_m\}, s_2\rangle\})$ *sont équivalents.*

Exemple 4.4

- *La a_équation* $\langle\{estvide(ensvide) == vrai\},\ min(ensvide) \leq max(ensvide) == vrai\rangle$ *de l'exemple 4.3 est un théorème équationnel car son ensemble de dérivées est égal à* $\{\langle\{estvide(ensvide) == vrai, faux == vrai\},\ min(ensvide) \leq max(ensvide) == vrai\rangle\}$ *qui est une a_équation inconsistante.*
- *Reprenons la a_équation*
  $\langle\{estvide(X) == faux,\ estvide(union(X,a)) == faux,\ min(X) \leq max(X) == vrai\},$
  $min(union(X,a)) \leq max(union(X,a)) == vrai\rangle$ *de l'exemple 4.1. D'après l'exemple 4.2, l'ensemble des dérivées est réduit à la a_équation elle même.*
  La réduction du a_terme
  $\langle\{estvide(X) == faux,\ estvide(union(X,a)) == faux,\ min(X) \leq max(X) == vrai\},$
  $min(union(X,a)) \leq max(union(X,a))\rangle$
  donne les quatre a_termes suivants :
  * $\langle\{estvide(X) == faux,\ estvide(union(X,a)) == faux,\ min(X) \leq max(X) == vrai,$
  $max(X) < a == vrai,\ a < min(X) == vrai\},\ a \leq a\rangle$
  * $\langle\{estvide(X) == faux,\ estvide(union(X,a)) == faux,\ min(X) \leq max(X) == vrai,$
  $max(X) < a == vrai,\ min(X) \leq a == vrai\},\ min(X) \leq a\rangle$
  * $\langle\{estvide(X) == faux,\ estvide(union(X,a)) == faux,\ min(X) \leq max(X) == vrai,$
  $a \leq max(X) == vrai,\ a < min(X) == vrai\},\ a \leq max(X)\rangle$
  * $\langle\{estvide(X) == faux,\ estvide(union(X,a)) == faux,\ min(X) \leq max(X) == vrai,$
  $a \leq max(X) == vrai,\ a < min(X) == faux,\ min(X) \leq a == vrai\},$
  $min(X) \leq max(X)\rangle$

La partie conclusion de ces quatre a_termes se reduit à vrai. Donc

$EFNA(\{\langle\{estvide(X) == faux,\ estvide(union(X,a)) == faux,$
$min(X) \leq max(X) == vrai\},\ min(union(X,a)) \leq max(union(X,a))\rangle\}) =$
$\{\langle\{estvide(X) == faux,\ estvide(union(X,a)) == faux,\ min(X) \leq max(X) == vrai,$
$max(X) < a == vrai,\ a < min(X) == vrai\},\ vrai\rangle$
$\langle\{estvide(X) == faux,\ estvide(union(X,a)) == faux,\ min(X) \leq max(X) == vrai,$
$max(X) < a == vrai,\ min(X) \leq a == vrai\},\ vrai\rangle$
$\langle\{estvide(X) == faux,\ estvide(union(X,a)) == faux,\ min(X) \leq max(X) == vrai,$
$a \leq max(X) == vrai,\ a < min(X) == vrai\},\ vrai\rangle$
$\langle\{estvide(X) == faux,\ estvide(union(X,a)) == faux,\ min(X) \leq max(X) == vrai,$
$a \leq max(X) == vrai,\ a < min(X) == faux,\ min(X) \leq a == vrai\},\ vrai\rangle\}$

$EFNA(\{\langle\{estvide(X) == faux,\ estvide(union(X,a)) == faux,$
$min(X) \leq max(X) == vrai\},\ vrai\rangle\}) =$
$\langle\{estvide(X) == faux, estvide(union(X,a)) == faux, min(X) \leq max(X) == vrai\},\ vrai\rangle$

La composition des deux ensembles de formes normales est égale à
$\{\langle\{estvide(X) == faux,\ estvide(union(X,a)) == faux,\ min(X) \leq max(X) == vrai,$
$max(X) < a == vrai,\ a < min(X) == vrai\},\ vrai == vrai\rangle$
$\langle\{estvide(X) == faux,\ estvide(union(X,a)) == faux,\ min(X) \leq max(X) == vrai,$
$max(X) < a == vrai,\ min(X) \leq a == vrai\},\ vrai == vrai\rangle$
$\langle\{estvide(X) == faux,\ estvide(union(X,a)) == faux,\ min(X) \leq max(X) == vrai,$
$a \leq max(X) == vrai,\ a < min(X) == vrai\},\ vrai == vrai\rangle$
$\langle\{estvide(X) == faux,\ estvide(union(X,a)) == faux,\ min(X) \leq max(X) == vrai,$
$a \leq max(X) == vrai,\ a < min(X) == faux,\ min(X) \leq a == vrai\},\ vrai == vrai\rangle\}$
Toutes ces a_équations ont leur partie conclusion qui contient vrai == vrai, donc vérifie la deuxième condition de la définition 4.13. On peut conclure que la a_équation
$\langle\{estvide(X) == faux,\ estvide(union(X,a)) == faux,\ min(X) \leq max(X) == vrai\},$
$min(union(X,a)) \leq max(union(X,a)) == vrai\rangle$
est un théorème équationnel

4 Correction de la réécriture sur les a_équations

L'objectif de cette section est de prouver que chaque étape de la réécriture dans une a_équation ne change rien à la validité de celle-ci. Ces preuves seront un intermédiaire pour prouver des propriétés plus générales sur la validité. Ces propriétés concernent la relation entre la validité d'une a_équation et la validité de l'ensemble de ses dérivées, puis la relation entre la validité opérationnelle que nous avons vu dans la définition 4.13. et la validité formelle d'une a_équation.

4.1 Correction des étapes intermédiaires

Les étapes de la réécriture dans les a_équations seront formulées à l'aide de règles d'inférences. Le dénominateur contient la a_équation initiale et le numérateur contient la ou les a_équations obtenues après l'application de la réécriture. Une telle règle d'inférence peut être lue de la façon suivante: la partie dénominateur est valide si et seulement si la partie numérateur l'est aussi. Cette nouvelle présentation des relations de réécriture sous forme de règles d'inférence

permettra de bien comprendre les définitions de ces relations [Laz90a]. Pour la preuve de correction nous allons utiliser la sémantique donnée aux règles d'inférences dans la définition 2.3. Nous commençons d'abord par la correction de la réécriture sur les termes.
Soit E un ensemble d'équations et R le système de réécriture associé.

Lemme 4.1 Réduction simple d'un terme

Soient t et t' deux termes.

$$t \longrightarrow_R t' \implies t == t' \text{ est valide dans } M(E)$$

Preuve :

Pour prouver ce lemme il faut prouver que pour toute algèbre A de $M(E)$, A valide $t == t'$.
Supposons le contraire, alors il existe une algèbre A de $M(E)$ et un homomorphisme $v' : T(F,X) \longrightarrow A$, telle que $v'(t) \neq v'(t')$.
$t \longrightarrow_R t'$ implique qu'il existe une équation e de E, une occurrence u de t et une substitution θ qui vérifient les conditions de la définition 4.1.
Or l'équation e peut avoir trois formes:

- $e = (g == d)$, qui est valide dans toute algèbre de $M(E)$. En particulier dans A. Donc pour tout homomorphisme $v : T(F,X) \longrightarrow A$, $v(g) = v(d)$ et par conséquent $v(\theta(g)) = v(\theta(d))$. Ce qui implique que $v(t) = v(t')$. Ce qui contredit le fait que A ne valide pas $t == t'$.
- $e = (\wedge_{i\in[1..n]} p_i == p'_i) \Rightarrow g == d$, qui est valide dans toute algèbre de $M(E)$ et en particulier dans A. Or d'après la définition de $t \longrightarrow_R t'$, $\forall\, i \in [1..n], \theta(p_i) = p'_i$ donc $v(\theta(p_i)) = v(p'_i)$, ce qui implique que $v(\theta(g)) = v(\theta(d))$ et par conséquent que $v(t) = v(t')$ ce qui contredit l'hypothèse initiale.
- $e = (\wedge_{i\in[1..n]} p_i == p'_i) \Rightarrow g == (d_1, d_2)$. Deux cas peuvent se présenter :
  - $\forall i \in [1..n], \theta(p_i) = p'_i$ donc $v(\theta(p_i)) = v(p'_i)$, ce qui implique que $v(\theta(g)) = v(\theta(d_1))$ et par conséquent que $v(t) = v(t')$ ce qui contredit l'hypothèse initiale.
  - $\exists i \in [1..n]$ tel que $\theta(p_i) = \neg p'_i$. Donc $v(\theta(p_i) = \neg v(p'_i)$. Ce qui implique que $v(\theta(g)) = v(\theta(d_2))$ et par conséquent que $v(t) = v(t')$ ce qui contredit l'hypothèse initiale.

Lemme 4.2

La règle d'inférence

$$\frac{\begin{array}{c}\langle\{a_1,\ldots,a_n,(p_i == p'_i)_{i\in[1..m]}\},e\rangle \\ \langle\{a_1,\ldots,a_n,p_1 == \neg p'_1\},e\rangle \\ \vdots \\ \langle\{a_1,\ldots,a_n,p_m == \neg p'_m\},e\rangle\end{array}}{\langle\{a_1,\ldots,a_n\},e\rangle}$$

est correcte.

On suppose que pour tout $i \in [1..n]$, $p_i == p'_i$ est une équation booléenne close sur $T(\Sigma)$ telle que que $p'_i = vrai$ ou $p'_i = faux$,

Preuve :

Il s'agit de montrer que $\langle\{a_1,\ldots,a_n,\ (p_i == p'_i)_{i\in[1..m]}\},e\rangle$, $\langle\{a_1,\ldots,a_n,p_1 == \neg p'_1\},e\rangle \ldots \langle\{a_1,\ldots,a_n,p_m == \neg p'_m\},e\rangle$ sont valides dans une algèbre A si et seulement si $\langle\{a_1,\ldots,a_n\},e\rangle$ est valide dans A.
$\langle\{a_1,\ldots,a_n,(p_i == p'_i)_{i\in[1..m]}\},e\rangle, \langle\{a_1,\ldots,a_n,p_1 == \neg p'_1\},e\rangle \ldots \langle\{a_1,\ldots,a_n,p_m == \neg p'_m\},e\rangle$ sont valides dans A si et seulement si
$A \models_\Sigma (\wedge_{i\in[1..n]} a_i \wedge_{i\in[1..m]} p_i == p'_i) \Longrightarrow A \models_\Sigma e \wedge$
$A \models_\Sigma (\wedge_{i\in[1..n]} a_i \wedge p_1 == \neg p'_1) \Longrightarrow A \models_\Sigma e \wedge$
$\vdots$
$A \models_\Sigma (\wedge_{i\in[1..n]} a_i \wedge p_n == \neg p'_n) \Longrightarrow A \models_\Sigma e$
Ce qui est équivalent à
$(A \models_\Sigma (\wedge_{i\in[1..n]} a_i) \wedge A \models_\Sigma \wedge_{i\in[1..m]} p_i == p'_i) \Longrightarrow A \models_\Sigma e \wedge$
$(A \models_\Sigma (\wedge_{i\in[1..n]} a_i) \wedge A \models_\Sigma p_1 == \neg p'_1) \Longrightarrow A \models_\Sigma e \wedge$
$\vdots$
$(A \models_\Sigma (\wedge_{i\in[1..n]} a_i) \wedge A \models_\Sigma p_n == \neg p'_n) \Longrightarrow A \models_\Sigma e$
Ce qui est équivalent à
$(A \models_\Sigma (\wedge_{i\in[1..n]} a_i) \wedge (A \models_\Sigma \wedge_{i\in[1..m]} p_i == p'_i \vee A \models_\Sigma p_1 == \neg p'_1 \vee \ldots \vee A \models_\Sigma p_n == \neg p'_n) \Longrightarrow A \models_\Sigma e$ (lemme 2.3 et distributivité du $\wedge$ par rapport au $\vee$).
Ce qui équivalent à
$A \models_\Sigma (\wedge_{i\in[1..n]} a_i) \wedge (\wedge_{i\in[1..m]} p_i == p'_i \vee p_1 == \neg p'_1 \vee \ldots \vee p_n == \neg p'_n) \Longrightarrow A \models_\Sigma e$
Ce qui est équivalent à
$A \models_\Sigma (\wedge_{i\in[1..n]} a_i) \wedge (\wedge_{i\in[1..m]} p_i == p'_i \vee \neg(\wedge_{i\in[1..m]} p_i == p'_i)) \Longrightarrow A \models_\Sigma e$
Ce qui est équivalent à
$A \models_\Sigma (\wedge_{i\in[1..n]} a_i) \Longrightarrow A \models_\Sigma e$
Ce qui est équivalent à
$A \models_\Sigma \langle\{a_1,\ldots,a_n\},e\rangle$.

Lemme 4.3

Soient a, b et c trois propositions.

$$(((a \wedge b) \Longrightarrow c) \wedge (a \Longrightarrow b)) \Longrightarrow (a \Longrightarrow c)$$

Lemme 4.4

$$(((a \Longrightarrow b) \Longrightarrow (a \Longrightarrow c)) \Longrightarrow ((a \wedge b) \Longrightarrow c)$$

Proposition 4.1 réduction simple

La règle d'inférence

$$\frac{\langle\{a_1,\ldots,a_n\},t_1 == t'_2\rangle}{\langle\{a_1,\ldots,a_n\},t_1 == t_2\rangle}$$

avec $t_2 \longrightarrow_R t'_2$

est correcte

Preuve :

Il s'agit de montrer que $\langle\{a_1,\ldots,a_n\},t_1 == t'_2\rangle$ est valide dans une algèbre A si et seulement si $\langle\{a_1,\ldots,a_n\},t_1 == t_2\rangle$

- $\Longrightarrow$
  Supposons que $\langle\{a_1,\ldots,a_n\}, t_1 == t_2\rangle$ ne soit pas valide dans A. Ce qui implique qu'il existe une valuation v, telle que pour tout i de $[1..n]$, $v(a_i) = vrai$ et $v(t_1) \neq v(t_2)$. Or d'après le lemme 4.1, $t_2 == t'_2$ est valide dans A, ce qui implique que $v(t_2) = v(t'_2)$. Or $\langle\{a_1,\ldots,a_n\}, t_1 == t'_2\rangle$ est valide dans A et $v(a_i) = vrai$ implique que $v(t_1) = v(t'_2)$. Et par transitivité de l'égalité $v(t_1) = v(t_2)$. Ce qui contredit l'hypothèse.
- $\Longleftarrow$ Même démarche

Proposition 4.2 Raisonnement par cas

La règle d'inférence

$$\frac{\begin{array}{c}\langle\{a_1,\ldots,a_n,(p_i == p'_i)_{i\in[1..m]}\}, t_1 == t'_2\rangle \\ \langle\{a_1,\ldots,a_n,p_1 == \neg p'_1\}, t_1 == t''_2\rangle \\ \vdots \\ \langle\{a_1,\ldots,a_n,p_m == \neg p'_m\}, t_1 == t''_2\rangle\end{array}}{\langle\{a_1,\ldots,a_n\}, t_1 == t_2\rangle}$$

telle que il existe une règle $\wedge_{i\in[1..m]}p_i == p'_i \Rightarrow g \rightarrow (d_1, d_2)$ *de* R, *une occurrence* u *de* t_2 *et une substitution* σ *telle que*

- $t_{2/u} = \sigma(g)$,
- *pour tout* i *de* $[1..m]$, $(\sigma(p_i) \downarrow_R = p''$, $(p'_i \neq p''_i)$ *et* $(p'_i \neq \neg p''_i))$
  et
- $t'_2 = t_2[u \leftarrow \sigma(d_1)]$ *et* $t''_2 = t_2[u \leftarrow \sigma(d_2)]$.

est correcte.

Preuve :
Il s'agit de montrer que les a_équations $\langle\{a_1,\ldots,a_n,(p_i == p'_i)_{i\in[1..m]}\}, t_1 == t'_2\rangle$, $\langle\{a_1,\ldots,a_n,p_1 == \neg p'_1\}, t_1 == t''_2\rangle$, ..., $\langle\{a_1,\ldots,a_n,p_m == \neg p'_m\}, t_1 == t''_2\rangle$ sont valides dans une algèbre A si et seulement si $\langle\{a_1,\ldots,a_n\}, t_1 == t_2\rangle$ est valide dans A.

$\langle\{a_1,\ldots,a_n\}, t_1 == t_2\rangle$ est valide si et seulement si
$\langle\{a_1,\ldots,a_n,(\sigma(p_i) == p'_i)_{i\in[1..m]}\}, t_1 == t_2\rangle$, $\langle\{a_1,\ldots,a_n,\sigma(p_1) == \neg p'_1\}, t_1 == t_2\rangle$, ..., $\langle\{a_1,\ldots,a_n,\sigma(p_m) == \neg p'_m\}, t_1 == t_2\rangle$ sont valides (lemme 4.2).
Or

- $\langle\{a_1,\ldots,a_n,(\sigma(p_i) == p'_i)_{i\in[1..m]}\}, t_1 == t_2\rangle$ est valide si et seulement si
  $\langle\{a_1,\ldots,a_n,(\sigma(p_i) == p'_i)_{i\in[1..m]}\}, t_1 == t'_2\rangle$ est valide (proposition 4.1),
  avec $t_2 \longrightarrow_R t'_2$ en utilisant la règle $\wedge_{i\in[1..m]}p_i == p'_i \Rightarrow g \rightarrow (d_1, d_2)$ et la substitution σ et telle que pour tout i de $[1..n]$, $\sigma(p_i) = p'_i$.

- pour tout j de $[1..m]$
  $\langle\{a_1,\ldots,a_n,\sigma(p_j) == \neg p'_j\}, t_1 == t_2\rangle$ est valide si et seulement si
  $\langle\{a_1,\ldots,a_n,\sigma(p_j) == \neg p'_j\}, t_1 == t''_2\rangle$ est valide, (proposition 4.1),
  avec $t_2 \longrightarrow_R t''_2$ en utilisant la règle $\wedge_{i\in[1..m]}p_i == p'_i \Rightarrow g \rightarrow (d_1, d_2)$, la substitution σ et telle que $\sigma(p_j) = \neg p'_j$.

Proposition 4.3 Réduction avec une assertion

La règle d'inférence

$$\frac{\langle\{a_1,\ldots,a_n\}, t_1 == t'_2\rangle}{\langle\{a_1,\ldots,a_n\}, t_1 == t_2\rangle}$$

telle qu' il existe une assertion $a_i = (g == d)$, *une occurrence* u *de* t_2 *telle que* $t_{2/u} = g$ *et* $t'_2 = t_2[u \leftarrow d]$
est correcte

Preuve :
Il s'agit de montrer que $\langle\{a_1,\ldots,a_n\}, t_1 == t'_2\rangle$ est valide dans une algèbre A si et seulement si $\langle\{a_1,\ldots,a_n\}, t_1 == t_2\rangle$ est valide dans A.

- $\Longrightarrow$
  Supposons le contraire, donc il existe une valuation v, telle que $v(a_j) = vrai$ pour tout $j \in [1..n]$ et $v(t_1) \neq v(t_2)$. En particulier $v(a_i) = vrai$. Donc $v(g) = v(d)$, ce qui implique que $v(t_2) = v(t'_2)$. Or $t_1 == t'_2$ est valide dans A ce qui implique que $v(t_1) = v(t'_2)$. Et par transitivité de l'égalité $v(t_1) = v(t_2)$. Ce qui contredit l'hypothèse initiale.
- $\Longleftarrow$ Même démarche.

Proposition 4.4 Réduction simple d'une assertion

La règle d'inférence

$$\frac{\langle\{a_1,\ldots,a_{i-1},t_{i_1} == t_{i_2},a_{i+1},\ldots,a_n,t_{i_1} == t'_{i_2}\},e\rangle}{\langle\{a_1,\ldots,a_{i-1},t_{i_1} == t_{i_2},a_{i+1},\ldots,a_n\},e\rangle}$$

avec $t_{i_2} \longrightarrow_R t'_{i_2}$
est correcte.

Preuve
Il s'agit de montrer que $\langle\{a_1,\ldots,a_{i-1},t_{i_1} == t_{i_2},a_{i+1},\ldots,a_n,t_{i_1} == t'_{i_2}\},e\rangle$ est valide dans une algèbre A si et seulement si $\langle\{a_1,\ldots,a_{i-1},t_{i_1} == t_{i_2},a_{i+1},\ldots,a_n\},e\rangle$ est valide dans A.

- $\Longrightarrow$
  $A \models_\Sigma \langle\{a_1,\ldots,a_n,t_{i_1} == t'_{i_2}\},e\rangle$ est équivalent à
  $A \models_\Sigma (\wedge_{i\in[1..n]}a_i \wedge t_{i_1} == t'_{i_2}) \Longrightarrow A \models_\Sigma e$. Ce qui est équivalent à

$(A \models_\Sigma \wedge_{i\in[1..n]} a_i \wedge A \models_\Sigma t_{i_1} == t'_{i_2}) \Longrightarrow A \models_\Sigma e$. **(1)**
Or $A \models_\Sigma t_{i_1} == t_{i_2} \Longrightarrow A \models_\Sigma t_{i_1} == t'_{i_2}$ ce qui implique que
$A \models_\Sigma \wedge_{i\in[1..n]} a_i \Longrightarrow A \models_\Sigma t_{i_1} == t'_{i_2}$ **(2)**
(1) *et* (2) $\Longrightarrow (A \models_\Sigma (\wedge_{i\in[1..n]} a_i) \Longrightarrow A \models_\Sigma e)$. (lemme 4.3)
Ce qui implique
$A \models_\Sigma (\{a_1, \ldots, a_n\}, e)$

- $\Longleftarrow$
  $A \models_\Sigma (\{a_1, \ldots, a_n\}, e)$ est équivalent à
  $A \models_\Sigma (\wedge_{i\in[1..n]} a_i) \Longrightarrow A \models_\Sigma e$. **(1)**
  Or $A \models_\Sigma t_{i_1} == t_{i_2} \Longrightarrow A \models_\Sigma t_{i_1} == t'_{i_2}$. Ce qui implique que
  $A \models_\Sigma \wedge_{i\in[1..n]} a_i \Longrightarrow A \models_\Sigma t_{i_1} == t'_{i_2}$ **(2)**
  (1) *et* (2) $\Longrightarrow ((A \models_\Sigma (\wedge_{i\in[1..n]} a_i) \wedge A \models_\Sigma t_{i_1} == t'_{i_2}) \Longrightarrow A \models_\Sigma e$. Ce qui est équivalent à
  $A \models_\Sigma \wedge_{i\in[1..n]} a_i \wedge t_{i_1} == t'_{i_2} \Longrightarrow A \models_\Sigma e$. Ce qui est équivalent à
  $A \models_\Sigma (\{a_1, \ldots, a_n, t_{i_1} == t'_{i_2}\}, e)$.

Proposition 4.5 Réduction d'une assertion avec une assertion

La règle d'inférence

$$\frac{\langle \{a_1, \ldots, a_{i-1}, t_{i_1} == t_{i_2}, a_{i+1}, \ldots, a_n, t_{i_1} == t'_{i_2}\}, e\rangle}{\langle \{a_1, \ldots, a_{i-1}, t_{i_1} == t_{i_2}, a_{i+1}, \ldots, a_n\}, e\rangle}$$

telle qu' il existe une assertion $a_j = (g == d)$, *une occurrence* u *de* t_{i_2} *telle que* $t_{i_2/u} = g$ *et* $t'_{i_2} = t_{i_2}[u \leftarrow d]$

Preuve :
Même démonstration que la proposition précédente.

Proposition 4.6 Raisonnement par cas dans une assertion

La règle d'inférence

$$\frac{\begin{array}{c}\langle \{a_1, \ldots, a_{j-1}, t_{j_1} == t_{j_2}, a_{j+1}, \ldots, a_n, t_{j_1} == t'_{j_2}, (p_i == p'_i)_{i\in[1..m]}\}, e\rangle \\ \langle \{a_1, \ldots, a_{j-1}, t_{j_1} == t_{j_2}, a_{j+1}, \ldots, a_n, t_{j_1} == t''_{j_2}, p_1 == \neg p'_1\}, e\rangle \\ \vdots \\ \langle \{a_1, \ldots, a_{j-1}, t_{j_1} == t_{j_2}, a_{j+1}, \ldots, a_n, t_{j_1} == t''_{j_2}, p_m == \neg p'_m\}, e\rangle\end{array}}{\langle \{a_1, \ldots, a_n\}, e\rangle}$$

telle que il existe une règle $\wedge_{i\in[1..m]} p_i == p'_i \Rightarrow g \rightarrow (d_1, d_2)$ *de* R, *une occurrence* u *de* t_{j_2} *et une substitution* σ *telle que*

- $t_{j_{2/u}} = \sigma(g)$,
- *pour tout* i *de* $[1..m]$, $(\sigma(p_i) \downarrow_R = p''$ $(p'_i \neq p''_i)$ *et* $(p'_i \neq \neg p''_i))$
  et

- $t'_{j_2} = t_{j_2}[u \leftarrow \sigma(d_1)]$ *et* $t''_{j_2} = t_{j_2}[u \leftarrow \sigma(d_2)]$.

est correcte

Preuve :
Il s'agit de montrer que les a_équations
$\langle \{a_1, \ldots, a_{j-1}, t_{j_1} == t_{j_2}, a_{j+1}, \ldots, a_n, t_{j_1} == t'_{j_2}, (p_i == p'_i)_{i\in[1..m]}\}, e\rangle$,
$\langle \{a_1, \ldots, a_{j-1}, t_{j_1} == t_{j_2}, a_{j+1}, \ldots, a_n, t_{j_1} == t''_{j_2}, p_1 == \neg p'_1\}, e\rangle$,
$\ldots$, $\langle \{a_1, \ldots, a_{j-1}, t_{j_1} == t_{j_2}, a_{j+1}, \ldots, a_n, t_{j_1} == t''_{j_2}, p_m == \neg p'_m\}, e\rangle$ sont valides dans une algèbre A si et seulement si $\langle \{a_1, \ldots, a_n\}, e\rangle$ est valide dans A.

$\langle \{a_1, \ldots, a_n\}, e\rangle$ est valide si et seulement si $\langle \{a_1, \ldots, a_n, (\sigma(p_i) == p'_i)_{i\in[1..m]}\}, e\rangle$, $\langle \{a_1, \ldots, a_n, \sigma(p_1) == \neg p'_1\}, e\rangle$, $\ldots$, $\langle \{a_1, \ldots, a_n, \sigma(p_m) == \neg p'_m\}, e\rangle$ sont valides (lemme 4.2).
Or

- $\langle \{a_1, \ldots, a_{j-1}, t_{j_1} == t_{j_2}, a_{j+1}, \ldots, a_n, (\sigma(p_i) == p'_i)_{i\in[1..m]}\}, e\rangle$ est valide si et seulement si
  $\langle \{a_1, \ldots, a_{j-1}, t_{j_1} == t_{j_2}, a_{j+1}, \ldots, a_n, t_{j_1} == t'_{j_2}, (\sigma(p_i) == p'_i)_{i\in[1..m]}\}, e\rangle$ est valide (proposition4.4),
  avec $t_{j_2} \longrightarrow_R t'_{j_2}$ en utilisant la règle $\wedge_{i\in[1..m]} p_i == p'_i \Rightarrow g \rightarrow (d_1, d_2)$, la substitution σ et telles que pour tout i de $[1..m]$, $\sigma(p_i) = p'_i$
- pour tout i de $[1..m]$,
  $\langle \{a_1, \ldots, a_{j-1}, t_{j_1} == t_{j_2}, a_{j+1}, \ldots, a_n, \sigma(p_i) == \neg p'_i\}, e\rangle$ est valide si et seulement si
  $\langle \{a_1, \ldots, a_{j-1}, t_{j_1} == t_{j_2}, a_{j+1}, \ldots, a_n, \sigma(p_i) == \neg p'_i, t_{j_1} == t''_{j_2}\}, e\rangle$ est valide (Proposition 4.4),
  avec $t_{j_2} \longrightarrow_R t''_{j_2}$ en utilisant la règle $\wedge_{i\in[1..m]} p_i == p'_i \Rightarrow g \rightarrow (d_1, d_2)$, la substitution σ et telles que $\sigma(p_i) = \neg p'_i$.

Remarque 4.2

Dans les propositions ci-dessus nous avons prouvé la correction des règles d'inférence dans lesquelles nous ne réécrivons que le terme gauche d'une équation. La réécriture du terme droit donnera des règles d'inférence semblables. Et la preuve de correction et la même.

4.2 Correction de la stratégie globale

Les deux propositions suivantes nous permettront de prouver la correction de la stratégie de preuve de validité d'une a_équation.

Proposition 4.7

Une a_équation $\langle \{a_1, \ldots, a_n\}, e\rangle$ *est équationnellement valide si toute a_équation de* $DERIV(\langle \{a_1, \ldots, a_n\}, e\rangle)$ *est équationnellement valide.*

Preuve :
La preuve de cette propriété passe par la preuve des propositions 4.4, 4.5 et 4.6 car chacune des étapes du calcul de l'ensemble des dérivées utilise l'une de ces règles d'inférence

Proposition 4.8

Si une a_équation est un théorème équationnel alors elle est équationnellement valide.

Preuve :

Il s'agit de prouver que la définition 4.13 implique les conditions du théorème 2.4.
Soit $\langle\{a_1,\ldots,a_n\},e\rangle$ une a_équation dont nous voulons prouver la validité équationnelle. Supposons que pour toute a_équation $\langle\{b_1,\ldots,b_m\},t_1 == t_2\rangle$ de $DERIV(\langle\{a_1,\ldots,a_n\},e\rangle)$, $EFNA(\langle\{b_1,\ldots,b_m\},t_1\rangle)$ et $EFNA(\langle\{b_1,\ldots,b_m\},t_2\rangle$ sont équivalents. Soit $\{\langle\{c_{k_1},\ldots,c_{k_{r_k}}\},t_{k_1} == t_{k_2}\rangle, k \in K\}$ l'ensemble composition de ces deux ensembles de formes normales. Pour chaque a_équation $\langle\{c_{k_1},\ldots,c_{k_{r_k}}\},t_{k_1} == t_{k_2}\rangle$ de cet ensemble et d'après la définition de l'équivalence, l'un des deux cas est possible :

- $t_{k_1} = t_{k_2}$ et dans ce cas nous avons $t_{k_1} == t_{k_2}$ est valide dans $M(E \cup \{c_{k_1},\ldots,c_{k_{r_k}}\})$. Ce qui vérifie la deuxième condition du théorème 2.4
  ou
- $\langle\{c_{k_1},\ldots,c_{k_{r_k}}\},t_{k_1} == t_{k_2}\rangle$ est inconsistante, donc il existe un $j \in [1..k_{r_k}]$ tel que $c_j = (vrai == faux)$ qui est une équation qui n'est valide dans aucun algèbre. Ce qui vérifie la première condition du théorème 2.4.

Ces a_équations de l'ensemble $\{\langle\{c_{k_1},\ldots,c_{k_{r_k}}\},t_{k_1} == t_{k_2}\rangle, k \in K\}$ qui sont toute équationnellement valides ont été obtenues en utilisant les règles d'inférences des propositions 4.1, 4.2 et 4.3. Ce qui implique que $\langle\{b_1,\ldots,b_m\},t_1 == t_2\rangle$ est équationnellement valide. Or cela est vrai pour chaque a_équation de $DERIV(\langle\{a_1,\ldots,a_n\},e\rangle)$, donc d'après la proposition 4.7, $\langle\{a_1,\ldots,a_n\},e\rangle$ est équationnellement valide.

5 Calcul des ensembles de formes normales

5.1 Stratégie de calcul des formes normales

Définir une grande spécification est une tâche difficile. La notion de hiérarchie dans la construction des spécifications [Rem82] permet de réduire cette difficulté, en manipulant des spécifications plus petites et plus compréhensibles. Nous pouvons voir cette hiérarchie quand une composante d'une spécification est définie à partir d'autres composantes. Par exemple, pour définir la spécification du type *table* (cf exemple 4.5 ci-dessous) nous avons besoin de définir le support du domaine de la *table* (Le type *ensemble*). Nous pouvons voir aussi la hiérarchie quand une opération est définie à partir d'une autre. Par exemple pour définir l'opération d'union de deux ensembles, il faut définir l'opération d'appartenance d'un élément à un ensemble.
Notre objectif est d'utiliser cette hiérarchie dans la définition des différentes composantes d'une spécification pour la réduction des termes. Normalement, pour réduire un terme nous devons parcourir toutes les règles du système de réécriture, jusqu'à en trouver une qui réécrit se terme. Or pourquoi chercher à réduire un terme sur le type table avec des règles qui correspondent aux axiomes qui définissent le type ensemble. Ce que nous voulons c'est chercher la forme normale d'un terme sur le type *table*, avec les règles qui définissent ce type, ensuite chercher la forme normale de ce résultat avec les règles du type *ensemble*.
En résumé, cette hiérarchie permet à l'utilisateur de décomposer la spécification d'un problème en des ensembles disjoints d'équations. Chaque ensemble donne une définition complète d'une composante du problème. Mais il n'est pas contraint à faire cette partition, il peut donner sa spécification en un seul ensemble. Cependant, nous ne supposons aucune contrainte sur la

manière de construction des équations.
Cette notion de hiérarchie diffère de celle proposée dans [Rem82] et [BR87a], où la hiérarchie intervient dans la construction des équations conditionnelles. En effet si une équation appartient à un niveau i alors la condition doit appartenir au niveau précédent. Pour se différencier nous parlerons de spécification par partie.

Définition 4.14 spécification par partie

Une spécification par partie à (n+1) niveaux $(SP_0, SP_1, \ldots, SP_n)$ est un n+1 uplet de spécifications, tel que pour chaque spécification $SP_i = (S_i, F_i, E_i)$ $i \in [0..n-1]$
$S_i \subseteq S_{i+1}, F_i \subseteq F_{i+1}$ et $E_i \cap E_{i+1} = \emptyset$.
Un système de réécriture par partie $R = (R_0 \cup \ldots \cup R_n)$ est le système de réécriture associé à la spécification par partie $(SP_0, \ldots, SP_n)$.

Définition 4.15 Forme normale par partie

Soient SP une spécification par partie avec p composantes $SP_1, \ldots, SP_p$,
$R = R_1 \cup \ldots \cup R_p$ le système de réécriture associé, Soit t un terme de la $i^{ème}$ composante de la spécification. ie $t \in T(F_i, X) - T(F_{i-1}, X)$.
On définit la forme normale par partie du terme t selon le système de réécriture R par

$$t \downarrow_R = (\ldots((t \downarrow_{R_i}) \downarrow_{R_{i-1}}) \ldots \downarrow_{R_1}).$$

Remarque 4.3

Dans le cas d'un système de réécriture convergent, la forme normale d'un terme ne dépend pas de la manière dont elle est calculée. Dans ce cas la forme normale par partie est équivalente à la forme normale.

Exemple 4.5

Pour définir la spécification du type table *nous aurons besoin de définir le support du domaine d'une table qui est le type* ensemble ordonné *(cf exemple 1.13).*

Type ensemble

$S_0 = \{ens\_ord, entier, bool\}$
$F_0 = \{ens\_vide, union, difference, estvide, min, max, =, <\}$
avec les profils définis dans l'exemple 1.13.
$E_0 =$
axiomes

difference(ensvide,xi) == ensvide
(xi = yi) == vrai ⇒ difference (union(xE,xi),yi) ==
(difference(xE,yi), union(difference(xE,yi),xi))
appartienta(ensvide,xi) == faux
(xi = yi) == vrai ⇒ appartienta(union(xE,xi),yi) == (vrai, appartienta(xE,yi))
estvide(ensvide) == vrai
estvide(union(xE,xi)) == faux
card(ensvide) == 0
card(union(xE,xi)) == (card(xE)+1)
min(ensvide) == 1000
min(union(xE,xi)) == inf(xi , min(xE))

max(ensvide) == 0
max(union(xE,xi)) == sup(xi , max(xE))
(xi < yi) == vrai ⇒ inf(xi , yi) == (xi, yi)
(xi > yi) ⇒ sup(xi, yi) == (xi, yi)
f-axiomes

Type table
$S_1 = \{table, ens\_ord, entier, bool\}$
$F_1 = \{tablevide, mettre, enlever, acces, domaine, maximum, minimum, ens\_vide, union, diff erence, estvide, min, max, =, <\}$ *avec les profils des nouvelles fonctions suivant:*
tablevide : ⟶ table
mettre : table × ELM ⟶ table
enlever : table × entier ⟶ table
acces : table × entier ⟶ ELM
domaine : table ⟶ ens_ord
maximum : table ⟶ entier
minimum : table ⟶ entier
$E_1 =$
axiomes

enlever(tablevide,xi) == tablevide
(xi = xj) == vrai ⇒ enlever (mettre(xt,xi,x),xj) ==
(enlever(xt,xj), mettre(enlever(xt,xj),xi,x))
acces(tablevide,y) == w
(xi = xj) == vrai ⇒ acces(mettre(xt,xi,x),xj) == (x, acces(xt,xj))
domaine(tablevide) == ensvide
domaine(mettre(xt,xi,x)) == union(domaine(xt),xi)
maximum(t) == max(domaine(t))
minimum(t) == min(domaine(t))

f-axiomes
Pour calculer la forme normale du terme "maximum(mettre(xt, xi, v))" Nous cherchons d'abord la forme normale dans le type table. C'est le terme "max(union(domaine(xt), xi))". La forme normale de ce dernier terme dans le type ensemble est égale à lui même.

Cette stratégie de calcul de la forme normale d'un terme, permet de gagner du temps et d'améliorer l'efficacité du système de preuve. Ainsi pour un système de réécriture divisé en p blocs de n règles de réécriture en moyenne, nous gagnons $n * p^2$ tests par rapport à la stratégie de réduction normale. Pour cela nous supposons que deux passages sont nécessaires dans chaque bloc: une fois pour réduire le terme et un autre passage pour vérifier qu'il n'existe pas d'autres règles qui réduisent le résultat.

5.2 Calculabilité de la forme normale

La calculabilité de la forme normale d'un terme n'est pas toujours garantie pour tous les systèmes de réécriture conditionnelle.

Exemple 4.6
Soit R un système de réécriture défini par: $R = \{r_1 : si\ f(f(x)) = 0\ alors\ f(x) = 0\}$.

Calculons f(0)

f(0) = si f(f(0)) = 0 alors 0
↓
si f(f(f(0))) = 0 alors 0
↓
si f(f(f(f (0)))) = 0 alors 0
↓
⋮
si f(...f(0)...) =0 alors ...

Nous remarquons que le calcul de f(0) diverge, car le la taille de la condition augmente indéfiniment. Nous pouvons l'expliquer par le fait que dans la définition de la règle r_1 le terme f(f(x)) était plus *grand* que f(x). En d'autres termes la condition est plus *grande* que la conclusion. Si nous nous restreignons au systèmes de réécriture qui n'ont que des règles avec des conditions plus petites que leurs conclusions, nous pouvons démontrer que la forme normale d'un terme est calculable. Définissons formellement un ordre sur les termes pour pouvoir les comparer.

Définition 4.16 Précédence par partie

Soit $R = R_1 \cup \ldots \cup R_n$ *un système de réécriture par partie et soit pour tout* $i \in [1..n] <_i$ *une relation d'ordre appelée précédence définie sur* $F_{i-1} - F_i$. *La relation d'ordre* $<$ *égale à* $<_1 + \ldots + <_n$ *est définie par :*
pour tout $f \in F_i - F_{i-1}$ *et pour tout* $g \in F_j - F_{j-1}$ *:*

- *si i=j alors* $f <_i g \Rightarrow f < g$
- *si* $i < j$ *alors* $f < g$.

Définition 4.17 Compatibilité avec le partitionnement

Soit $R = R_1 \cup \ldots \cup R_n$ *un système de réécriture par partie et soit* $<$ *un ordre sur les termes.* $<$ *est compatible avec le partitionnement si pour tous termes s, t tels que* $s \in T(F_i, X) - T(F_{i-1}, X)$ *et* $t \in T(F_{i-1}, X)$ *et tel que* $var(t) \subseteq var(s)$ *alors* $t < s$.

Définition 4.18 Ordre de réduction

Un ordre $<$ *sut* $T(F, X)$ *est un ordre de réduction, si et seulement si il possède les propriétés suivantes:*

- $<$ *est bien fondé (il n'existe pas de chaîne infini de termes* $\ldots, t_2 < t_1$*),*
- *(propriété de compatibilité):* $t_1 < t_2 \Rightarrow f(\ldots, t_1, \ldots) < f(\ldots, t_2, \ldots)$
- *(propriété de stabilité par instanciation) Pour toute substitution* σ*,* $t_1 < t_2 \Rightarrow \sigma(t_1) < \sigma(t_2)$

Définition 4.19 Système de réécriture bien ordonné

Un système de réécriture conditionnelle $R = R_1 \cup \ldots \cup R_n$ *est bien ordonné si est seulement si il existe un ordre de réduction* $<$ *sur* $T(F, X)$ *compatible avec le partitionnement tel que pour toute règle r de R*

- $r = g \rightarrow d$, *alors* $d < g$
- $r = \wedge_{i \in [1..n]} p_i == p'_i \Rightarrow g \rightarrow d$, *alors* $p_i < g$ *pour tout* $i \in [1..n]$ *et* $d < g$
- $r = \wedge_{i \in [1..n]} p_i == p'_i \Rightarrow g \rightarrow (d_1, d_2)$, *alors* $p_i < g$ *pour tout* $i \in [1..n]$, $d_1 < g$ *et* $d_2 < g$

Il existe plusieurs méthodes syntaxiques pour tester si deux termes sont comparables avec un ordre de réduction donné. Il existe plusieurs catégories d'ordres, toutes sont des extensions d'un ordre sur les symboles de fonctions, appelé *précédence*. Parmis ces catégories nous pouvons citer les ordres de type *SPO* (Subterm Path Ordering) [Pla78], les ordres qui utilisent l'ordre récursif sur les chemins *RPO* (Recursive Path ordering) [Der82], les ordres qui utilisent l'ordre récursif de décomposition *RDO* (Recursive Decomposition Ordering) [JLR82], [Rus85].
Nous pouvons donc adopter facilement une de ces méthodes pour assurer la calculabilité des formes normales.

Théorème 4.1

Soit R un système de réécriture bien ordonné, alors la forme normale de chaque terme est calculable.

Ce théorème peut être prouvé si nous donnons la procédure du calcul de la forme normale, puis nous vérifions que cette procédure termine et qu'elle calcule bien la forme normale.
Soit $R = R_1 \cup \ldots \cup R_p \cup Th_r$ le système de réécriture qui correspond à une spécification donnée. Th_r est un ensemble de règles de réécriture qui sera utilisé avec chaque R_i. Il contient les règles qui sont susceptibles d'être utilisées à n'importe quelle étape de la réduction. *E_terme* est l'ensemble des termes à réduire. *Rc* est le système de réécriture courant, il contient l'ensemble des théorèmes plus la partie du système de réécriture concernant cette étape.

```
1 : Fonction FN(t, R1 ∪ ... ∪ Rp)
2 : E_terme ← {t}
3 : résultat ← {}
4 : Pour i de p à 1 faire
5 :     tant_que non vide de E_terme faire
6 :         Rc ← Th_r ∪ Ri
7 :         trouve ← faux
8 :         tant_que non trouve faire
9 :             r ← premier(Rc)
10 :            Rc ← décapiter(Rc)
11 :            trouve ← (∃ u occurrence de t et ∃ σ tel que t/u = σ(g))
                % g est le membre gauche de la règle r
12 :        fin_tant_que
13 :        si trouve alors
14 :          cas r =
15 :                * g → d alors E_terme ← E_terme − {t} ∪ {t[u ← σ(d)]}
16 :                * ∧i∈[1..n] pi == p'i ⇒ g → d
17 :                Soient p''i = FN(σ(pi), Ri ∪ ... ∪ Rp) pour tout i ∈ [1..n]
18 :                si pour tout i de [1..n] p''i = p'i
19 :                  E_terme ← E_terme − {t} ∪ {t[u ← σ(d)]}
```



```
20 :                * ∧i∈[1..n] pi == p'i ⇒ g → (d1, d2)
21 :                Soient p''i = FN(σ(pi), Ri ∪ ... ∪ Rp) pour tout i ∈ [1..n]
22 :                si pour tout i de [1..n] p''i = p'i alors
23 :                  E_terme ← E_terme − {t} ∪ {t[u ← σ(d1)]}
24 :                  sinon
25 :                  si il existe un i de [1..n] tel que p''i = ¬p'i
26 :                    alors E_terme ← E_terme − {t} ∪ {t[u ← σ(d2)]}
27 :                  fin_si
28 :                fin_si
29 :          fin_cas
30 :          sinon résultat ← résultat ∪ {t}
31 :      fin_si
32 :  fin_tant_que
33 :  E_terme ← résultat
34 :  résultat ← {}
35 : fin_pour
36 : FN ← E_terme
37: fin
```

Proposition 4.9

Si R est un système de réécriture bien ordonné alors

1. *FN(t) termine*
2. *FN(t) calcule bien la forme normale de t*

Preuve

1. Soit $<$ l'ordre de réduction utilisé pour prouver que R est bien ordonné.
   Si nous prouvons que chaque étape de calcule de la forme normale, pour chaque R_i termine alors alors l'algorithme global termine.
   Supposons maintenant que nous nous situons à une étape i de l'algorithme.
   Si $<$ est bien fondé, alors pour chaque terme il existe un nombre fini de termes $t_1, \ldots, t_n$ tel que
   $$(*) \quad t_1 < t_2 < \ldots < t_n = t$$
   A chaque terme t nous associons un entier n défini par n = poids(t), qui est la longueur de la plus grande suite de termes $(*)$.
   FN(t) peut ne pas terminer dans deux cas:
   - E_terme n'est jamais vide
   - FN est appelé récursivement.

   Pour monter que FN termine il faut montrer que ces deux cas sont impossibles.
   Dans le premier cas ce sont seulement les lignes 15, 19, 23, 26 qui peuvent ajouter un terme de la forme $t[u \leftarrow \sigma(d)]$ à *E_terme*. Puisque $d < g$ et $t[u \leftarrow \sigma(d)] < t$ (propriétés de compatibilité et de stabilité par substitution de $<$), poids($t[u \leftarrow \sigma(d)]$) < poids(t). Or nous ne pouvons ajouter à *E_terme* que poids(t) termes. Donc un nombre fini. Dans les lignes 15, 19, 23, 26 nous enlevons à chaque passage un élément de *E_terme*, qui doit être vide au bout d'un nombre fini de passages.

dans le deuxième cas $p < g$, donc $poids(p) < poids(g) \leq poids(\sigma(g)) < poids(t_{/u}) \leq poids(t[u \leftarrow \sigma(d)])$. "Poids" est donc strictement décroissant dans chaque appel récursif. Il ne peut donc exister une infinité d'appels récursifs.

2. les ligne 7...33 sont une implantation de la définition de la réduction, ce qui implique que la correction de la procédure est immédiate.

5.3 Algorithme de calcul de l'ensemble de formes normales d'un a_terme

Soit $R = R_1 \cup \ldots \cup R_p$ le système de réécriture correspondant à une spécification donnée. Soit Thr l'ensemble des règles de réécriture susceptibles d'être utilisées dans n'importe quelle étape de réécriture.
Soit $\langle \{a_1, \ldots, a_n\}. t \rangle$ un a_terme, avec t un terme quelconque.
Soient *Ea_terme* l'ensemble des a_termes à réduire et Rc le système de réécriture courant. Il contient pour chaque a_terme l'ensemble des assertions de celui ci, l'ensemble Thr plus la partie du système de réécriture concernant cette étape.

1 : **Fonction** EFNA($\{\langle \{a_1, \ldots, a_n\}, t\rangle\}, R_1 \cup \ldots \cup R_p$)
2 : *résultat* $\leftarrow \{\}$
3 : *Ea_terme* $\leftarrow \{\langle \{a_1, \ldots, a_n\}, t\rangle\}$
4 : Pour i de p à 1 faire
5 : tant_que non vide de *Ea_terme* faire
6 : $Rc \leftarrow Orienter(\{a_1, \ldots, a_n\}) \cup Thr \cup R_i$
7 : *trouve* $\leftarrow$ *faux*
8 : tant_que non *trouve* faire
9 : $r \leftarrow$ premier(Rc)
10 : $Rc \leftarrow$ *décapiter*(Rc)
11 : *trouve* $\leftarrow (\exists\ u$ *occurrence de* t *et* $\exists \sigma$ *tel que* $t_{/u} = \sigma(g))$
% g est le membre gauche de la règle choisie
12 : fin_tant_que
13 : si *trouve* alors
14 : cas r =
15 : $* \ g \rightarrow d$ alors *Ea_terme* $\leftarrow$ *Ea_terme* $\cup \{\langle \{a_1, \ldots, a_n\}, t[u \leftarrow \sigma(d)]\rangle\}$
16 : $* \ \wedge_{i \in [1..n]} p_i == p'_i \Rightarrow g \rightarrow d$
17 : soient $p''_i = FN(\sigma(p_i), R_i \cup \ldots \cup R_p)$ pour tout i de [1..n]
18 : si pour tout i de [1..n] $(p''_i = p'_i)$ alors
19 : *Ea_terme* $\leftarrow$ *Ea_terme* $\cup \{\langle \{a_1, \ldots, a_n\}, t[u \leftarrow \sigma(d)]\rangle\}$
20 : $* \ \wedge_{i \in [1..n]} p_i == p'_i \Rightarrow g \rightarrow (d_1, d_2)$
21 : soient $p''_i = FN(\sigma(p_i), R_i \cup \ldots \cup R_p)$ pour tout i de [1..n]
22 : si pour tout i de [1..n] $(p''_i = p'_i)$alors
23 : *Ea_terme* $\leftarrow$ *Ea_terme* $\cup \{\langle \{a_1, \ldots, a_n\}, t[u \leftarrow \sigma(d_1)]\rangle\}$
24 : sinon
25 : si il existe un i de [1..n] $(p''_i = \neg p'_i)$
26 : alors *Ea_terme* $\leftarrow$ *Ea_terme* $\cup \{\langle \{a_1, \ldots, a_n\}, t[u \leftarrow \sigma(d_2)]\rangle\}$
27 : sinon
28 : *Ea_terme* $\leftarrow$ *Ea_terme* $\cup$
$\{ \langle \{a_1, \ldots, a_n\} \cup \{(\sigma(p_i) == p'_i)_{i \in [1..n]}, (p''_i == p'_i)_{i \in [1..n]}\}, t[u \leftarrow \sigma(d_1)]\rangle,$
$\langle \{a_1, \ldots, a_n\} \cup \{\sigma(p_1) == \neg\sigma(p'_1), p''_1 == \neg p'_1\}, t[u \leftarrow \sigma(d_2)]\rangle\}$

$\vdots$
$\langle \{a_1, \ldots, a_n\} \cup \{\sigma(p_n) == \neg p'_n, p''_n == \neg p'_n\}, t[u \leftarrow \sigma(d_2)]\rangle\}$
29 : fin_cas
30 : *Ea_terme* $\leftarrow$ *Ea_terme* $- \{\langle \{a_1, \ldots, a_n\}, t\rangle\}$
31 : sinon *résultat* $\leftarrow$ *résultat* $\cup \{\langle \{a_1, \ldots, a_n\}, t\rangle\}$
32 : fin_si
33 : fin_tant_que
34 : *Ea_terme* $\leftarrow$ *résultat*
35 : *résultat* $\leftarrow \{\}$
36 :fin_pour
37 :$EFN \leftarrow$ *résultat*
38 :fin

Remarque 4.4

- *Sur la ligne 6, les assertions sont utilisées sans distinction, comme des règles de réécriture supplémentaires, pour la réduction du terme qui les accompagne.*
- *La fonction orienter prend un ensemble d'équations et rend un ensemble de règles de réécriture, obtenue par orientation de chaque équation de gauche à droite.*

Proposition 4.10
Si R est un système de réécriture bien ordonné alors

1. $EFNA(\{\langle \{a_1, \ldots, a_n\}, t\rangle\})$ *termine et*
2. $EFNA(\{\langle \{a_1, \ldots, a_n\}, t\rangle\})$ *calcule bien l'ensemble des formes normales de* $\langle \{a_1, \ldots, a_n\}, t\rangle$.

La preuve de cette proposition ressemble à celle de la proposition 4.9, car la correction de $EFNA$ découle de la correction de FN.

6 Algorithmes pour la décision de validité d'une a_équation

6.1 Calcul de l'ensemble des dérivées d'une a_équation

Soit $\langle \{a_1, \ldots, a_n\}, e\rangle$ un a_équation quelconque

0 : **Fonction** Reduct_assert($\langle \{a_1, \ldots, a_n\}, e\rangle, R$)
1 : Résultat_final $\leftarrow \{\}$
2 : Pour i de 1 à n faire
3 : $a_i = (t_1 == t_2)$
% Une assertion est toujours de la forme $t_1 == t_2$. t_2 *peut être égale à vrai*
% ou à faux ou un autre terme de même sorte que t_1

4 : Soit $A_1 = EFNA(\langle \{a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n\}, t_1\rangle, R)$
% Nous calculons l'ensemble des formes normales de t_1 *sans l'assertion* a_i

```
5 :     soit A_2 = EFNA(({a_1,...,a_{i-1},a_{i+1},...,a_n},t_2),R)
        % Nous calculons l'ensemble des formes normales de t_2 sans l'assertion a_i

6 :     A = composer(=, A_1, A_2)
7 :     Résultat ← {}
8 :     pour chaque a_terme ⟨{b_1,...,b_m}, s_1 = s_2⟩ de A faire
        % {b_1,...,b_m} contient les anciennes assertions plus éventuellement des nouvelles
        % ajoutées au moment de la réduction de t_1 ou de t_2

9 :         cas
10 :            * (s_1 = s_2) alors
11 :                    Résultat ← Résultat ∪ {({a_1,...,a_n} ⊎ {b_1,...,b_m}, t)}
                % Si le résultat de la réduction de l'assertion est équivalent à vrai
                % alors nous ajoutons à l'ensemble des assertions initiales
                % les éventuelles assertions crées pendant la réduction

12 :            * (s_1 == s_2)  = vrai == faux ou faux == vrai alors
13 :                    Résultat ← Résultat ∪ {({a_1,...,a_n} ⊎ {b_1,...,b_m}, trivial)}
                % Si se résultat est équivalent à faux, c'est que l'assertion était invalide,
                % le a_terme devient donc un a_terme trivial

14 :            * sinon Résultat ← Résultat ∪ {({a_1,...,a_n} ⊎ {b_1,...,b_m} ⊎ {s_1 == s_2}, e)}
                % Sinon nous ajoutons la nouvelle assertion aux anciennes après
                % avoir tester si elle n'introduit pas d'inconsistance

15 :        fin_cas
16 :    fin_pour
17 : Résultat_final ← union_listes(Résultat_final, Résultat)
    % Le résultat final contient la réunion des assertions résultantes de la réduction
    % de chaque assertion, quand celle-ci n' à pas donné d'inconsistance
18 : fin_pour
19 : Reduct_assert ← Résultat_final


1 : Fonction Union_listes(liste_1, liste_2)
2 : si vide(liste_1) alors {}
3 :   sinon
4 :       ({a_1,...,a_n}, e) ← début(liste_1)
5 :       si e = trivial alors
6 :           {({a_1,...,a_n}, trivial)} ∪ union_listes(reste(liste_1), liste_2)
7 :                 sinon
8 :           union_terme(l_1, liste_2) ∪ union_listes(reste(liste_1), liste_2)
9 :       fin_si
10 : fin_si
11 : fin


1 : Fonction Union_terme(({a_1,...,a_n}, e), liste)
```



```
2 : si vide(liste) alors {}
3 :   sinon
4 :       ({b_1,...,b_m}, e_1) ← début(liste)
5 :       si e_1 = trivial alors
6 :       {({a_1,...,a_n}, trivial)} ∪ union_terme(({a_1,...,a_n}, e), reste(liste))
7 :                 sinon
8 :         {({a_1,...,a_n} ⊎ {b_1,...,b_m}, e)} ∪ union_terme(({a_1,...,a_n}, e), reste(liste))
9 :       fin_si
10 : fin_si
11 : fin
```

6.2 Algorithme de décision de validité d'une a_équation

Soit $\langle\{a_1,\ldots,a_n\},e\rangle$ un a_équation quelconque

```
1 : Fonction Reduct_theo(({a_1,...,a_n}, e), R)
2 :  E_candidat ← reduct_assert(({a_1,...,a_n}, e), R)
3 :  pour chaque a_équation ⟨{b_1,...,b_m}, e_1⟩ de E_candidat faire
4 :  cas
5 :      * e_1 = trivial :
6 :              imprimer("a_équation valide par suite d'hypothèses inconsistantes")
9 :      * e_1 = t_1 == t_2,
        % Nous cherchons l'ensemble de formes normales
        % de chaque a_terme puis nous composeons les deux résultats

10 :             A_1 = EFNA(({b_1,...,b_m}, t1), R)
11 :             A_2 = EFNA(({b_1,...,b_m}, t2), R)
12 :             A = Composer (==, A_1, A_2)
13 :             Pour chaque a_terme ⟨{c_1,...,c_l}, r_1 == r_2⟩ de A
13 :                 cas
14 :                     * r_1 = r_2 alors imprimer("a_équation valide ")
15 :                     * (r_1 == r_2)  =  (vrai == faux) ou faux == vrai alors
16 :                     imprimer("a_équation non valide")
17 :                     imprimer(({c_1,...,c_l}, (r_1 == r_2))
18 :                 fin_cas
19 :             fin_pour
20 :     fin_cas
21 : fin_pour
30 : fin
```

7 conclusion

La conclusion de ce chapitre portera sur deux axes : Le premier consiste à rappeler les conséquences de la méthode que nous avons développée sur la validité des formules équationnelles. Le second concerne les limites de cette méthode et les remèdes à ces limites.

Dans ce chapitre nous avons donné une méthode informatique pour prouver la validité équationnelle d'une a_équation. Or, dans le chapitre sur la décomposition nous avons vu que certaines formules équationnelles pouvaient être décomposées en un ensemble de a_équations et que la validité de ces formules était équivalente à celle des a_équations correspondantes. Par conséquence la validité d'une formule consiste à regrouper les solutions obtenues pour chacune des a_équations correspondantes.

La première limite concerne les propriétés qui ne peuvent pas être exprimées sous forme de règles de réécriture, et par conséquence ne peuvent pas être utilisées pour simplifier les termes. Dans le chapitre suivant nous allons étudier l'une de ces propriétés les plus utilisées dans les spécifications équationnelles. Il s'agit de la propriété de transitivité des relations d'ordre.

La deuxième limite concerne la complétude de la méthode de réécriture. En effet, nous avons vu dans la proposition 4.8 que si une a_équation est un théorème équationnel alors elle équationnellement valide. Nous aurions aimé que la réciproque soit aussi vraie. Cela est possible lorsque, en plus de la terminaison que nous avons montré, la confluence du système de réécriture associé à l'ensemble des équations initial augmenté des assertions, est garantie. Or, dans nos objectifs, nous n'avons pas voulu imposer de propriétés globales sur les spécifications initiales. Cela a des conséquences sur la réduction des termes et sur le calcul des formes normales. En effet, ne pas exiger ou ne pas démontrer la confluence d'un système de réécriture associé à une spécification donnée a pour conséquence de ne pas garantir l'unicité de la forme normale d'un terme. Autrement dit, l'ordre d'utilisation des règles de réécriture aura une influence sur le résultat obtenu à la fin. Cependant, la réduction des assertions permet de combler ce vide et permet de détecter les éventuelles inconsistances ou d'augmenter l'ensemble des équations initial avec des nouvelles assertions. Ainsi, le fait de ne pas pouvoir prouver qu'une a_équation est un théorème équationnel ne veut pas dire qu'elle n'est pas équationnellement valide. Mais peut être en complétant le système de réécriture avec une nouvelle règle ou bien en modifiant l'ordre des règles initiales, le résultat pourrait changer. Cette possibilité de complétion ou de modification dépend de la convivialité du système de preuve qui implémente les algorithmes présentés dans les paragraphes précédents. Le chapitre consacré au système *PAUSE* aura comme objectif de présenter les qualités de ce système qui permettront de réaliser ces modifications.

Chapitre 5

Chaînage transitif

1 Introduction

Nous avons vu dans le chapitre précédent que les techniques de réécriture permettaient de calculer les formes normales des termes ou des a_termes, à partir d'un ensemble de règles de réécriture. Ces formes normales sont ensuite comparées afin de décider la validité d'une équation ou d'une a_équation. Nous avons vu aussi que ces règles de réécriture étaient des équations qui définissaient des propriétés relatives aux fonctions de la spécification. Or il existe certaines propriétés que nous ne pouvons pas exprimer sous forme d'équations. Et même si nous arrivions à le faire, ces propriétés perdent toutes leurs puissances. C'est le cas de la propriété de transitivité des relations d'ordre: $x \leq y$ *et* $y \leq z \Rightarrow x \leq z$. L'impossibilité d'exprimer ces propriétés sous forme d'équations, empêche de prouver beaucoup d'autres propriétés relatives aux relations d'ordre, en utilisant la réécriture. Ainsi, ce que nous proposons ici, c'est d'utiliser cette propriété de transitivité comme une base d'une méthode de déduction, et non pas seulement comme une règle de réécriture. Cette méthode de déduction sera utilisée dans le cas où la réécriture n'a pas permis de décider la validité d'une équation. Elle sera appliquée sur des équations qui sont soit la conclusion ou une assertion d'une a_équation, soit une équation parmi les conditions d'une règle conditionnelle. En résumé, ce mode de raisonnement est utilisé en présence d'équations qui contiennent des relations d'ordre. Il est basé sur la réfutation (déduction de faux à partir d'un ensemble d'équations contradictoires) et sur la transitivité des relations d'ordre.

2 Exemple

L'exemple suivant a pour objectif de dégager les principes généraux de notre approche.

Exemple 5.1
Nous voulons prouver la validité de l'équation $max(A) \leq max(B) + p == vrai$ *en utilisant les équations suivantes :*
$max(A) < max(C) + 1 == vrai$
$max(C) \leq max(B) == vrai$
$x + y < x + z == y < z$
et $p < 1 == faux$

C'est impossible en utilisant seulement les techniques de la réécriture. Par contre, si nous considérons la négation de $max(A) \leq max(B) + p$, *qui est* $max(B) + p < max(A)$ *et si nous appliquons dessus la relation de transitivité de* $<$ *avec* $max(A) < max(C) + 1$, *nous obtenons* $max(B) + p < max(C) + 1$. *Si nous appliquons la transitivité sur cette formule résultat et la formule* $max(C) \leq max(B)$, *nous obtenons* $max(B) + p < max(B) + 1$, *en remplaçant* $max(C)$ *par* $max(B)$. *Avec une simplification de cette dernière formule en utilisant la règle* $x + y < x + z \rightarrow y < z$, *nous obtenons* $p < 1$. *Une autre simplification avec la règle* $p < 1 == faux$ *nous donne* $faux$. *Nous pouvons conclure, alors que* $max(A) \leq max(B) + p == vrai$ *est une équation valide puisque nous avons commencé notre raisonnement avec sa négation.*

Dans cet exemple, nous pouvons remarquer qu'il y a deux types de raisonnement qui caractérisent ce mécanisme de déduction, ce sont *le chaînage transitif* (utilisation de la transitivité pour remplacer un sous terme par un autre terme plus petit), puis *la simplification* (utilisation des règles de réécriture pour simplifier le résultat du chaînage transitif précédent). La suite de ce chapitre sera consacrée à la définition de ces deux types de raisonnement puis a' la stratégie utilisée pour les appliquer.

3 Chaînage transitif

Intuitivement le chaînage transitif consiste à remplacer dans la partie droite d'un terme contenant une relation d'ordre $<$ ou $\leq$, un sous terme par un terme plus petit. C'est l'équivalent du remplacement d'égaux par égaux dans le cas du raisonnement équationnel.

Définition 5.1 Chaînage transitif simple

Soient $t_1 \mathbb{Q}_1 s_1 == vrai$, $t_2 \mathbb{Q}_2 s_2 == vrai$ *et* $t_3 \mathbb{Q}_3 s_3 == vrai$ *trois équations telles que* $(\mathbb{Q}_1, \mathbb{Q}_2, \mathbb{Q}_3) \in \{<, \leq, \}^3$.
$t_3 \mathbb{Q}_3 s_3$ *est le résultat du chaînage transitif simple entre* $t_1 \mathbb{Q}_1 s_1$ *et* $t_2 \mathbb{Q}_2 s_2$ *si et seulement si il existe une occurrence* u *de* s_1 *et un unificateur principal* σ *tel que* $\sigma(s_{1/u}) = \sigma(t_2)$, *dans ce cas*

- $t_3 = \sigma(t_1)$,
- $s_3 = \sigma(s_1)[(v_i)_{i \in I} \leftarrow \sigma(s_2)]$, *avec* $(v_i)_{i \in I}$ *la famille des occurrences de* s_1 *telles que* $s_{1/v_i} = s_{1/u}$,
- $\mathbb{Q}_3 = tr(\mathbb{Q}_1, \mathbb{Q}_2)$ *avec* $tr(\mathbb{Q}_1, \mathbb{Q}_2) = si\ \mathbb{Q}_1 = <\ ou\ \mathbb{Q}_2 = <\ alors <\ sinon \leq$.

$t_2 \mathbb{Q}_2 s_2 == vrai$ *est appelée la résolvante.*

Exemple 5.2 Chaînage transitif simple

Soit l'équation $x + y < (z + (a * a)) * y == vrai$, *sur laquelle nous allons appliquer le chaînage transitif en utilisant l'équation* $(x * x) + y < x + y == vrai$. *L'unificateur principal est* $\sigma(z) = x * x$ *et* $\sigma(y) = a * a$. *Nous obtenons alors* $(x * x) + (a * a) < ((x * x) + (a * a)) * (a * a) == vrai$

La résolvante peut être une équation conditionnelle, dans ce cas, avant d'appliquer le chaînage transitif, il faut tester si la condition est vérifiée.

Définition 5.2 Chaînage transitif conditionnel

Soient R *un système de réécriture,* $t_1 \mathbb{Q}_1 s_1 == vrai$, $t_3 \mathbb{Q}_3 s_3 == vrai$, *deux équations simples et* $\wedge_{i \in [1..n]} p_i == p'_i \Rightarrow (t_2 \mathbb{Q}_2 s_2 == vrai)$ *une équation conditionnelle, avec* $(\mathbb{Q}_1, \mathbb{Q}_2, \mathbb{Q}_3) \in \{<$

$, \leq, \}^3$
$t_3 \mathbb{Q}_3 s_3$ *est le résultat du chaînage transitif conditionnel entre* $t_1 \mathbb{Q}_1 s_1$ *et* $t_2 \mathbb{Q}_2 s_2$ *si et seulement si il existe une occurrence* u *de* s_1 *et un unificateur principal* σ *tel que* $\sigma(s_{1/u}) = \sigma(t_2)$ *et pour tout* $i \in [1..n], \sigma(p_i) \downarrow_R = p'_i$, *dans ce cas*

- $t_3 = \sigma(t_1)$,
- $s_3 = \sigma(s_1)[(v_i)_{i \in I} \leftarrow \sigma(s_2)]$, *avec* $(v_i)_{i \in I}$ *la famille des occurrences de* s_1 *telles que* $s_{1/v_i} = s_{1/u}$,
- $\mathbb{Q}_3 = tr(\mathbb{Q}_1, \mathbb{Q}_2)$ *avec* $tr(\mathbb{Q}_1, \mathbb{Q}_2) = si\ \mathbb{Q}_1 = <\ ou\ \mathbb{Q}_2 = <\ alors <\ sinon \leq$.

Exemple 5.3 Chaînage transitif conditionnel

Soient l'équation $max(A) \leq max(B) + p == vrai$ *de l'exemple 5.1 avec les équations*

1. $vide(A) == faux \wedge vide(C) == faux \Rightarrow max(A) < max(C) + max(C) + 1 == vrai$
2. $max(C) \leq max(B) == vrai$
3. $x + y < x + z == y < z$
4. $p < 1 == faux$
5. $vide(A) == faux$
6. $vide(C) == faux$

Le chaînage transitif de $max(B) + p < max(A) == vrai$ *avec l'équation* 1 *donne, après la réduction de la condition, l'équation* $max(B) + p < max(C) + max(C) + 1 == vrai$. *Qui à son tour donne, avec la résolvante* $max(C) \leq max(B) == vrai$, *l'équation* $max(B) + p < max(B) + max(B) + 1 == vrai$

Remarque 5.1
Le chaînage transitif ne peut pas être appliqué à toutes les équations contenant des relations d'ordre. En effet, si nous remplaçons $max(C)$ *par* p *dans* $max(A) < max(B) - max(C)$, *en utilisant la résolvante* $max(C) < p == vrai$, *nous obtenons* $max(A) < max(B) - p$. *Cette inéquation n'est pas correcte par rapport à l'inéquation initiale. Cela vient du fait que la fonction "-" est décroissante. Donc, une condition nécessaire pour appliquer le chaînage transitif est que les fonctions englobant les occurrences à remplacer soient des fonctions croissantes.*
Dans le cas de la fonction "-", nous pouvons résoudre ce problème en amenant la partie droite de la fonction de l'autre côté de la relation d'ordre. Ainsi $max(A) < max(B) - max(C)$ *devient* $max(A) + max(C) < max(B)$.

4 Chaînage transitif avec simplification

Sur l'exemple 5.1, nous pouvons voir qu'il est impossible de déduire $faux$ en utilisant seulement le chaînage transitif. Cela vient du fait que le chaînage fait seulement le remplacement des sous termes, mais ne teste pas si les deux sous termes résultats sont en relation ou non. Il nous faut donc un autre mécanisme de raisonnement pour pouvoir réaliser ce test. Cela est possible si nous utilisons les règles de réécriture, relatives à la relation d'ordre utilisée, pour simplifier le terme résultat du chaînage. Cette simplification consiste à calculer la forme normale du terme

contenant le symbole de relation d'ordre. Ainsi si cette simplification donne $faux$ le chaînage est terminé, sinon nous aurons obtenu un terme plus compact et plus simple sur lequel sera appliquée la prochaine étape de chaînage.

Définition 5.3 Simplification

Soit R un système de réécriture contenant les règles qui définissent les relations d'ordre $\{<, \leq\}$. Une équation $t == vrai$ est le résultat de simplification de l'équation $t_1 @ t_2 == vrai$, avec $@ \in \{<, \leq\}$, si et seulement si $t = (t_1 @ t_2) \downarrow_R$.

Exemple 5.4

Le résultat final de l'exemple 5.3, $max(B) + p < max(B) + max(B) + 1$, pourra être simplifié en utilisant la règle correspondante à l'équation 3, $x + y < x + z == y < z$, pour donner l'équation $p < max(B) + 1 == vrai$.

5 Stratégie de chaînage

Dans l'exemple 5.1, nous avons commencé par calculer la négation de l'équation à prouver, puis nous lui avons appliqué une étape de chaînage transitif. Sur le résultat de cette étape, nous avons appliqué le même mécanisme, et ainsi de suite jusqu'à obtenir l'équation contradictoire $faux == vrai$ ou explorer toutes les possibilités de chaînage. Cette stratégie de raisonnement est une stratégie linéaire. Elle utilise toujours l'équation résultante de l'étape précédente comme équation d'entrée pour l'étape suivante. Cette stratégie a été initialement introduite par [Lov70] et [KK70] pour faire de la résolution linéaire dans le cadre de la logique de premier ordre. Nous allons suivre une stratégie similaire que nous appelons *Le chaînage linéaire.*

5.1 Chaînage linéaire avec simplification

Définition 5.4 Chaînage linéaire avec simplification

Soient S un ensemble d'équations de la forme $t @ s == vrai$ ou $\wedge_{i \in [1..n]}(p_i == p_i') \Rightarrow t @ s == vrai$, avec $@ \in \{<, \leq\}$, et E_0 une équation de S. Une équation E_n de S est un chaînage linéaire à partir de S avec E_0 comme équation de départ si et seulement si:
pour tout $i \in [0, n-1]$, E_{i+1} est le résultat de la simplification du chaînage transitif entre E_i (équation centrale) et B_i (équation latérale). Les B_i peuvent être des équations de S, comme elles peuvent être des équations centrales des étapes précédentes. La figure 5.1 est un chaînage linéaire de E_n.

5.2 Algorithme de chaînage linéaire avec simplification

Ce paragraphe sera consacré à l'implantation du chaînage linéaire avec simplification. En effet, nous pouvons espérer qu'il soit complet, mais encore faut il trouver un algorithme qui termine et qui soit efficace pour l'implanter.
Soit E_0 l'équation du début. Nous devons chercher toutes les équations latérales qui peuvent être des résolvantes pour E_0. Après le chaînage de E_0 avec ces équations latérales puis la simplification, nous obtenons des résultats $R_1, \ldots, R_n$. Chaque R_i peut être une équation centrale qui peut donner une preuve. S'il existe un i tel que $R_i = (faux == vrai)$ la preuve est terminée, sinon nous recommençons le même processus pour chaque R_i jusqu'à trouver l'équation $faux == vrai$.

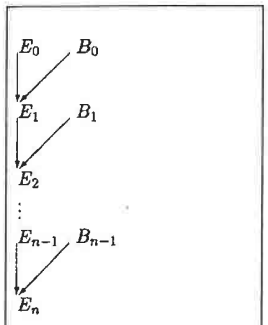


Figure 5.1: Chaînage linéaire

Il est clair que ce type de recherche d'un chaînage linéaire est un problème d'arbre de recherche classique que nous pouvons trouver par exemple dans [Sla71] et [Nil71]. La stratégie en largeur d'abord est une des stratégies les plus utilisées dans ce domaine. Elle consiste à faire une recherche de gauche à droite en consultant les frères d'un nœud avant de consulter ses fils. Cette stratégie permet toujours de trouver une solution quand elle existe. Cependant, elle peut générer beaucoup d'équations inutiles avant de trouver l'inconsistance. Une autre stratégie de recherche possible est la stratégie en profondeur d'abord. Il s'agit de faire une recherche de haut en bas en consultant les fils d'un nœud avant de consulter ses frères. Cette stratégie n'a pas l'inconvénient de générer beaucoup d'équations inutiles, mais elle risque de générer des boucles incontrôlables et de ce fait ne pas terminer. C'est la stratégie que nous avons choisi pour implanter le chaînage linéaire avec simplification. Et pour contrôler les boucles nous avons choisi de fixer le nombre de fois où une équation est utilisée comme équation latérale.

L'algorithme que nous allons présenter implante le chaînage linéaire. Il utilise une stratégie en profondeur d'abord avec retour arrière. Une résolvante est choisie lorsqu'elle satisfait les conditions de la définition 5.1 ou la définition 5.2 et lorsqu'elle n'a pas été utilisée plus de d fois. d est le nombre d'utilisation autorisées d'une équation comme résolvante. Le retour arrière se fait lorsqu'il n'y a plus de résolvante applicable sur l'équation courante; ou bien lorsque après simplification nous trouvons l'équation $vrai == vrai$. Dans ces deux cas l'algorithme revient à l'équation centrale précédente. L'algorithme s'arrête lorsqu'il a généré l'équation $faux == vrai$. Dans ce cas, la négation de l'équation initiale est valide. Il peut s'arrêter aussi lorsque aucune résolvante ne peut convenir pour effectuer un chaînage sur l'équation centrale courante. Dans ce cas, nous ne pouvons pas décider de la validité de l'équation initiale.

Cet algorithme prend en entrée une équation booléenne du type $t @ s == vrai$, avec $@ \in \{<, \leq\}$. Et Il utilise les ensembles suivants:

- *E_résolvante* contient les équations qui vont servir comme résolvantes (équations latérales),
- *R_ordre* est l'ensemble des règles qui vont être utilisées comme des règles de simplification, et
- R est le système de réécriture associé à la spécification dans laquelle nous voulons effectuer

la preuve.

```
:  Fonction chain_linéaire(e, E_résolvante)
:  résolu ← faux
:  pile ← e
:  tant_que non vide(pile) et non résolu faire
:      (e1 @e e2 == vrai) ← tête(pile)
:      pile ← dépiler(pile)
:      trouve ← faux
:      tant_que non trouve et non vide (E_résovate) faire
:          r ← premier(E_résolvante)
:          E_résolvante ← décapiter(E_résolvante)
:          trouve ← (∃ une occurrence u de e2
                     et ∃σ tel que σ(e2/u) = σ(r1))
                        et nombre d'utilisation de r est inférieur à d.
:      fin_tant_que
:      si trouve alors
:        cas r =
:          * r1 @r r2 == vrai alors
:              res1 ← σ(e1)
:              res2 ← σ(e2)[(vi)i∈I ← σ(r2)] avec vi tel que e2/vi = e2/u
:              @res ← tr(@e, @r)
:          * ∧i∈[1..n] pi == p'i ⇒ (r1 @r r2)  alors
:          si pour tout i [1..n], σ(pi) ↓R= p'i alors
:            res1 ← σ(e1)
:            res2 ← e2[(vi)i∈[1..n] ← σ(r2)] avec vi tel que e2/vi = e2/u
:            @res ← tr(@e, @r)
:          fin_si
:        fin_cas
:        res ← (res1 @res res2) ↓Rordre
:        si (res = vrai) alors
:          écrire("Retour arrière")
:          sinon si (res = faux)
:                   alors résolu ← vrai
:                   sinon pile ← empiler(pile, res == vrai)
:                   fin_si
:        fin_si
:      fin_si
:  sinon écrire("Retour arrière ")
:  fin_tant_que
:  Si résolu alors chain_linéaire ← (faux == vrai)
: fin
```

La fonction *tr* calcul le symbole de la relation d'ordre résultat


```
:  Fonction tr(@1, @2)
:  si (@1 = " < ") ou (@2 = " < ") alors tr ← " < "
:                                  sinon tr ← " ≤ "
:  fin_si :                        fin
```

6 Validité des équations contenant des relations d'ordre

Maintenant que nous avons donné l'algorithme du chaînage linéaire, il faut définir comment nous allons l'utiliser pour prouver la validité des équations booléennes contenant des relations d'ordre. Pour cela il faut définir la forme des équations qui vont servir comme équations latérales, ainsi que la forme de l'équation initiale.

6.1 Equations résolvantes

D'après les définitions 5.1 et 5.2, les équations centrales comme les équations latérales ont toutes une forme standard, $t@s == vrai$ ou $\wedge_{i\in[1..n]} p_i == p'_i \Rightarrow t@s == vrai$, avec $@ \in \{<, \leq\}$. Or les équations contenant des relations d'ordre n'ont pas toutes initialement une de ces formes. Elles sont généralement de la forme $t@s == p$ ou $\wedge_{i\in[1..n]} p_i == p'_i \Rightarrow t@s == p$, avec $@ \in \{<, \leq, >, \geq\}$ et $p = vrai$ ou $p = faux$. Il faut donc les ramener sous la forme standard correspondante. Cela est possible grâce aux règles classiques de définition des relations d'ordre, et qui sont les suivantes :

- $t > s == p \iff s < t == p$, avec $p = vrai$ ou $p = faux$;
- $t \geq s == p \iff s \geq t == p$, avec $p = vrai$ ou $p = faux$;
- $t < s == faux \iff s \leq t == vrai$;
- $t \leq s == faux \iff s < t == vrai$.

Nous avons vu aussi dans l'exemple 5.1 que le mécanisme de chaînage transitif est basé sur la réfutation. Il faut donc calculer la négation de l'équation initiale. Cela est possible grâce aux règles suivantes :

- $\neg(t < s == vrai) \iff s \leq t == vrai$;
- $\neg(t \leq s == vrai) \iff s < t == vrai$.

Remarque 5.2
Les quatre dernières transformations ne sont pas réalisables dans un ordre partiel. En effet, pour que ces transformation soient valides, il faut que tous les éléments manipulés soient comparables ($x \leq y$ ou $y > x$). Il faut, donc que les relations d'ordre utilisées soient des relations d'ordre total.

Ces quatre règles de transformation jouent le rôle de l'axiome de comparabilité comme le chaînage transitif qui joue le rôle de l'axiome de transitivité.

La question qui se pose maintenant, c'est comment déterminer les équations résolvantes parmi les équations d'une spécification ?. Les critères de choix sont ceux que nous avons vu au cours des définitions et des exemples précédents. Nous pouvons les résumer par :

- Une équation résolvante doit être une équation booléenne,
- son membre gauche doit contenir un des symboles de relations d'ordre $\{<, \leq, >, \geq\}$.

Ces équations seront choisies parmi les assertions qui correspondent à l'équation à prouver et parmi les parties théorème de la spécification.

6.2 Règles de simplification

Les règles de simplification correspondent aux équations qui définissent la relation d'ordre utilisée, ainsi que son environnement. Dans le cas des relations d'ordre sur les entiers, la spécification de l'annexe A est une spécification complète des entiers avec les relations d'ordre $<$ et $\leq$. Cet ensemble de règle de simplification peut contenir aussi les assertions et les théorèmes non booléens.

6.3 Test de validité

Nombreux sont les cas où nous voulons savoir si un terme booléen t est équivalent, modulo un ensemble d'équations E, à $vrai$ ou à $faux$. Cela pourrait être utilisé, par exemple, pour évaluer la condition d'une règle conditionnelle, ou bien pour calculer la valeur de vérité d'une proposition représentée par ce terme. Les mécanismes de réécriture permettent de calculer la forme normale de ce terme. Cette forme normale peut être égale à $vrai$ ou à $faux$, dans ce cas nous avons atteint notre objectif. Mais elle peut être égale à un autre terme irréductible. Lorsque ce terme irréductible est de la forme $t_1@t_2$ avec $@ \in \{<, \leq, >, \geq\}$, nous pourrons utiliser le chaînage transitif. La première étape consiste à mettre ce terme sous une forme équivalente $t'_1@'t'_2$ avec $@' \in \{<, \leq\}$. Puis à calculer sa négation $t''_1@''t''_2$ avec $@'' \in \{<, \leq\}$. Puis à appliquer l'algorithme du chaînage linéaire sur $t''_1@''t''_2 == vrai$. Si cet algorithme termine avec $faux == vrai$ comme résultat alors nous pouvons conclure que $t_1@t_2 == vrai$ est une conséquence logique de E. Sinon et pour voir si ce terme est équivalent à $faux$, on applique l'algorithme de chaînage linéaire sur l'équation $t'_1@'t'_2 == vrai$. S'il termine, nous pouvons conclure que $t_1@t_2 == faux$ est une conséquence logique de l'ensemble d'équations E.
L'algorithme suivant illustre ce que nous avons dit

fonction chaînage $(t_1@t_2 == vrai, E)$
: $E\_résolvante \leftarrow Choix\_résolvante(E)$
: $t'_1@'t'_2 \leftarrow \neg(t_1@t_2)$
: $res \leftarrow chain\_linéaire(t'_1@'t'_2 == vrai, E\_résolvante)$
: si res = (faux == vrai) alors écrire("$t_1@t_2 == vrai$ est valide")
: sinon
: $res \leftarrow chain\_linéaire(t_1@t_2 == vrai, E\_résolvante)$
: si $res = (faux == vrai)$ alors écrire("$t_1@t_2 == faux$ est valide")

Choix_resolvante est une fonction qui parcourt l'ensemble des équations associées à une spécification pour déterminer les équations résolvantes selon les critères définis ci-dessus.

Exemple 5.5
Cet exemple a pour objectif de montrer l'utilisation du chaînage transitif pour évaluer une condition d'une règle conditionnelle.

Soient A, B et C trois objets du type ensemble ordonné tels qu'il est défini dans l'exemple 4.5. Il s'agit de réduire le terme $inf(max(A)+1, min(B))$ *avec la règle conditionnelle* $x < y == vrai \Rightarrow inf(x,y) \rightarrow (x,y)$. *Pour cela il faut évaluer la condition* $max(A)+1 < min(B) == vrai$. *Nous supposons que la forme normale de cette condition est égale à elle même. Nous sommes, donc, obligé d'utiliser le chaînage transitif. Les équations résolvantes que nous allons utiliser sont :*
$max(A)+1 < p+min(B) == vrai$,
$estvide(X) == faux \Rightarrow min(X) \leq max(X) == vrai$,
$max(C) < max(A) == vrai$.
La règle de simplification en plus des règles associées à la spécification des entiers de l'annexe 1, est :
$max(B) \rightarrow max(C)$,
$0 < p \rightarrow vrai$
Nous commençons par calculer la négation de l'équation initiale. C'est : $min(B) \leq max(A)+1 == vrai$. *Le chaînage transitif entre cette équation et la première équation résolvante, donne* $min(B) < p + min(B) == vrai$. *La simplification de cette équation avec la règle* $x < y + x \rightarrow 0 < y$ *puis avec la règle* $0 < p \rightarrow vrai$ *donne* $vrai == vrai$. *Nous sommes donc obligé de recommencer le processus du chaînage transitif avec l'équation* $max(A)+1 < min(B) == vrai$. *Le chaînage transitif de cette équation avec la deuxième équation résolvante donne, en supposant que la condition est vérifié,* $max(A)+1 < max(B) == vrai$. *Après simplification avec la règle* $max(B) \rightarrow max(C)$, *nous obtenons* $max(A)+1 < max(C) == vrai$. *Le chaînage transitif entre cette dernière équation et la troisième équation résolvante donne* $max(A)+1 < max(A) == vrai$. *Elle sera simplifiée avec la règle* $x+y < x \rightarrow y < 0$, *pour donner* $1 < 0 == vrai$. *Qui à son tour, après simplification avec la règle* $x < 0 \rightarrow faux$, *donnera* $faux == vrai$.

Nous pouvons conclure, donc, que $max(A)+1 < max(B) == faux$ *est valide et nous pouvons réécrire* $inf(max(A)+1, max(B))$ *en* $max(B)$.

7 Comparaison avec les travaux similaires

Dans le domaine de la preuve automatique, l'impossibilité de prouver certains théorèmes est souvent dûe à l'impossibilité d'inclure certains axiomes d'une théorie donnée dans un mécanisme de raisonnement ou à l'inefficacité d'un mécanisme en présence de certains axiomes. Pour contourner ce problème J. R. Slagle [Sla72] avait présenté l'idée de remplacer ces axiomes, dans le cas de quelque théorie spéciale et dans le cadre des techniques de résolution appliquées aux formules du premier ordre, par des règles de déduction qui soient équivalentes, efficaces et complètes. Ainsi la paramodulation remplace les axiomes d'égalité d'un ensemble de clauses. Les autres théories présentées par l'auteur sont la théorie des ordres partiels et la théorie des ensembles. Cette idée à été reprise par J. R. Slagle et L. Norton [SN73b] pour réaliser un système de preuve automatique pour les formules du premier ordre contenant des relations d'ordre partiel. Ils utilisaient les techniques de résolution et de paramodulation plus les règles de déduction qui simulent les axiomes d'ordre partiel. Ils l'ont ensuite amélioré, dans [SN73a], pour prendre en compte les relations d'ordre total, en ajoutant les règles de déduction qui simulent cet ordre. La même idée à été utilisée par W. Bledsoe, K. Kunen et R. Shotak [BKS85], pour exprimer les relations d'ordre sur les réels qui sont des relations d'ordre total, dense et sans point de fin. Son but était de prouver des propriétés sur le calcul élémentaire dans les réels, exprimées avec des formules du premier ordre.

Cependant, toute relation d'ordre est une relation irreflexive, antisymétrique et transitive.

Donc parmi les règles de déduction de ces différents articles se trouvait une règle qui simulait cette propriété. C'est la règle de chaînage transitif. La différence fondamentale entre notre chaînage et le leur, c'est que dans le cas des fonctions croissantes, nous pouvons remplacer des sous termes du membre droit de la relations d'ordre, alors que eux remplacent le terme tout entier.

Une autre différence se manifeste dans le fait que ces auteurs transforment toutes les propriétés des relations d'ordre qu'ils manipulent (irreflexivité, transitivité, comparabilité, ...), alors que nous n'avons transformé que la propriété de transitivité que nous ne pouvons pas exprimer sous forme de règle de réécriture. L'irreflexivité de $<$ ou la reflexivité de $\leq$ peuvent être déduites à partir du chaînage transitif d'une équation avec elle même, puis la simplification du résultat avec une des règles $x < x \rightarrow faux$ ou $x \leq x \rightarrow vrai$.

Si nous comparons le cadre d'utilisation, nous pouvons constater que tous ces auteurs expriment les propriétés à prouver avec des formules de la logique du premier ordre, alors que nous nous restreignons aux formules de la logique équationnelle. Notre utilisation des formules de la logique équationnelle n'est pas explicite. En effet le passage par l'étape de décomposition, nous permet de ne manipuler que des équations simples. Et de ce fait de n'appliquer le chaînage transitif que sur des équations simples. Cela permet de réduire la complexité des formules utilisées, comme nous l'avons signalé dans le chapitre 2. Le fait de se placer dans le cadre de la logique équationnelle a un autre avantage, c'est la possibilité d'utiliser l'induction pour prouver certaines propriétés contenant des relations d'ordre. Cela n'est pas possible avec les techniques de résolution appliquées sur des formules de la logique du premier ordre. L'utilisation des règles de simplification après le chaînage transitif est un autre avantage de la logique équationnelle.

Chapitre 6

Le système de preuve PAUSE

1 Description sommaire

PAUSE[1][Laz90b] est un système de preuve fondé sur les concepts définis dans les chapitres précédents à savoir la décomposition, l'induction structurelle, la réécriture, le raisonnement par cas et le chaînage transitif. Il est destiné à prouver des formules équationnelles dans le cadre des spécifications équationnelles en utilisant une série de transformations fondées sur les notions précédentes.

Le noyau de ce système est écrit en *Lelisp* tandis que la partie interface est écrite en *C*. Il est opérationnel sur *SUN3* et utilise les facilités de fenêtrage de cette machine.

1.1 Définition de l'environnement de la preuve

La première étape dans le processus de preuve est la définition de l'environnement dans lequel sera effectuée cette preuve. Pour cela il faut fournir à l'utilisateur un langage de spécification qui lui permettra de formaliser son problème sans rentrer dans les détails d'implantations. Il s'agit du langage des types abstraits algébriques [Fin79]. Ainsi l'utilisateur doit formaliser son problème avec un ensemble d'équations telles qu'elles sont définies dans le chapitre 1. Le chapitre suivant donnera un exemple concrêt du processus de définition de l'environnement de la preuve.

Pour des raisons de simplicité d'utilisation et de lisibilité, la forme externe des équations sera différente de celle vue dans les chapitres précédents. Cette différence apparaît dans le cas

- des équations booléennes $t == vrai$ qui peuvent être notées par le prédicat t.
- des équations conditionnelles $\wedge_{i\in[1..n]} p_i == p'_i \Rightarrow g == d$ qui seront notées par
$$g == \ si\ et_{i\in[1..n]}\ p_i == p'_i\ alors\ d\ fsi.$$
- des équations conditionnelles $\wedge_{i\in[1..n]} p_i == p'_i \Rightarrow g == (d_1, d_2)$ qui seront notées par
$$g == \ si\ et_{i\in[1..n]}\ p_i == p'_i\ alors\ d_1\ sinon\ d_2\ fsi.$$

Ces équations sont ensuite transformées sous forme de règles de réécriture dont les représentations internes respectent les définitions du chapitre 4.

Lorsqu'il s'agit d'une spécification composée de plusieurs types, l'utilisateur doit donner l'ordre d'utilisation des règles de ces différentes parties lors de la réduction (cf chapitre 4).

[1]Preuve AUtomatique dans les Spécifications Equationnelles

1.2 Définition de la propriété élémentaire

La deuxième étape dans le processus de preuve est la formalisation de la propriété élémentaire à prouver sous forme d'une formule équationnelle.

1.3 Déroulement de la preuve

Une fois l'environnement de la preuve défini et la propriété élémentaire à prouver formalisée, il ne reste plus qu'à lancer la preuve. Deux cas sont alors à distinguer :

- Une preuve équationnelle suffit pour tester la validité de cette propriété.
- Une preuve par induction est nécessaire; dans ce cas ou bien le schéma d'induction pourrait être généré automatiquement à partir des constructeurs et des variables d'induction; ou bien c'est l'utilisateur qui doit le fournir. Une nouvelle formule équationnelle est alors générée en substituant dans le schéma d'induction la formule initiale.

L'étape suivante pour les deux cas est l'étape de décomposition : il s'agit de transformer la formule en un ensemble de a_équations.
Pour chaque a_équation résultante de cette étape, le système va essayer de réduire les assertions pour détecter les inconsistances et calculer les formes normales. Ces assertions avec leurs formes normales seront ensuite utilisées à côté du système de réécriture initial pour réduire la partie conclusion.
Après cette étape, si la partie conclusion n'est ni prouvée ni inconsistante et dans le cas où elle est de la forme $a@b == p$ avec $@ \in \{<, \leq, >, \geq\}$ et $p \in \{vrai, faux\}$, le système essaiera le chaînage transitif.
Si après tout, la preuve a abouti à un problème ouvert, l'utilisateur, après visualisation des traces de la preuve, pourra s'il le juge nécessaire ajouter des lemmes pour corriger sa formalisation du problème puis relancer la preuve ou bien refaire une autre induction sur d'autres variables.

Remarque 6.1

Après cette présentation générale de la stratégie de preuve utilisée par notre système PAUSE, nous voudrions rappeler qu'elle est consistante car nous avons montré dans les chapitres 2, 3 et 4 que chaque composante l'est. Pour la complétude, autrement dit, est ce que tout théorème valide peut être prouvé par cette stratégie ? La réponse est :

- *Cette stratégie ne peut pas assurer que l'utilisateur donne le bon schéma d'induction pour assurer la validité inductive.*
- *Pour que la réécriture d'un a_terme $(\{a_1, \ldots, a_n\}, t)$ soit complète et permette ainsi l'existence d'un ensemble de formes normales unique, il faut que le système de réécriture associé à $E \cup \{a_1, \ldots, a_n\}$ soit canonique.*

1.4 Visualisation de la preuve

Il peut arriver plus souvent qu'on peut l'espérer qu'un système de preuve ne permette pas de prouver une propriété du premier coup. Cela peut venir d'une mauvaise formalisation du problème avec une mauvaise définition de certaines opérations, ou d'une mauvaise formalisation de la propriété à prouver. Seule dans ce cas une bonne visualisation des différentes étapes de la preuve pourrait aider l'utilisateur à reformuler sa propriété ou à corriger ou compléter sa

spécification en ajoutant les lemmes nécessaires.
Or, nous avons vu dans les chapitres précédents que la preuve d'une formule équationnelle est constituée d'une succession de a_formules obtenues les unes à partir des autres par des applications de certaines règles d'inférence. La visualisation de la preuve consiste donc à visualiser les a_formules des différentes étapes de la preuve et à visualiser comment le système à pu passer d'une a_formule à une autre. Nous avons choisi de représenter cette succession de a_formules par un arbre. Chaque nœud de cette arbre va représenter une a_formule. Chaque nœud contient en plus de la partie assertion et de la partie conclusion de la a_formule correspondante, un numéro hiérarchisé qui va indiquer sa profondeur dans l'arbre. Par exemple la racine est numérotée par 0. les nœuds qui y sont directement dérivés sont numérotés par $1, 2, \ldots$. Les nœuds dérivés d'un nœud numéro x sont numéroté par $x1, x2, \ldots$. La racine de cet arbre sera constituée du nœud numéro 0, d'une partie assertion vide et de la formule à prouver.
Pour ne pas trop encombrer cet arbre , seules les a_formules correspondantes aux étapes significatives de la preuve y seront représentées : la a_formule initiale, la a_formule obtenue par application d'un schéma d'induction, les a_équations obtenues par le processus de décomposition et les a_équations obtenues par un raisonnement par cas. Les autres étapes (réduction, chaînage transitif) seront visualisées avec les termes sur lesquels ils opèrent.
Pour visualiser la preuve, l'utilisateur a deux choix de trace :

- Le premier niveau de trace est un niveau de débogage. Il permet de visualiser toutes les étapes de la preuve, avec toutes les réductions des termes et des sous termes, tous les chaînages transitifs, avec tous les retours arrières possibles. Cela permet à l'utilisateur de comprendre sa preuve et dans le cas où elle échoue de pouvoir détecter pourquoi.
- Le deuxième niveau de trace permet de visualiser seulement les différents nœuds de l'arbre au fur et à mesure de leur construction

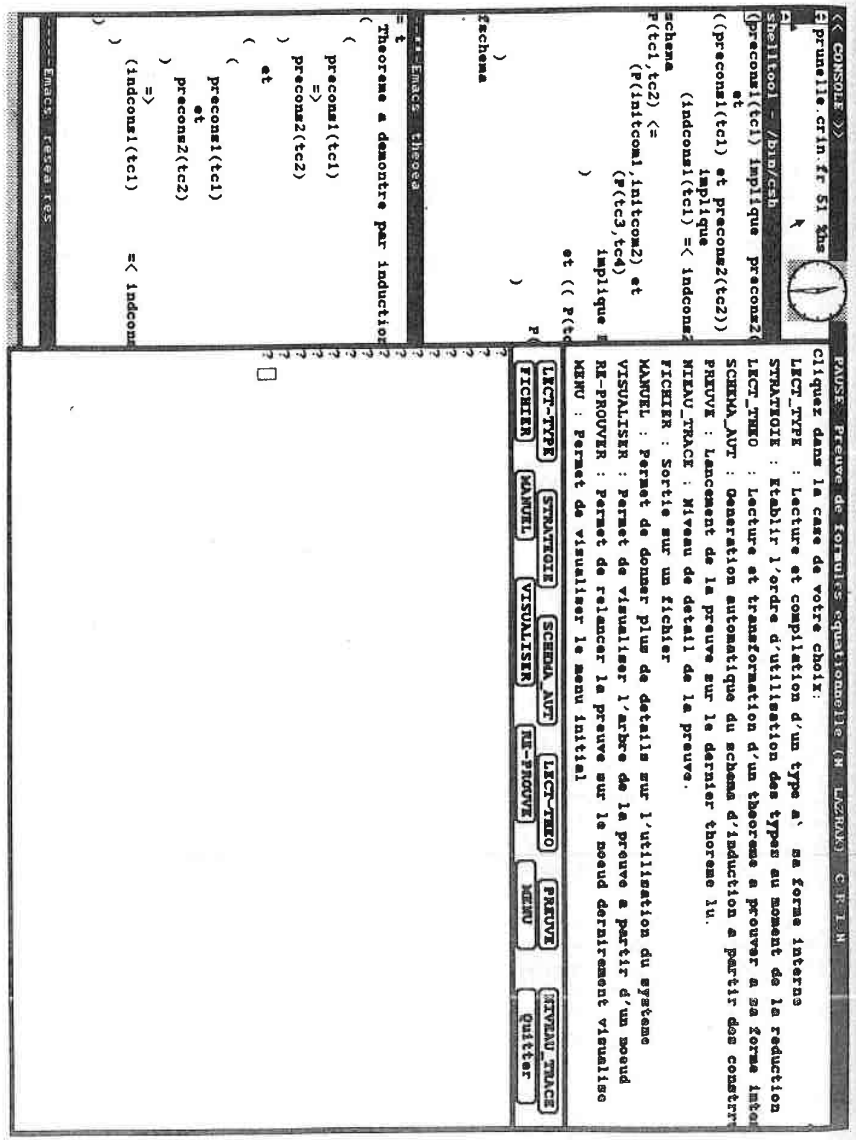
L'annexe C contient un exemple d'arbre de preuve.

2 Interface utilisateur

L'interface du système *PAUSE* est réalisé avec *Suntools* un environnement de programmation d'applications interactives construit au dessus de *C* et *UNIX*.
La principale composante de ce système est la fenêtre; elle peut être composée de sous-fenêtres de type *panel*, *texte* ou *tty*. Un *panel* peut contenir des libellés, des boutons, des menus, ...
Une sous fenêtre de type *texte* contient du texte, une fenêtre de type *tty* contient les commandes *UNIX*.

Dans notre cas, et comme signalé dans le figure 6.1, l'utilisateur dispose de la fenêtre principale *PAUSE*, plus d'autres fenêtres pour les commandes *UNIX* et en particulier pour l'éditeur *EMACS* avec lequel il pourra composer ou modifier la spécification et les propriétés à prouver.
Sur la fenêtre initiale *PAUSE*, la sous-fenêtre *texte* contient la description de chaque commande disponible dans le système; la sous- fenêtre *panel* contient les cases correspondantes à chacune d'elles; et la sous-fenêtre *tty* contient l'environnement *Lisp* composé des différente fonctions qui réalisent la preuve.
Quand l'utilisateur clique sur une case, la sous fenêtre texte change pour lui donner plus de détails sur la commande concernée.

La suite de ce paragraphe sera consacrée à la description des fonctionalités de chacune de ces commandes.



2.1 Création de l'environnement de la preuve

La création de l'environnement de la preuve se fait en deux étapes :

- la définition des différents types qui composent la spécification.
- La définition de l'ordre d'utilisation de ces types pendant la réduction.

Lecture d'un type

LECT_TYPE Figure 6.2
Cette directive permet de lire un type qui se trouve dans le fichier spécifié en paramètre, et de transformer ses équations en règles de réécriture en respectant les définitions et les conditions vues dans le chapitre 4.
Syntaxe : La description de cette syntaxe est donnée par une grammaire à contexte libre. Les non-terminaux sont écrits en majuscules. la notation $X \| Y$ désigne une alternative entre X et Y. La notation $\{X\}^*$ désigne une répétition 0 ou n fois de la clause X.

type	→ OPERATEURS l-opérateurs AXIOMES l-axiomes FAXIOMES THEOREMES l-théorèmes FTHEO
l-opérateurs	→ opérateur {, opérateur}*
opérateur	→ IDENTIFICATEUR
(un opérateur peut être un identificateur quelconque)	
l-axiomes	→ axiome {axiome}*
axiome	→ NATUREL : mg == md
mg	→ expression
md	→ expression ‖ SI l-cond ALORS expression FSI ‖ SI l-cond ALORS expression SINON expression
l-cond	→ cond {ET cond}*
cond	→ expression-bool ‖ expression-bool == VRAI ‖ expression-bool == FAUX
l-théorèmes	→ théorème {théorème}*
théorème	→ expression {connect expression}*
connect	→ ET ‖ OU ‖ IMPLIQUE ‖ EQUIVALENT

La partie *théorème* d'un type contient des théorème triviaux ou supposés vrai et qui pourront être démontrés après.

L'ordre dans lequel sera donnée la liste des opérateurs de la spécification sera utilisé pour orienter les règles dans les cas litigieux.

Stratégie d'utilisation des types

STRATEGIE Figure 6.3
Cette directive permet dans le cas où la spécification est composée de plusieurs types, de définir l'ordre dans le quel seront utilisés ces types pendant la réduction. Les parties théorèmes de tous les types seront regroupées pour être utilisées dans chaque étape de réduction. Cela permet, comme nous avons vu dans le chapitre 4 de réduire le temps d'exécution au moment de la réécriture des termes.

2.2 Définition de la propriété élémentaire à prouver

LECT_THEO Figure 6.4
Cette directive permet de lire la formule équationnelle à prouver qui se trouve dans le fichier donné en paramètre.
Si la preuve de cette formule nécessite une induction et si le schéma d'induction ne peut pas être généré automatiquement, l'utilisateur doit utiliser ce même fichier pour donner le schéma d'induction.
Syntaxe

théorème {schéma}$^{0/1}$
schéma → SCHEMA P(l-var-induc) <= l-prop FSCHEMA .
l-var-induc → IDENTIFICATEUR {, IDENTIFICATEUR}*
(liste des variable de la formule à prouver sur les quelles se fera l'induction)
l-prop → prop {connect prop}*
prop → expression || P(l-param)
l-param → param {, param}*
param → opérateur(IDENTIFICATEUR {, IDENTIFICATEUR}*)
(Ces identificateurs doivent être différents des variables d'induction)

SCHEMA-AUT Figure 6.5
Cette directive permet de générer automatiquement le schéma d'induction à partir des variables d'induction et des constructeurs associés aux sortes de chacune d'elles. Les fonctions de cette directive sont une implantation des définitions du chapitre 3.

2.3 Visualisation de la preuve

Niveau de trace

NIVEAU_TRACE
Pour visualiser la preuve, l'utilisateur a deux choix de trace :

- Le premier niveau de trace est un niveau de débogage, permet de visualiser toutes les étapes de la preuves, avec toutes les réductions des termes et des sous termes, tous les chaînages transitifs, avec tous les retours arrières possibles. Cela permet à l'utilisateur de comprendre sa preuve et dans le cas où elle échoue de pouvoir détecter pourquoi.
- Le deuxième niveau de trace permet de visualiser seulement les différents nœuds de l'arbre au fur et à mesure de leur construction

Positionnement sur un nœud

VISUALISER
Cette directive permet de se positionner et de visualiser le nœud dont le numéro est donné par l'utilisateur

Exécution sur un fichier

FICHIER
L'exécution se fait par défaut sur l'écran; mais si l'utilisateur le désire il peut la diriger sur un fichier. Il doit alors donner le nom de celui-ci.

2.4 Lancement de la preuve

PREUVE Figure 6.6
Cette directive permet de lancer la preuve sur le dernier théorème lu. Si ce théorème était accompagné d'un schéma d'induction ou si ce schéma a été généré automatiquement alors la preuve se fait par induction; sinon ça sera une preuve équationnelle.

RE-PROUVER
Après s'être positionné sur un nœud, cette directive permet de relancer la preuve sur ce nœud. Cela peut se faire lors d'une nouvelle induction sur d'autres variables, ou après modification de la spécification.

2.5 Aide à l'utilisateur

MANUEL
Cette directive permet de visualiser sur une fenêtre un manuel contenant plus de détail sur chaque commande.

Figure 6.3

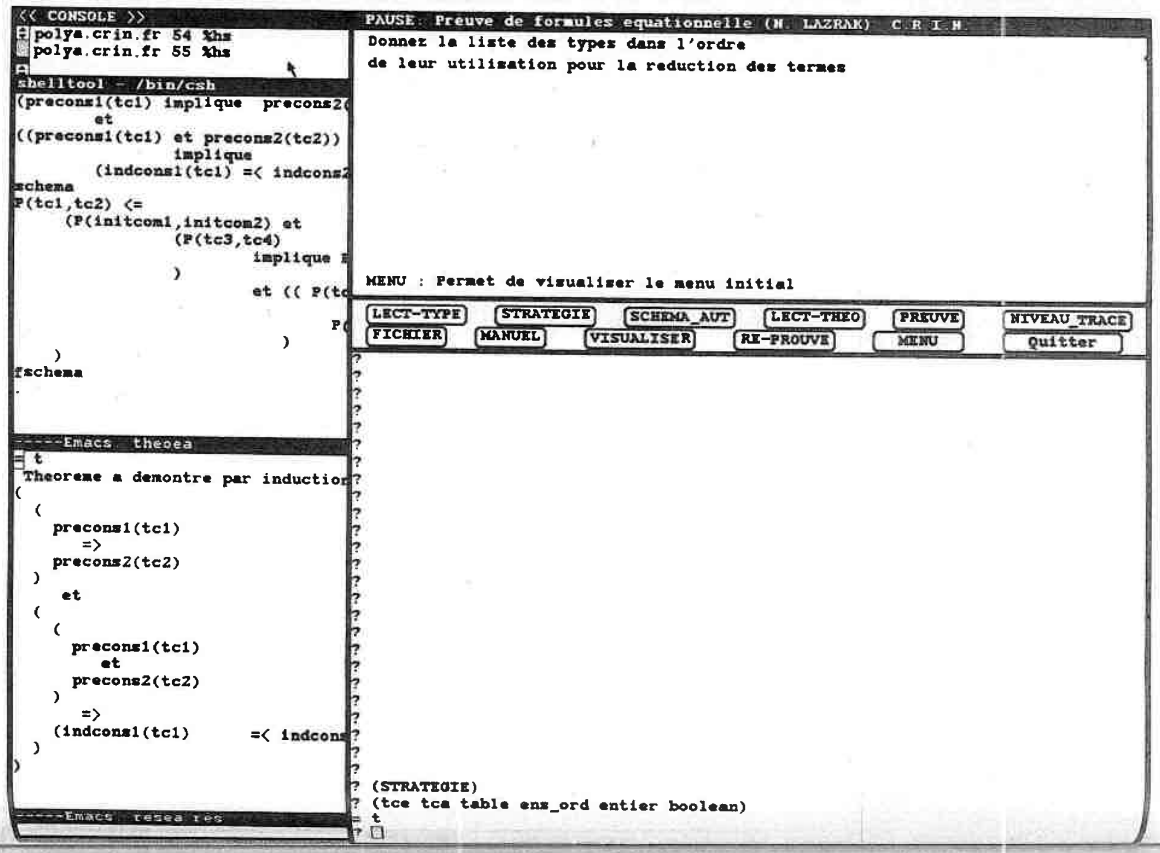


Figure 6.2

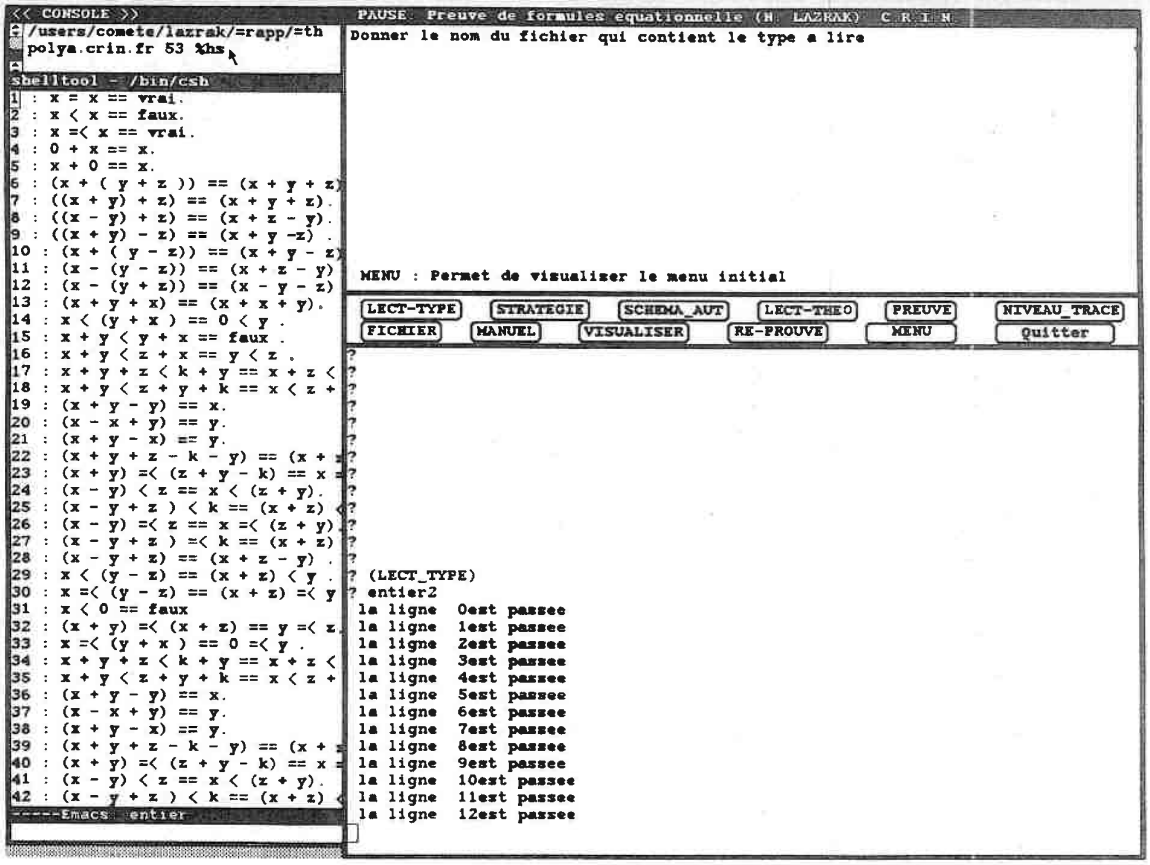


Figure 6.4

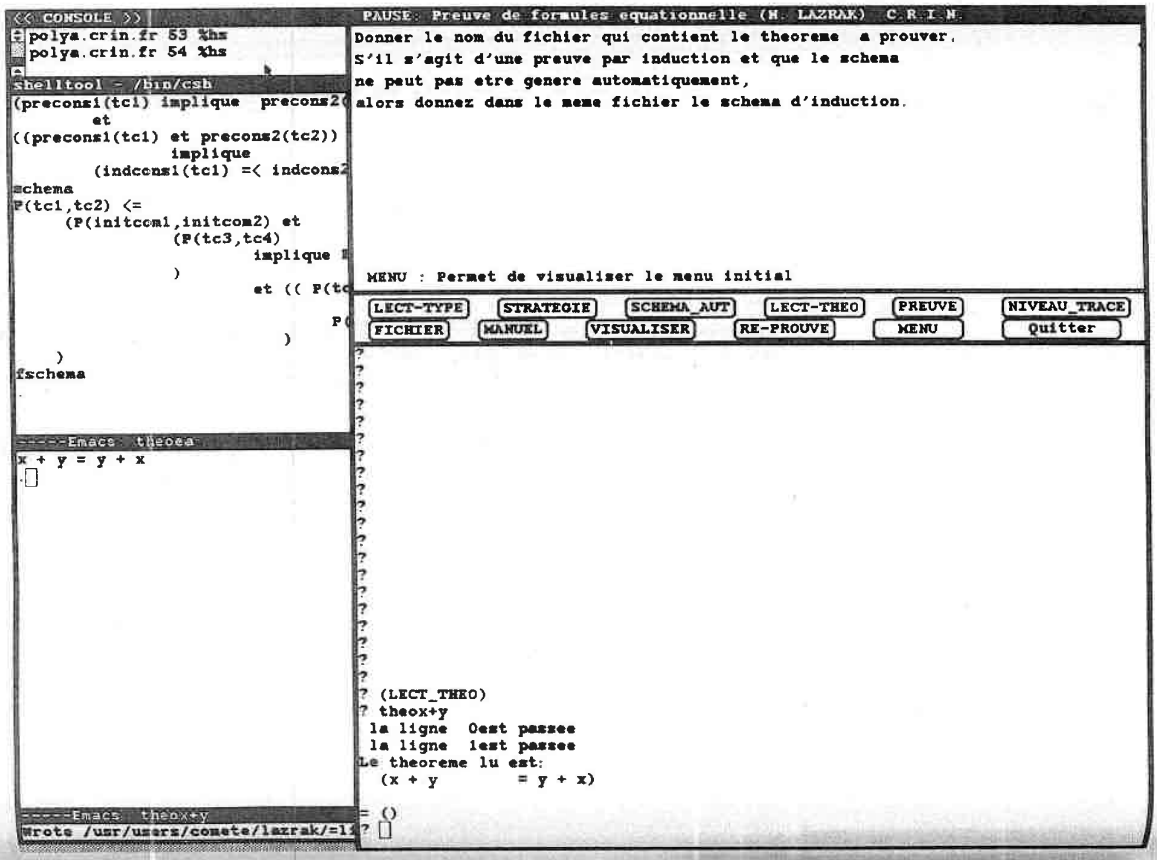
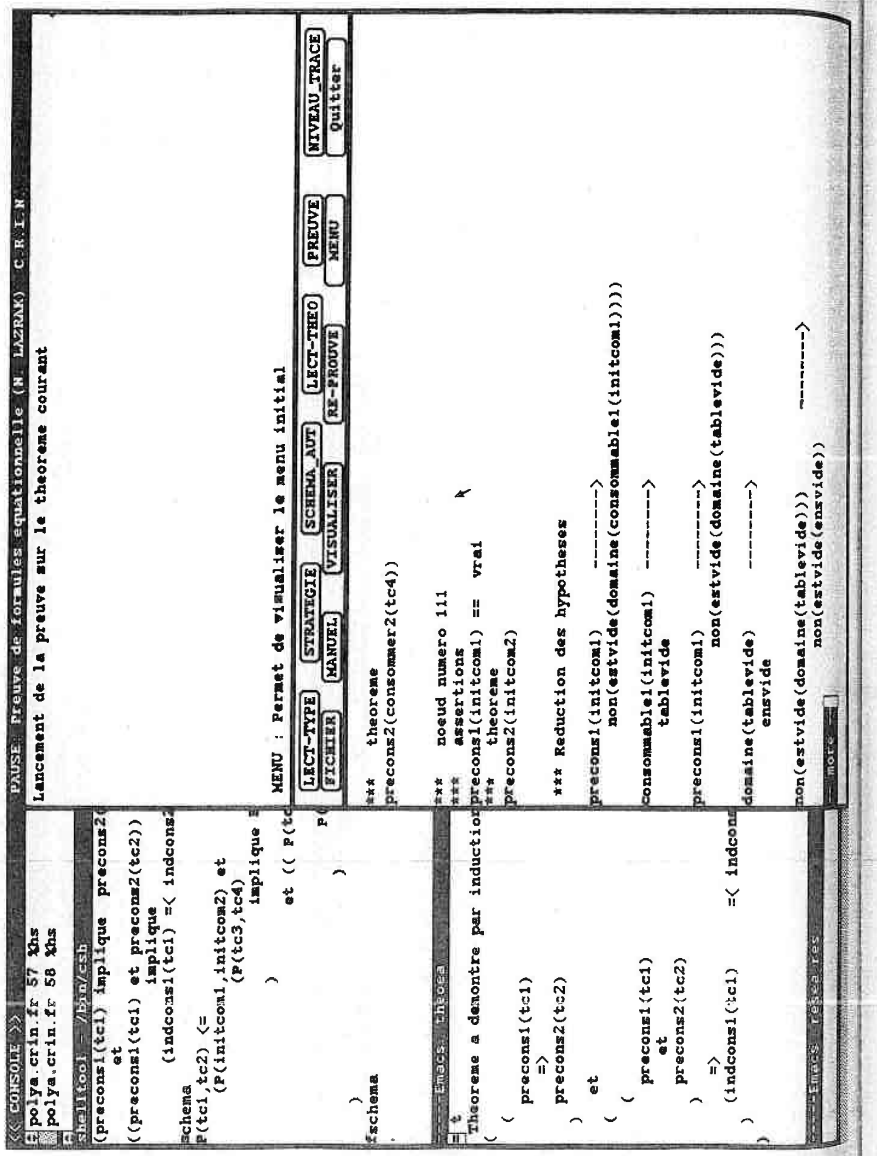


Figure 6.5

Figure 6.6



3 Comparaison avec les systèmes de preuve existants

Avant de faire la comparaison entre notre système et les systèmes de preuve existants, nous allons donner trois exemples de trois catégories de systèmes de preuve. Il ne s'agit pas d'une étude détaillée, mais seulement d'un résumé pour donner une idée sur ces systèmes.

3.1 Les système de preuve génériques

Isabelle [Pau87] et [Pau88] est un système de preuve interactif, écrit en *ML* [CPH*85], qui supporte une variété de logique comme la théorie des types de Martin-Löf [Mar73], la logique classique, la logique intuitionniste [Lau70] ou la théorie des ensembles de Zermelo-Fraenkel. Pour pouvoir raisonner sur ces différentes logiques objets, ce système possède sa propre méta-logique fondée sur la logique d'ordre supérieur (heigher-order logic) construite à partir du λ-calcul typé [HS86] et des règles de méta-inférence.

Pour faire une preuve l'utilisateur doit commencer par définir sa logique objet. Pour cela il doit donner sa syntaxe sous forme de λ-termes ainsi que son ensemble de règles d'inférence et d'axiomes (un axiome est une règle d'inférence sans premisses).

Avec *Isabelle*, une étape de preuve est une instance du même objet qu'une règle d'inférence (c'est une preuve à la Gentzen [Gen55]); et seul la nomenclature permet de ne pas confondre les deux concepts. Ainsi pour une étape de preuve, il s'agira du but et des sous-buts tandis que pour une règle d'inférence il s'agira de conclusion et de prémisses.

La déduction se fait par chaînage arrière à partir des buts en respectant les étapes suivantes :

- Le sous-but à réduire est unifié avec la conclusion de la règle d'inférence choisie;
- Les premisses de cette règles sont instanciées avec la substitution résultante;
- Le sous-but est remplacé par les prémisses instanciées.

Puisque les expressions manipulées sont des λ-termes l'unification utilisée est une unification d'ordre supérieur telle qu'elle est décrite dans [Hue73] ou dans [Pau86].

Durant le déroulement de la preuve, il se peut que l'utilisateur soit obligé d'appliquer la même règle d'inférence plusieurs fois; la définition des tactiques peut faciliter cette tache. L'environnement de programmation ML [CPH*85] permet de définir des structures de contrôles sur l'application des tactiques comme des conditionnelles, des répétitions, des recherches en profondeur, ...

3.2 Environnements de spécification

OASIS [Pau85] est un environnement d'aide au développement des spécifications algébrique. Le but de ce système est de fournir à l'utilisateur un laboratoire de spécification avec un ensemble d'outils qui lui permettent de construire et de valider ses spécifications.

Parmi les outils disponibles dans *OASIS* on trouve : un compilateur de types algébrique, un évaluateur symbolique de termes, un évaluateur de représentation des types, une bibliothèque des types les plus utilisés et un prouveur de théorème. Nous allons nous intéresser plus particulièrement à ce prouveur de théorème. Il s'agit d'un éditeur de preuve fondé essentiellement sur l'induction structurelle, la réécriture et le raisonnement par cas manuel. La preuve est dirigée par l'utilisateur en fournissant chaque fois les commandes et les données nécessaires pour la poursuite de la preuve.

3.3 Environnements de réécriture conditionnelle

Reveur4 [Zha84] et [BR87b] est un environnement de réécriture conditionnelle dont le but est de tester la consistance des spécifications algébriques et de prouver la validité des équations conditionnelles. Il est fondé sur le concept de la réécriture hiérarchique à deux niveaux appelé aussi la réécriture contextuelle [Rem82] et dont nous avons parlé dans le chapitre 4 lors de la comparaison avec la réécriture entre les a_termes. La spécification initiale est composée de deux parties:

- la partie primitives, qui définit les opérations primitives et les prédicats. Elle ne contient pas d'équations conditionnelles.
- La partie supérieure contient la définition des nouveaux opérateurs. les équations de cette partie peuvent être des équations conditionnelles. Les conditions sont des équations booléennes qui ne doivent contenir que des opérateurs primitives.

Cette spécification doit être suffisament complète par rapport aux booléens et doit vérifier une condition de bon recouvrement qui dit que la conjonction des conditions qui correspondent à un même terme gauche d'une équation doit être égale à *vrai*.
Dans ce cas le système utilise une extension de la procédure de Knuth et Bendix à la théorie conditionnelle [ZR85] et [Bou88b] pour prouver la confluence close du système de réécriture associé à la spécification.

La preuve de validité inductive d'une équation conditionnelle se fait par la méthode de l'induction sans induction: dans le cas d'un système de réécriture confluent sur les termes clos, l'équation à prouver est ajoutée au système et la procédure de complétion est rappelée.

3.4 Critique

Remarque 6.2

Nous voulons tout d'abord signaler que la conception de notre système de preuve PAUSE a été influencé par

- *Le système OASIS; Cela vient du fait que celui-ci est un éditeur de preuve, ce qui est un de ses avantages, car il permet à l'utilisateur de diriger la preuve lui même et par suite de voir les raisonnements qui sont automatisables et ceux qui manquent. Il nous a permis de découvrir que la réduction des assertions est nécessaire dans de nombreux cas, que l'utilisation des équations conditionnelles de la forme si – alors – sinon permettrait d'automatiser le raisonnement par cas. Il nous a permis aussi de constater la nécessité du chaînage transitif en présence de termes contenant des relations d'ordre.*
- *Le système Reveur4 qui nous a permis de bien comprendre les mécanismes de la réécriture.*

Nous n'allons pas faire cette critique pour dire qu'un système est le meilleur, car chacun de ces système a été conçu pour des buts différents et chacun d'eux possède ses avantages dans son domaine.

Les systèmes de preuves génériques ont l'avantage de supporter plusieurs logiques objets et permettent ainsi la preuve d'une large classe de propriétés appartenant à ces différentes logiques. Mais cela n'est pas sans peine pour l'utilisateur car il lui faut pour chaque logique objet et pour chaque classe de propriétés écrire les tactiques nécessaires pour la preuve. Dans [Nip88] l'auteur

avait proposé les tactiques nécessaires pour le raisonnement équationnel en utilisant *Isabelle* et il a conclu son article par: *Il est clair qu'il y a un problème évident:c'est le problème de l'efficacité. Les tactiques écrites avec Isabelle sont cent fois moins rapides que les systèmes spécialisés dans le raisonnement équationnel. D'où le problème éternel du choix entre la généricité et l'efficacité.*

Le même problème d'efficacité se pose dans le cas de l'utilisation des éditeurs de preuves tel que *OASIS*, surtout quand l'utilisateur est amené à raisonner sur des spécifications avec un grand nombre d'équations. Dans ce cas il ne pourra faire confiance qu'à un système qui fait cette preuve automatiquement.

Le système *Reveur4* est un bon outil pour tester la consistance des spécifications hiérarchiques à deux niveaux. Et nous avons signalé dans le chapitre 4 que des études théoriques sont entamés dans [Bou88b] pour l'étendre aux spécifications hiérarchiques à plusieurs niveaux. Tandis que pour le cas de la validité inductive, nous avons montré dans le chapitre 3 les avantages de l'induction structurelle sur l'induction sans induction.

Il existe d'autres systèmes de preuve, tel que *LP* [GG89] qui est une extension du système *Reve* [Les83]. Le système *LP* fournit à l'utilisateur un nombre important de techniques de preuves dans le cadre du raisonnement équationnel; mais cela seulement dans le cas des spécifications non conditionnelles.

Pour conclure cette critique, nous voudrions signaler que chaque système de preuve admet des avantages et des limites; et qu'il peut être acceptable qu'un système soit incapable de prouver certains théorèmes dans certaines théories, à condition que celui-ci puisse le faire d'une manière efficace pour d'autres théorèmes qui intéressent l'utilisateur. Autrement dit, il peut être acceptable qu'un système soit incomplet mais efficace.

3.5 Comparaison

Dans [Laz89] nous avons fait une première comparaison entre notre système *PAUSE* et les autres systèmes de preuves existants. Cette comparaison avait pour objectif de montrer quelques raisons qui nous ont amener à concevoir un nouveau système de preuve. Elle portait sur quatre points:

- La forme des équations dans les spécifications: Alors que la plupart des systèmes de preuve existants utilisent des spécifications avec seulement des équations inconditionnelles ou des équations conditionnelles de la forme *si – alors*; nous autorisons en plus l'utilisation des équations de la forme *si – alors – sinon*. Cela n'a pas d'influence directe sur la sémantique des spécifications, mais il présente plus d'avantages puisqu'il permet de définir les opérations d'une manière exhaustive, et de faire des raisonnements par cas automatique.
- La forme des propriétés élémentaires à prouver: Les systèmes de preuves existants permettent de prouver des propriétés sous forme d'équations simples ou de clauses de Horn équationnelles. Notre objectif est de pouvoir prouver une classe plus grande de formules équationnelles telle que nous l'avons définie dans le chapitre 2.
- La supposition de propriétés globales sur la spécification: Nous avons choisi la méthode fondée sur l'induction structurelle pour prouver les théorèmes inductifs. Et nous avons montré dans les chapitre 3 et 4 que cette méthode ne nécessite que la terminaison comme propriété globale sur les spécifications initiales. Ce qui n'est pas le cas pour les méthodes fondées sur l'induction sans induction.

- Le degré d'interaction avec l'utilisateur : Le fait qu'il y ait beaucoup d'interaction présente un grand inconvénient d'efficacité en présence des spécifications avec un grand nombre d'équations; et l'utilisateur risque de ne plus pouvoir gérer sa preuve. Ainsi nous avons voulu réduire cette interaction au minimum telle que la donnée du schéma d'induction ou l'adjonction d'équations et la relance de la preuve.

Partie III

Application

Chapitre 7

Vérification de propriétés de programmes parallèles

1 Introduction

Le domaine d'application des prouveurs de théorèmes est très varié. Ils sont utilisés dans presque tous les domaines qui touchent à l'informatique, comme la validation des circuits VLSI [Gor87], [SBGP87] et [GGS88], ou la validation des protocoles de communication [STE*82].
Nous avons choisi comme domaine d'application le cadre des programmes parallèles car la tendance actuelle est de concevoir des programmes qui peuvent s'exécuter sur des machines parallèles.

L'objectif de ce chapitre est de présenter un langage intermédiaire dans la conception des programmes parallèles; puis de montrer comment vérifier qu'un programme est *"plus asynchrone"* qu'un autre.

Ce langage d'expression de programmes parallèles a été défini dans [Per85]. Il a été conçu pour être un outil de construction méthodologique et de validation de programme parallèle.

Le parallélisme introduit au moment de la résolution d'un problème est un parallélisme dit de *résolution* ou de *conception*. Il a été introduit pour des raisons d'efficacité du programme obtenu.

Contrairement aux travaux qui s'intéressent au parallélisme du point de vu synchronisation entre processus [Sif79,Mil83], nous nous sommes basés pour exprimer le parallélisme sur la communication entre des processus non déterministes indépendants. De ces relations de communication sont déduites les synchronisations des processus nécessaires à leur réalisation. Ces communications sont réalisées par l'intermédiaire de ports typés par des *types de communication*. Cette notion de type de communication permet de séparer entre les relations de calcul et les relations de communication. Cette indépendance permet de jouer sur le degré de synchronisation des processus communicants, et par conséquence sur l'efficacité des programmes.

Notre contribution dans ce chapitre se situe dans

- l'expression des types de communication sous forme d'une spécification équationnelle,

- la formalisation de certaines relations entre les types de communication ainsi que leur preuve automatique,
- puis la présentation sur un exemple des conséquences sur la modification des programmes parallèles.

2 Type de Communication

Comme nous l'avons signalé dans l'introduction de ce chapitre, la méthode proposée permet d'exprimer le parallélisme en s'appuyant sur la communication entre les processus. Ces communications sont définies en terme de relations entre les valeurs de données échangées. En effet, dans un univers parallèle, deux types de relation sont distinguées

- Une relation statique, appelée relation de calcul, entre les suites de données et les suites de résultats.
- Une relation, appelée relation de communication [JP85], qui transforme généralement le nombre d'occurrences et l'ordre des termes des suites sans en modifier les valeurs.

Une relation de communication exprime le choix du processus consommateur de la valeur à consommer parmi les valeurs émises par le processus producteur. Elle permet aussi d'indiquer comment doit être modifié cet ensemble après cette consommation. Par exemple une relation de communication *égalité* spécifie un système composé d'un processus producteur et d'un processus consommateur, tel que à tout instant de communication, la valeur utilisée par le processus consommateur est la plus ancienne valeur émise par le producteur et non encore consommée. Cette valeur doit disparaître de l'ensemble des valeurs émises après la consommation.

Ces relations de communication sont exprimées dans un premier temps en terme de relations entre des suites temporelles [Laz86] et [Mar89]. En effet, il s'agit d'établir les relations entre les suites de données échangées. La nature même de ces relations sous-entend l'expression du temps à travers expressions comme "à tout instant", "plus ancienne", ... Ensuite la notion de *type de communication* permet de donner une représentation algébrique et constructive de ces relations. La preuve de représentation d'une relation de communication par un type de communication est étudiée dans [Laz86] et [Mar89].

Nous n'allons pas donner plus de détails sur les relations de communication entre les suites temporelles. Par contre nous allons nous intéresser aux types de communication.

2.1 Définition

L'élément essentiel d'un type de communication est cet ensemble de valeurs émises par le processus producteur et dont dispose le processus consommateur. Cet ensemble sera appelé *ensemble consommable*. L'ensemble des valeurs émises sera appelé *ensemble produit* et l'ensemble des valeurs reçues sera appelé *ensemble consommé*.

Formellement un objet tc d'un type de communication $TYPCOM$ est un triplet d'objets, chacun d'eux étant de type *table*, noté $tc =< EP, A, EC >$:

- EP: Ensemble produit,
- A : Ensemble consommable,

- EC: Ensemble consommé.

Un type de communication est muni de trois opérations pour manipuler les données:

- *init_com*: Initialiser la communication entre les processus.
- *produire*: Emettre une donnée. Tout processus qui active cette opération est appelé *producteur*.
- *consommer*: Recevoir une donnée communiquée. Tout processus qui active cette opération est appelé *consommateur*.

Ce sont les constructeurs d'un type de communication.

Un type de communication est muni aussi de trois projections:

- *ens_prod*: pour accéder à l'ensemble produit.
- *ens_cons*: pour accéder à l'ensemble consommé.
- *consommable*: pour accéder à l'ensemble consommable.

Les quatre dernières opérations permettent de constater l'évolution d'un type de communication:

- *pre_cons*: Précondition sur la consommation. Elle restreint le domaine de définition de l'opération de consommation du processus consommateur.
- *post_cons*: Post_condition de la consommation. Elle définit l'état de l'ensemble consommable, résultant d'une opération de consommation.
- *val_cons*: Permet de déterminer la valeur de la donnée communiquée prise en compte par l'opération de consommation.
- *ind_cons*: Permet de donner l'indice dans l'ensemble consommable de cette valeur communiquée prise en compte par l'opération de consommation.
- *indice*: Permet de donner dans le cas d'une production l'indice dans l'ensemble produit de la valeur produite; et dans le cas d'une consommation l'indice dans l'ensemble consommable de la valeur consommée (ind_cons).

Le profil de ces opérations est le suivant:

init_com	:	$\longrightarrow TYPCOM$
produire	: $TYPCOM \times ELT$	$\longrightarrow TYPCOM$
consommer	: $TYPCOM$	$\longrightarrow TYPCOM$
ens_prod	: $TYPCOM$	$\longrightarrow$ *table*
ens_cons	: $TYPCOM$	$\longrightarrow$ *table*
consommable	: $TYPCOM$	$\longrightarrow$ *table*
indice	: $TYPCOM$	$\longrightarrow$ *entier*
pre_cons	: $TYPCOM$	$\longrightarrow$ *bool*
post_cons	: $TYPCOM$	$\longrightarrow$ *table*
ind_cons	: $TYPCOM$	$\longrightarrow$ *entier*
val_cons	: $TYPCOM$	$\longrightarrow ELT$

Remarque 7.1

Les quatre dernières opérations sont caractéristiques d'un type de communication. En effet, pour choisir la valeur à consommer parmi l'ensemble des valeurs émises par le processus producteur, nous utilisons les opérations pre_cons, val_cons et ind_cons; et pour déterminer comment doit être modifié cet ensemble, nous utilisons l'opération post_cons.

2.2 Équations communes à tous les types de communication

Les équations communes aux définitions de tous les types de communication sont les suivantes :

$tc : TYPCOM$
$v : ELT$

1 : $consommable(init\_com) == tablevide$
2 : $ens\_prod(init\_com) == tablevide$
3 : $ens\_cons(init\_com) == tablevide$
4 : $consommable(produire(tc, v)) == mettre(consommable(tc), maximum(ens\_prod(tc)) + 1, v)$
5 : $ens\_prod(produire(tc, v)) == mettre(ens\_prod(tc), maximum(ens\_prod(tc)) + 1, v)$
6 : $ens\_cons(produire(tc, v)) == ens\_cons(tc)$
7 : $consommable(consommer(tc)) == post\_cons(tc)$
8 : $ens\_prod(consommer(tc)) == ens\_prod(tc)$
9 : $ens\_cons(consommer(tc)) == mettre(ens\_cons(tc), maximum(ens\_cons(tc)) + 1, val\_cons(tc))$
10: $indice(init\_com) == 0$
11: $indice(produire(tc, v)) == maximum(ens\_prod(tc)) + 1$
12: $indice(consommer(tc)) == ind\_cons(tc)$
13: $val\_cons(tc) == acces(consommable(tc), ind\_cons(tc))$

Précondition :
La précondition de l'opération $consommer(tc)$ est $pre\_cons(tc)$.

Pour que la spécification d'un type de communication ait la forme générale d'une spécification définie dans le chapitre 1, nous allons intégrer la précondition de consommation dans les équations. Nous obtenons alors les équations suivantes :

1 : $consommable(init\_com) == tablevide$
2 : $ens\_prod(init\_com) == tablevide$
3 : $ens\_cons(initcom) == tablevide$
4 : $consommable(produire(tc, v)) == mettre(consommable(tc), maximum(ens\_prod(tc)) + 1, v)$
5 : $ens\_prod(produire(tc, v)) == mettre(ens\_prod(tc), maximum(ens\_prod(tc)) + 1, v)$
6 : $ens\_cons(produire(tc, v)) == ens\_cons(tc)$
7 : $consommable(consommer(tc))$== si $pre\_cons(tc)$ alors $post\_cons(tc)$
8: $ens\_prod(consommer(tc))$ == si $pre\_cons(tc)$ alors $ens\_prod(tc)$
9: $ens\_cons(consommer(tc)$ == si $pre\_cons(tc)$ alors $mettre(ens\_cons(tc), maximum(ens\_cons(tc)) + 1, val\_cons(tc))$
10: $indice(init\_com)$ == 0
11: $indice(produire(tc, v))$ == $maximum(ens\_prod(tc)) + 1$
12: $indice(consommer(tc))$ == si $pre\_cons(tc)$ alors $ind\_cons(tc)$
13: $val\_cons(tc)$ == $acces(consommable(tc), ind\_cons(tc))$

Les opérations *mettre, tablevide, acces, maximum, minimum, domaine* sont des opérations importées du type *table* (exemple 4.5).

2.3 Équations caractéristiques d'un type de communication

Ces équations caractérisent la stratégie de communication d'un type donné.

Exemple 7.1

Le type de communication égalité *est caractérisé par le fait que toute valeur émise est reçue une et une seule fois, et l'ordre des réceptions est identique à l'ordre des émissions. Les équations caractéristiques de ce type sont les suivantes :*

- $pre\_cons(tc) == non(estvide(domaine(consommable(tc))))$
- $ind\_cons(tc) == minimum(consommable(tc))$
- $post\_cons(tc) == enlever(consommable(tc), minimum(consommable(tc)))$

Exemple 7.2

Le type de communication aléatoire *est caractérisé par le fait que chaque valeur reçue est la dernière émise à cet instant, certaines valeurs ne sont pas reçues et d'autres peuvent l'être plusieurs fois, selon le rythme aléatoire des processus producteur ou consommateur. Les équations caractéristiques de ce type sont les suivantes :*

- $pre\_cons(tc) == non(estvide(domaine(consommable(tc))))$.
- $ind\_cons(tc) == maximum(consommable(tc))$
- $post\_cons(tc) == mettre(tablevide, ind\_cons(tc), val\_cons(tc))$

Exemple 7.3

Le type de communication égalité-presque-partout *est caractérisé par le fait que toute valeur émise est reçue une fois et une seule dans le cas où la production survient assez tôt. En d'autres termes le $i^{ème}$ élément produit est remplacé par ω dans la suite consommée, si la production de cet élément est en retard sur la consommation. Les équations caractéristiques de ce type sont les suivantes :*

- $pre\_cons(tc) == vrai$
- $ind\_cons(tc) == si\ cardinal(ens\_prod(tc)) < cardinal(ens\_cons(tc)) == vrai$
  $alors\ \omega$
  $sinon\ minimum(consommable(tc))$
- $post\_cons(tc) == si\ cardinal(ens\_prod(tc)) < cardinal(ens\_cons(tc)) == vrai$
  $alors\ tablevide$
  $sinon\ enlever(consommable(tc), minimum(consommable(tc)))$

Remarque 7.2

Dans [Per85], les types de communication ont été définis avec une spécification algébrique en pre_post. *Nous avons choisi une présentation équationnelle dans un but de faciliter la manipulation des opérations et des termes, car notre but, rappelons le, est d'automatiser la preuve de relations entre les types de communication.*

2.4 Automate associé à un type de communication

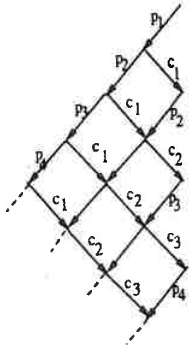
Pour modéliser d'un système parallèle réduit à ses opérations de production et de consommation sur un port donné, nous associons à chaque type de communication un automate figuré par un graphe orienté représentant les transitions possibles. Les nœuds de ce graphe représentent les états du système. Les arcs sont étiquetés par l'opération sur la donnée communiquée :

- p : correspond à une production.
- c : correspond à une consommation.

Ce graphe doit respecter les conditions suivantes :

- Une opération de consommation c est activable si et seulement si la précondition de consommation *pre_cons* est vérifiée.
- A chaque étiquette p ou c d'un arc est associée un indice tel que :
  - Pour une opération de production p, cet indice est égal à l'indice dans l'ensemble produit de l'élément résultant de l'occurrence de production associée.
  - Pour une opération de consommation c, cet indice, s'il existe, est égal à
    - l'indice dans l'ensemble consommable de l'élément résultant de l'occurrence de consommation associée (ind_cons)
      ou
    - ω, si ω est la valeur indéfinie résultat de cette opération.
- Les productions sont étiquetées dans l'ordre croissant des indices.

Exemple 7.4



Automate associé au type de communication égalité

3 Langage d'expression du parallélisme

Après avoir introduit les types de communication qui sont les éléments de base, nous allons donner un aperçu général sur les autres concepts de ce langage.

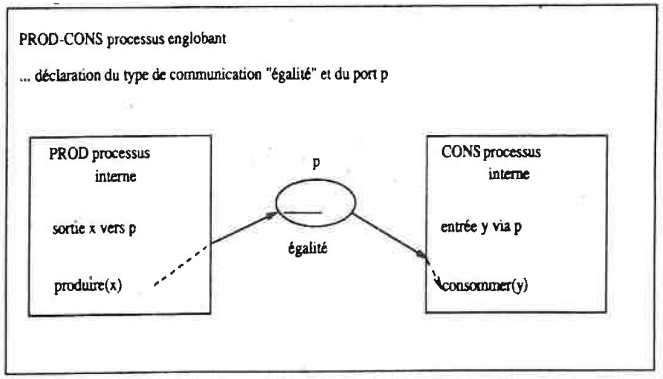
Le langage présenté dans [Per85] ne prétend pas être un langage de plus pour la programmation des systèmes parallèles, mais il a été conçu pour être utilisé comme un outil de construction méthodologique et de validation de programmes parallèles. Le passage aux autres langage d'expression du parallélisme a été étudié dans [KFJP88] pour le langage *ADA* ou dans [EJ89] pour le langage *OCCAM*.

L'expression du parallélisme dans ce langage est fondée sur la coopération de processus non déterministes définis de manière statique qui communiquent de manière asynchrone par des ports. L'utilisation des types de communication permet de séparer dès le début de la conception les relations de calcul et les relations de communication. Cela permet aussi de définir les relations de communication entre processus indépendemment de tout souci de synchronisation. Ce concept constitue le principal apport par rapport aux langages contemporains d'expression du parallélisme qui sont fondées sur la communication tels que *CSP* [Hoa78], *ADA* [Ada80], *LCS* [Ber85] ou LC^3 [LC86].

3.1 Programme parallèle

Un programme ou système parallèle est constitué d'un ensemble hiérarchique de processus communicants. Les données communiquées circulent via des ports typés par des types de communication.

Exemple 7.5



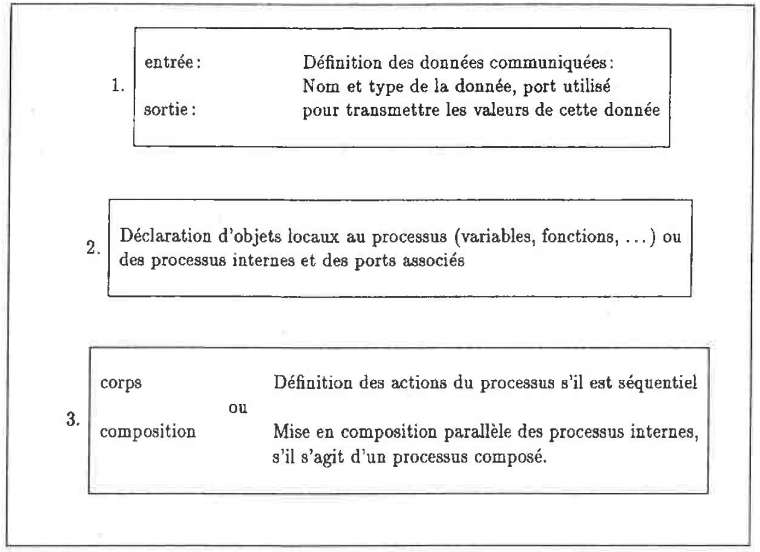
Ce schéma visualise un exemple simple de système parallèle (producteur consommateur).

Dans cette hiérarchie nous pouvons distinguer deux types de processus :

- Les processus élémentaires (feuilles de la hiérarchie) sont des processus séquentiels classiques non déterministes qui utilisent des constructions proches des processus CSP ou des taches ADA : séquence d'énoncés, choix et itération non déterministes).

- Les processus composés obtenus par composition parallèle d'autres processus.(en utilisant un type de communication)

En règle générale un processus communicant élémentaire ou composé a la structure suivante :



- La partie (1) constitue l'interface du processus. Dans le cas du processus racine cette partie est vide.

- Dans le cas d'un processus composé communicant, la partie (2) définit les processus internes et décrit leurs relations de communication par la déclaration :
  - des ports de communication utilisés :
    - nom du port,
    - type du port : type de communication utilisé.
  - des types de communication associés et non prédéfinis, s'ils n'ont pas été déjà déclarés dans un processus englobant.

- La partie (1) des processus internes définit l'utilisation des ports déclarés au niveau du processus englobant. Cette définition est réalisée par les clauses *entrée* ou *sortie* qui déclarent les données communiquées.

- L'activation en parallèle des processus internes déclarés dans la partie (2) d'un processus composé est définie de manière statique; la partie (3) est alors réduite à l'opérateur de composition //.

- L'activation des opérations *produire* ou *consommer* réalise la communication.

3.2 Les principales constructions du langage

Dans ce paragraphe nous allons nous contenter de donner la sémantique intuitive des principales constructions de ce langage. La syntaxe concrète de celui-ci est décrite dans l'annexe B par

une grammaire à contexte libre. Notons que la syntaxe de la partie séquentielle du langage est un sous ensemble de *Pascal*. La sémantique des différentes constructions est donnée en détail dans [Per85] en utilisant des compositions d'automates, ou dans [Mar89] où l'auteur combine cette dernière notion avec les formalismes proposés dans [Bou87].

1. **typcom** $TC(ELT)$,
   permet la déclaration du type de communication TC en donnant la définition algébrique des ses opérations caractéristiques (*pre_cons*, *post_cons*, *ind_cons*).

2. **port** p: $TC(ELT)$,
   permet la création d'un port p de type TC, résultat de l'opération *init_com* associé à ce type.
   Le port p est local au processus où il est déclaré, ce qui permet d'encapsuler les communications d'un sous système à l'intérieur d'un système composé.

3. **entrée** y: t **via** p,
   permet de déclarer une donnée communiquée dans une liste d'entrées d'un processus consommateur et de créer ainsi une variable y de type t locale à celui-ci. Cette déclaration permet aussi d'associer la variable au port de communication p dans l'objectif d'une éventuelle consommation.

4. **sortie** x: t **vers** $p_1, \ldots, p_n$,
   permet de déclarer une donnée communiquée dans une liste de sorties d'un processus producteur et de créer ainsi une variable x de type t locale à ce processus. Cette déclaration permet aussi d'associer la variable aux ports $p_1, \ldots, p_n$ dans l'objectif d'une éventuelle production.

5. Dans le contexte de l'énoncé 4, l'instruction **produire**(x) permet l'activation de l'opération *produire* définie dans les types de communication des ports $p_1, \ldots, p_n$.

6. Dans le contexte de l'énoncé 3, l'instruction **consommer(y)** permet l'activation de l'opération *consommer* définie dans le type de communication associé au port p. Selon que la précondition de consommation est vraie ou fausse deux cas sont à envisager:

   - *pre_cons*(p) est égale à *vrai* au moment de l'activation de cette instruction, dans ce cas l'opération *consommer* est activée.
   - *pre_cons*(p) est égale à *faux* au moment de l'activation de cette instruction, dans ce cas le processus consommateur est mis en attente jusqu'au moment où *pre_cons*(p) devient *vrai*.

   Si plusieurs processus consommateurs liés à un même port sont en attente, ils seront activés dans un ordre arbitraire, un seul pouvant à la fois accéder à ce port.
   Dans le cas où plusieurs processus veulent accéder simultanément à un port de communication, les accès seront séquentialisés selon un ordre arbitraire avec une priorité des opérations de production sur les opérations de consommation.

7. Le choix non-déterministe:

   select
   $g_1 \longrightarrow$ *énoncé*$_1$

   ou
   ⋮
   ou
   $g_n \longrightarrow$ *énoncé*$_n$
   fin select

   Les gardes g_i sont des expressions booléennes
   Cette instruction a pour signification:

   - d'évaluer simultanément toutes les gardes,
   - s'il existe une garde g_i égale à *vrai*, alors activer les instructions de *énoncé*$_i$,
   - si plusieurs gardes sont égaux à *vrai*, alors choisir l'une d'elles au hasard,
   - si aucune garde n'est égale à *vrai*, alors l'état résultant est un état d'erreur.

8. **composition**,
   permet d'activer en parallèle des processus fils qui ont été déclarés dans un processus englobant. La communication entre ces processus se fait à travers les ports déclarés dans la partie déclaration via les entrées et les sorties correspondantes dans chaque processus.

   Nous avons vu que ces processus fils peuvent être soit des processus élémentaires, soit eux aussi composés de processus communicants. Cela implique qu'un système parallèle peut avoir plusieurs schémas de composition suivant des hiérarchisations plus ou moins compliquées. L'exemple 7.5 montre une situation simple de communication bi-point. Cependant, plusieurs autres situations peuvent être considérées; citons par exemple:

   - Les situations de diffusion dans les quelles un processus producteur communique avec **n** processus consommateurs **via n ports typés** soit avec le même type de communication soit avec des types de communication différents.
   - Les situations de divergence dans les quelles un processus producteur communique avec un processus consommateur via un seul port. Mais le processus consommateur est composée de plusieurs processus, chacun d'eux est susceptible d'activer une opération de consommation.
   - Les situations de convergence diffèrent des situations de divergence par le fait que c'est le processus producteur qui est composé avec plusieurs processus, chacun d'eux est susceptible d'activer une opération de production.

Exemple 7.6

Ce programme parallèle est un exemple qui présente un système producteur consommateur avec un producteur qui communique avec deux consommateurs (diffusion des données communiqué) via deux ports associés à deux types de communication égalité.

programme *prod_conss::*
  typcom *égalité(entier)*
  - *définition du type de communication égalité*
  port p_1, p_2 *: égalité*

  processus *prod::*
    sortie x **entier vers** p_1, p_2;

```
        corps
          x:=1;
          répéter x ≤ 100 ⟶
                  produire(x);
                  x := x + 1
          fin répéter
      fin prod;

      processus cons1;
          entrée y : entier via p1;
          fonction pair(x:entier):boolean;
          - déterminer si un entier x est un nombre pair -
          fin pair;
          corps
            répéter consommer(y)
                    si pair(y) alors écrire(y) fsi
            fin répéter
      fin cons1

      processus cons2::
          entrée z : entier via p2;
          fonction premier(x:entier):boolean;
          - déterminer si un entier x est un nombre premier -
          fin premier;
          corps
            répéter consommer(z)
                    si premier(z) alors écrire(z) fsi
            fin répéter
      fin cons2

      composition
          prod//cons1//cons2
fin prod_conss
```

4 Relation entre les types de communication

Nous avons vu que l'expression du parallélisme retenue dans le langage présenté dans les paragraphes précédents est fondée sur la communication entre processus. Nous avons vu aussi que la notion de type de communication permettait de séparer les relations de calcul des relations de communication et que les seules synchronisations entre les processus sont dues aux contraintes d'accès en exclusion mutuelle aux ports de communication ou aux attentes d'un processus prêt à réaliser une opération de consommation alors que la précondition n'est pas satisfaite. Dans ce contexte, établir des relations d'ordre entre l'ensemble des types de communication permettra de jouer sur la synchronisation entre les processus d'un système parallèle. Cela est dû au fait que ce sont ces types de communication qui gèrent la communication et par conséquence régulent la synchronisation. Ces relations sont telles que si pour une donnée communiquée, nous substitutions un type de communication TC_2 à un type de communication TC_1 tel que TC_1 soit

en relation avec TC_2, le système devient *plus asynchrone* dans le sens que nous préciserons ci dessous.

Définition 7.1 Ensemble des termes définis associé à un type de communication

L'ensemble des termes définis associé à un type de communication TC, noté $DEF(TC)$ est un sous ensemble de l'ensemble des termes clos sur les constructeurs $T(C)$, où C = {init_com, produire, consommer}. *Il est défini par :*

- *init_com* $\in DEF(TC)$
- *soit* $t \in DEF(TC)$ *alors*
  - $produire(t, v) \in DEF(TC)$,
  - *si la précondition de consommation* $pre\_cons(t)$ *est vérifié alors* $consommer(t) \in DEF(TC)$.

Cet ensemble représente l'évolution d'un système parallèle réduit à ses opérations de production et de consommation sur un port de communication. En effet, chaque terme représente une séquence d'exécution possible de ce système.

Remarque 7.3

L'ensemble des termes définis associé à un type de communication TC est isomorphe à l'ensemble des chemins du graphe associé à celui-ci par l'homomorphisme suivant :

- $f(\wedge) = init\_com$. $\wedge$ *est le chemin vide.*
- $f(ch.p_i) = produire(f(ch), v)$.
- $f(ch.c_i) = consommer(f(ch))$. *ch est un chemin du graphe et le point définit la concaténation d'un arc à un chemin.*

Nous allons maintenant donner la définition des relations entre les types de communication en utilisant leur ensemble de termes définis respectifs.

Définition 7.2 Relation d'asynchronisme

Soient TC_1 et TC_2 deux types de communication et F_1 et F_2 l'ensemble des opérateurs associé tels que $F_i = \{init\_com_i, produire_i, consommer_i, consommable_i, ens\_prod_i, ens\_cons_i, ind\_cons_i, val\_cons_i\}$. Soient X_{TC_1}, X_{TC_2} et X_{ELT} trois ensembles de variables de sorte respectif TC_1 TC_2, et ELT Soit h un homomorphisme entre $T(F_1, X_{TC_1} \cup X_{ELT})$ et $T(F_2, X_{TC_2} \cup X_{ELT})$ tel que $h(op_1) = op_2$ où op_i représente chacun des opérateurs de F_i.
Le type de communication TC_2 est dit plus asynchrone que TC_1 si et seulement si :

- *la restriction de h à $DEF(TC_1)$ est un homomorphisme et*
- $h(DEF(TC_1)) \subseteq DEF(TC_2)$.

Remarque 7.4

- *L'introduction d'un homomorphisme dans la définition des relations entre deux types de communication signifie que nous nous intéressons qu'aux séquences d'exécution équivalentes.*

- *Cet asynchronisme vient du fait que l'ensemble des séquences d'exécutions possibles associé à TC_1 est inclus dans celui de TC_2. Cela n'est possible que parce que les préconditions sur les opérations de TC_2 sont plus faibles que celles sur TC_1. Cela a pour conséquence de réduire le temps d'attente des processus pendant les communications.*

Le concept de type de communication a été créé pour s'occuper plus de la gestion des communication que de la synchronisation entre les processus; c'est pour cela que nous allons nous intéresser plus aux relations qui font intervenir, en plus de la relation d'asynchronisme, des relations entre les positions des valeurs des données communiquées.

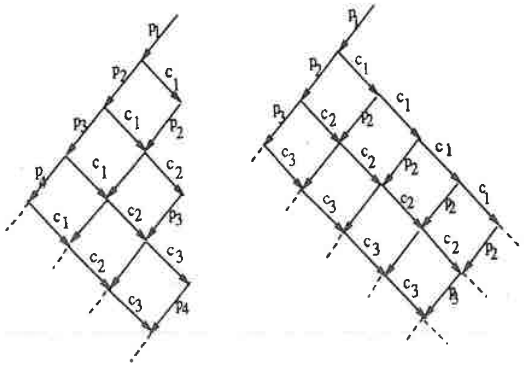
Définition 7.3 Relation d'asynchronisme avec perte des valeurs

Soient TC_1 et TC_2 deux types de communication. TC_2 est dit plus asynchrone que TC_1 avec possibilité de perte de valeurs; noté TC_1 as< TC_2 si et seulement si :

- *TC_2 est plus asynchrone que TC_1*
  et
- *pour chaque couple de termes (t_1, t_2) de $DEF(TC_1) \times DEF(TC_2)$ tel que $h(t_1) = t_2$, $indice_1(t_1) \leq indice_2(t_2)$*

Cette relation as< est une relation d'ordre partiel sur l'ensemble des types de communication.

Exemple 7.7



égalité as< aléatoire

Définition 7.4 Relation d'asynchronisme avec respect des valeurs

Soient TC_1 et TC_2 deux types de communication. TC_2 est dit plus asynchrone que TC_1 en respectant les valeurs; noté TC_1 op< TC_2 si et seulement si :

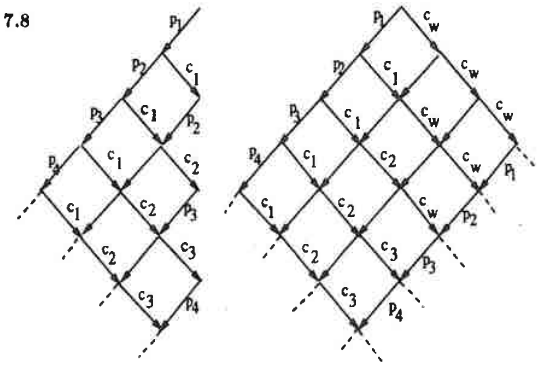
- *TC_2 est plus asynchrone que TC_1*
  et
- *pour chaque couple de termes (t_1, t_2) de $DEF(TC_1) \times DEF(TC_2)$ tel que $h(t_1) = t_2$, $indice_1(t_1) = indice_2(t_2)$*

Proposition 7.1

La relation op< est une relation d'ordre partiel sur l'ensemble des types de communication telle que :

$$TC_1 \ op< \ TC_2 \Longrightarrow TC_1 \ as< TC_2$$

Exemple 7.8



égalité op< égalité-presque-partout

5 Formalisation des relations entre les types de communication

Pour pouvoir automatiser la preuve des relations $as<$ et $op<$ entre les types de communication, nous allons les reformuler sous forme de relations entre les termes qui leurs sont associés.

5.1 Formalisation de la relation d'asynchronisme

Soit (t_1, t_2) un couple de termes de $DEF(T(C_1)) \times DEF(T(C_2))$ tels que $h(t_1) = t_2$.
Il est clair que le nombre d'opérations de production dans t_1 est égal au nombre d'opérations de production dans t_2. Nous pouvons constater la même chose pour le nombre d'opérations de consommation. nous pouvons donc déduire les hypothèses suivantes :

1.

$$\begin{array}{c} cardinal(ens\_prod_1(t_1)) = cardinal(ens\_prod_2(t_2)) \\ et \\ cardinal(ens\_cons_1(t_1)) = cardinal(ens\_cons_2(t_2)) \end{array}$$

Nous pouvons en déduire que :

2.

$$\begin{array}{|c|}\hline maximum(ens\_prod_1(t_1)) = maximum(ens\_prod_2(t_2)) \\ et \\ maximum(ens\_cons_1(t_1)) = maximum(ens\_cons_2(t_2)) \\ \hline\end{array}$$

car $cardinal(A) = maximum(A) - minimum(A) + 1$ et
$minimum(ens\_prod(t_i)) = minimum(ens\_cons(t_i)) = 1$

Maintenant pour savoir si un type de communication TC_2 est plus asynchrone qu'un autre type TC_1, il faut tester si l'image homomorphe d'un terme défini de $T(C_1)$ est un terme défini de $T(C_2)$. En d'autres termes il faut montrer que si les préconditions sont vérifiées sur un terme t_1 de $T(C_1)$ alors elles sont vérifiées sur son image homomorphe $t_2 = h(t_1)$. Or la seule précondition qui existe sur les types de communication est celle sur les opérations de consommation (pre_cons). La relation d'asynchronisme entre deux types de communication se définit alors comme suit :

Proposition 7.2

Un type de communication TC_2 est plus asynchrone qu'un type de communication TC_1 si et seulement si :
pour tout couple de termes définis (t_1, t_2) de $DEF(T(C_1)) \times DEF(T(C_2))$ tel que $h(t_1) = t_2$

$$pre\_cons_1(t_1) == vrai \supset pre\_cons_2(t_2) == vrai$$

5.2 Formalisation de la relation d'asynchronisme avec perte des valeurs

Il reste maintenant à formaliser la relation entre les indices.

- L'indice associé à un terme avec une opération de production en tête est égal à l'indice dans l'ensemble produit ens_prod de l'élément résultant de l'occurrence de production associée. Or d'après l'équation 11 de la définition d'un type de communication cet indice est égal à

  $$maximum(ens\_prod(tc)) + 1.$$

  Soit maintenant (t_1, t_2) un couple de termes définis de $DEF(T(C_1)) \times DEF(T(C_2))$ tels que $t_2 = h(t_1)$ et $t_1 = produire(t'_1, v)$.
  Pour comparer les indices de ces deux termes il faut comparer

  $$maximum(ens\_prod_1(t_1)) \text{ et } maximum(ens\_prod_2(t_2))$$

  qui sont toujours égaux d'après l'hypothèse 2 ci-dessus.

- L'indice associé à un terme avec une opération de consommation en tête est égal à l'indice dans l'ensemble consommable de l'élément résultant de l'occurrence de consommation associée. Qui est par définition égal à $ind\_cons(tc)$.
  Soit maintenant (t_1, t_2) un couple de termes définis de $DEF(T(C_1)) \times DEF(T(C_2))$ tels que $t_2 = h(t_1)$ et $t_1 = consommer(t'_1)$.
  pour comparer les indices sur ces deux termes quand les consommations sont définies il faut comparer

  $$ind\_cons_1(t) \text{ et } ind\_cons_2(t_2)$$

La propositions suivante permet de donner une définition formelle de la relation $as <$ entre deux types de communication.

Proposition 7.3

Un type de communication TC_2 est plus asynchrone qu'un type de communication TC_1 avec possibilité de perte de valeurs (TC_1 $as <$ TC_2) si et seulement si :
pour tout couple de termes définis (tc_1, tc_2) de $DEF(T(C_1)) \times DEF(T(C_2))$ tel que $h(tc_1) = tc_2$,

$$\begin{array}{|l|}\hline pre\_cons_1(tc_1) == vrai \supset pre\_cons_2(tc_2) == vrai \\ \qquad\qquad \wedge \\ pre\_cons_1(tc_1) == vrai \wedge pre\_cons_2(tc_2) == vrai \\ \qquad\qquad\qquad\qquad \supset ind\_cons1(tc_1) \leq ind\_cons(tc_2) == vrai \\ \hline\end{array}$$

5.3 Formalisation de la relation d'asynchronisme avec respect des valeurs

Par analogie avec ce qui a été fait dans le paragraphe précédent, la proposition suivante permet de donner une définition formelle de la relation $op <$ entre deux types de communication.

Proposition 7.4

Un type de communication TC_2 est plus asynchrone qu'un type de communication TC_2 avec respect des valeurs (TC_1 $op <$ TC_2) si et seulement si :
pour tout couple de termes définis (tc_1, tc_2) de $T(C_1) \times T(C_2)$ tel que $h(tc_1) = tc_2$,

$$\begin{array}{|l|}\hline pre\_cons_1(tc_1) == vrai \supset pre\_cons_2(tc_2) == vrai \\ \qquad\qquad \wedge \\ pre\_cons_1(tc_1) == vrai \wedge pre\_cons_2(tc_2) == vrai \\ \qquad\qquad\qquad\qquad \supset ind\_cons1(tc_1) == ind\_cons(tc_2) \\ \hline\end{array}$$

Remarque 7.5

S'il y avait des precondition sur les productions que nous aurions notée pre_prod, la propriété qu'il faudrait vérifier pour que $TC_1 op < TC_2$ est :
pour tout couple de termes définis (tc_1, tc_2) de $T(C_1) \times T(C_2)$ tel que $h(tc_1) = tc_2$

$$\begin{array}{|l|}\hline pre\_cons_1(tc_1) == vrai \supset pre\_cons_2(tc_2) == vrai \\ \qquad\qquad \wedge \\ pre\_prod_1(tc_1) == vrai \supset pre\_prod_2(tc_2) == vrai \\ \qquad\qquad \wedge \\ pre\_cons_1(tc_1) == vrai \wedge pre\_cons_2(tc_2) == vrai \\ \qquad\qquad\qquad\qquad \supset ind\_cons1(tc_1) == ind\_cons(tc_2) \\ \hline\end{array}$$

6 Preuve des relations d'asynchronisme

6.1 Environnement de la preuve

Nous avons vu dans le paragraphe 2 que les types de communication peuvent être spécifiés par un ensemble d'équations ce qui implique qu'ils peuvent s'adapter aux techniques de preuve que nous avons définies dans les chapitres précédents.
L'environnement de la preuve d'une relation d'asynchronisme entre deux types de communication sera donc constitué :

- de la spécification des deux types,
- de la spécification des types *Table* et *Ensemble ordonné*, définis dans l'exemple 4.5, puisque la spécification des types de communication importe certaines de leurs opérations et
- de la spécification des types *Entier* et *booléen*.

Le paragraphe précédent a permis de calculer la formule équationnelle correspondante à chaque relation d'asynchronisme.

6.2 Schéma d'induction

Il est clair que nous aurons besoin d'une induction pour prouver ces formules équationnelles. Nous allons donc faire une récurrence sur les constructeurs respectifs des types de communication TC_1 et TC_2; et qui sont, rappelons le, $\{init\_com_1, produire_1, consommer_1\}$ et $\{init\_com_2, produire_2, consommer_2\}$.
Pour calculer le schéma d'induction associé à cette récurrence nous allons faire appel aux notions que nous avons vu dans le chapitre 3 sur les principes de l'induction structurelle.
Ainsi pour calculer un schéma d'induction associé à une liste de sortes ou de types il faut calculer l'ensembles des termes générateurs associés : c'est l'ensemble de toutes les combinaisons dans l'ordre de tous les constructeurs associés à chaque type; puis calculer l'ensembles des prédécesseurs de chacun d'eux.
Le schéma d'induction associé à deux types de communication ne peut pas être généré automatiquement à cause de certaines contraintes que nous allons découvrir. Cependant nous allons suivre la même démarche que le processus de génération automatique.
Les termes générateurs associés à ces ensembles de constructeurs sont :

1. $init\_com_1, init\_com_2$,
2. $init\_com_1, produire_2(tc_2, v)$,
3. $init\_com_1, consommer_2(tc_2)$,
4. $produire_1(tc_1, v), init\_com_2$,
5. $produire_1(tc_1, v), produire_2(tc_2, v)$,
6. $produire_1(tc_1, v), consommer_2(tc_2)$,
7. $consommer_1(tc_1), init\_com_2$,

8. $consommer_1(tc_1), produire_2(tc_2, v)$,
9. $consommer_1(tc_1), consommer_2(tc_2)$.

Mais il faut que chaque terme générateur t_1, t_2 vérifie la condition $h(t_1) = t_2$. Cela n'est pas possible pour les termes $2, 3, 4, 6, 7, 8$.
Donc les termes générateurs restants sont :

- $init\_com_1, init\_com_2$,
- $produire_1(tc_1, v), produire_2(tc_2, v)$,
- $consommer_1(tc_1), consommer_2(tc_2)$.

Les prédécesseurs du terme $consommer_1(tc_1), consommer_2(tc_2)$. sont :

1. tc_1, t; où t et un terme de type TC_2. Mais il faut que la condition $h(tc_1) = t$ soit vérifiée. Ce qui implique que $t = tc_2$ et que le prédécesseur est tc_1, tc_2.
2. $consommer_1(tc_1), tc_2$ qui ne peut pas vérifier la condition $h(consommer_1(tc_1)) = tc_2$ sera donc exclu.

La même chose pour les prédécesseurs de $produire_1(tc_1, v), produire_2(tc_2, v)$. En plus il faut aussi que ces termes soit définis; donc pour le terme générateur $consommer_1(tc_1), consommer_2(tc_2)$ il faut supposer les préconditions $pre\_cons(tc_1) == vrai \wedge pre\_cons(tc_2) == vrai$. Nous obtenons donc le schéma d'induction suivant :

$$\frac{\begin{array}{c} P(init\_com_1, init\_com_2) \wedge \\ P(tc_1, tc_2) \supset P(produire_1(tc_1, v), produire_2(tc_2, v)) \wedge \\ (P(tc_1, tc_2) \wedge pre\_cons_1(tc_1) \wedge pre\_cons_2(tc_2)) \supset P(consommer_1(tc_1), consommer_2(tc_2)) \end{array}}{P(t_1, t_2)}$$

Remarque 7.6

Ce schéma d'induction est en accord avec le but pour le quel a été introduit l'homomorphisme entre les termes définis respectives de TC_1 et de TC_2; c'est à dire que ne nous intéressons qu'aux séquences d'exécutions équivalentes. En effet, l'induction de base se fait sur deux séquences équivalentes $init\_com_1$ et $init\_com_2$. Et si nous supposons que tc_1 et tc_2 sont des séquences équivalentes alors $produire_1(tc_1, v)$ et $produire_2(tc_2, v)$ le sont aussi. La même chose pour $consommer(tc_1)$ et $consommer(tc_2)$

6.3 Déroulement de la preuve

Le déroulement de la preuve se fait normalement comme nous l'avons décrit dans le chapitre précédent, à savoir la substitution dans le schéma d'induction de la formule à prouver, la décomposition de la formule obtenue en un ensemble de a_équations, la réduction des assertions puis la réduction de la conclusion pour chacune de ces a_équations puis si c'est nécessaire l'utilisation du chaînage transitif.

Exemple 7.9

L'annexe C contient l'arbre de preuve de la relation d'asynchronisme avec perte des valeurs entre le type de communication égalité *et* aléatoire *:* égalité $as<$ aléatoire.

7 Application à la modification des programmes parallèles

Rappelons que le parallélisme que nous voulons introduire au moment de la résolution d'un problème est un parallélisme de conception qui a été introduit pour des raisons d'efficacité du programme obtenu. Ce parallélisme se traduit par des processus indépendants qui communiquent entre eux en utilisant la notion de types de communication. Cela implique qu'un moyen de réduire le temps d'exécution d'un tel programme est de réduire le temps d'attente des processus communicants. En d'autres termes il faut que l'exécution soit le *plus asynchrone* possible au sens des relations d'ordre ci-dessus.

Dans ce contexte, vouloir modifier un programme donné en un autre programme plus asynchrone revient à agir sur les communications entre les processus de celui-ci en remplaçant un type de communication par un autre tels qu'ils vérifient une des relations précédemment définies.

Cette modification va entrainer une modification des séquences d'opérations exécutées par ces programmes. Le programme obtenu peut avoir une sémantique différente du programme initial. Il ne s'agit donc pas des techniques habituelles de transformation de programme préservant la sémantique (fusion, pliage, dépliage, ...). Nous parlerons donc de *modification* des programmes.

7.1 Modification utilisant la relation $as<$

Dans un programme et pour une donnée communiquée, lorsque nous remplaçons un type de communication TC_1 par un autre type de communication TC_2 tel que TC_1 $as<$ TC_2, sans changer les opérations dans les processus, nous obtenons un programme plus asynchrone dans le sens où

- toutes les valeurs produites ne sont pas consommées (les indices de consommation dans TC_1 sont inférieurs à ceux dans TC_2),
- Les situations d'attente du processus consommateurs sont plus rares (la précondition de consommation dans TC_2 est plus faible que celle dans TC_1).

Le programme ainsi obtenu n'est plus défini sur les même données. C'est pourquoi il faut faire attention lors de cette modification que la post-condition du programme reste vérifiée.

Le passage des communications synchrones à des communications plus asynchrones a été aussi étudié dans [Gri88].

Exemple 7.10 Calcul de factorielle

Soit n un entier strictement positif. Le nombre $n!$ peut être calculé de deux manières différentes :

- *Il peut être considéré comme le terme de rang n de la suite f définie par :*
  $f_0 = 1$
  $f_{i+1} = f_i \times (i+1)\ \forall\ i > 0$
  Le terme de rang i est tel que $f_i = i!$ pour tout i supérieur ou égal à 0.

- *Il peut être considéré comme le terme de rang n de la suite g définie par :*
  $g_0 = n$
  $g_{i+1} = g_i \times (n-i)\ \forall i > 0$
  Le terme de rang i est tel que $g_i = n!/(n-i)!$ pour tout i supérieur ou égal à 0.

Nous pouvons utiliser ces deux modes de calcul pour construire une solution parallèle constituée de deux processus indépendants qui calculent respectivement les termes des deux suites f et g sur deux intervalles complémentaires. Le résultat est le produit des résultats des deux processus.

La répartition des calculs entre ces deux processus doit se faire de telle sorte que

- *Le premier calcule les termes de la suite f jusqu'à $n/2^{\text{ième}}$ terme. Le résultat de ce processus est $(n/2)!$.*
- *Le deuxième calcule les termes de la suite g jusqu'à $n/2^{\text{ième}}$ terme. Le résultat de ce processus est $n!/(n-n/2)!$.*

Le résultat final est $(n/2)! \times n!/(n-n/2)!$ qui est égal à $n!$.

Cet algorithme peut être généralisé de la manière suivante :

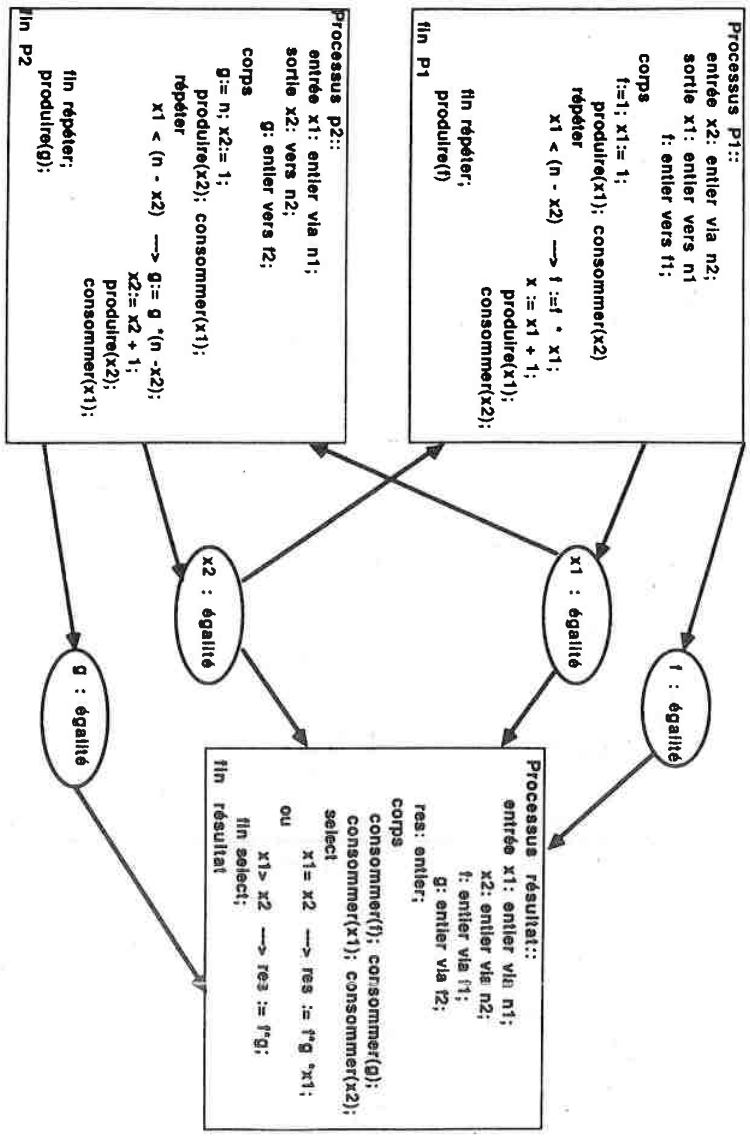
- *Le premier processus p_1, calcule les termes de la suite f dans un intervalle $[0..n_1]$.*
- *Le deuxième processus p_1, calcule les termes de la suite g dans un intervalle $[0..n_2]$. La condition d'arrêt est $n_1 \geq n - n_2$.*

Pour évaluer cette condition d'arrêt, il faut que ces deux processus P_1 et P_2 échangent les valeurs successives de n_1 et n_2. Supposons que n_1 est la suite des valeurs produites par P_1 et susceptibles d'être consommées par P_2 et inversement pour n_2. La relation de communication qui vient immédiatement à l'esprit est celle qui consiste à ce que chaque processus consomme chaque valeur produite par l'autre et dans cet ordre de production. Cette relation correspond au type de communication égalité. *Nous obtenons donc une solution synchrone définie par le programme suivant :*

Programme *fact* ::
  port n_1, n_2 : *égalité;*
    f_1, f_2 : *égalité;*
  processus P_1 ::
    entrée x_2: *entier* **via** n_2;
    sortie x_1: *entier* **vers** n_1,
      f: *entier* **vers** f_1;
    corps
      $f := 1$; $x_1 := 1$;
      produire(x_1); **consommer**(x_2);
      répéter
        $x_1 < (n - x_2)$ $\longrightarrow f := f \times x_1$;
                $x_1 := x_1 + 1$;
                produire(x_1); **consommer**(x_2);
      fin répéter;
      produire(f)
    fin P_1
  processus P_2 ::

entrée x_1*: entier* **via** n_1*;*
sortiex_2*: entier* **vers** n_2*,*
g: entier **vers** f_2*;*
corps
$g := n; x_2 := 1;$
produire(x_2); **consommer**(x_1);
répéter
$x_1 < (n - x_2) \longrightarrow g := g \times (n - x_2);$
$x_2 := x_2 + 1;$
produire(x_2);
consommer(x_1);
fin répéter;
produire(g)
fin P_2
processus *résultat::*
entrée x_1*: entier* **via** n_1*,*
x_2*: entier* **via** n_2*,*
f*: entier* **via** f_1*,*
g*: entier* **via** f_2*;*
res: entier;
corps
consommer(f); **consommer**(g);
consommer(x_1); **consommer**(x_2);
select
$x_1 = x_2 \longrightarrow res := f \times g \times x_1;$
ou
$x_1 > x_2 \longrightarrow res := f \times g;$
fin select
fin P_2
composition
$P_1 // P_2 // résultat$
fin *fact*

Le schéma suivant permet de bien visualiser les communications dans ce programme.



La post condition de ce programme est :

$$(1) \quad f = x_1! \text{ et } g = n!/(n-x_2)! \text{ et } (n-x_2) \leq x_1 \leq (n-x_2+1)$$

Ce programme donne une solution parallèle synchrone du calcul du factorielle, dans le sens où toutes les valeurs produites sont consommées avec les attentes que ça peut engendrer. Or chacun des processus P_1 ou P_2 n'a pas besoin dans son calcul de toutes les valeurs produites par l'autre. En effet pour évaluer la condition d'arrêt chacun des deux processus n'a besoin que de la dernière valeur émise par l'autre. Les autres valeurs peuvent être ignorées sans incidence sur les calculs. Le type de communication aléatoire *satisfait bien cette relation de communication. Et comme* égalité $<_{as}$ aléatoire, *nous pouvons remplacer le premier type par le deuxième dans le programme ci-dessus pour obtenir une solution plus asynchrone. Pour cela il suffit de remplacer les déclarations :*

- n_1*: égalité par* n_1 *: aléatoire*
  et
- n_2*: égalité par* n_2 *: aléatoire*

On montre dans [JP85] que le programme obtenu satisfait la post condition (1).

8 Conclusion

L'objectif de ce chapitre était de montrer un exemple de l'utilité de notre système de preuve dans un environnement de conception de programmes parallèles. Il faut signaler que les qualités de cet environnement, tel que la modularité, ont permis à notre système de preuve de jouer un rôle important dans les étapes de la conception. En effet, il contribue à côté du système *COMEDIE* [JM86] et *COMEDIE-ADA* [KFJP88] à la conception d'une bibliothèque de types de communication en établissant les relations d'ordre qui peuvent exister entre eux. Le rôle de notre système sera de localiser la place d'un nouveau type de communication dans le treillis formé par les types déjà ordonnés. Cette bibliothèque sera ensuite utilisée pour modifier les programmes.

Cet outil de preuve automatique peut aussi être utilisé pour prouver d'autres propriétés sur ces programmes, comme par exemple le fait qu'un type de communication est un organe passif qui ne modifie pas les valeurs des données communiquées.

Conclusion

Tout au long de ce travail, nous nous sommes efforcé d'utiliser des techniques simples combinées entre elles pour pouvoir étendre la classe de propriétés élémentaires que nous pouvons prouver.

Nous avons vu tout d'abord que les définitions formelles de validité peuvent être étendues sans problème aux cas des formules équationnelles.

La méthode de preuve par induction nous a permis de donner une solution pratique au problème de validité inductive. Nous avons vu que les schémas d'induction pouvaient être générés automatiquement dans le cas où les propriétés à prouver ne contenaient que des opérations définies recursivement à partir des constructeurs.

Nous avons vu que pour étendre le mécanisme de remplacement de termes par leurs égaux aux formules équationnelles, il fallait choisir quelques équations représentatives telles que leur validité soient équivalente à celle de la formule associée. Pour pouvoir effectuer ce choix, il fallait décomposer la formule équationnelle en un ensemble de a_équations. Les équations représentatives sont constituées à partir des parties conclusion respectives de chaque a_équation. Cela vient du fait que la validité d'une a_équation dans un ensemble d'équations est équivalente à la validité de sa partie conclusion dans l'ensemble d'équations initial plus l'ensemble des assertions. Selon ce critère de décomposition en un ensemble de a_équations, nous avons pu déterminer la classe de formules équationnelles susceptibles d'être prouvées en utilisant cette méthode.

Pour la mise en œuvre de cette approche nous avons défini la réécriture conditionnelle sur les a_termes, ainsi que le raisonnement par cas en présence des équations conditionnelles de la forme *si-alors-sinon*.

Nous avons vu que dans ce contexte, la seule propriété globale exigible pour l'utilisation de la réécriture, était la propriété de terminaison. C'est à la fois un avantage et un inconvénient. C'est un avantage puisque l'utilisateur n'a pas beaucoup de restrictions lors de la définition de sa spécification, et puisque jusqu'à maintenant il n'existe pas de méthode générale pour prouver la confluence ou la complétude sans faire de restriction sur les spécifications. C'est un inconvénient car pour que la méthode soit complète, il faut que pour chaque a_équation, l'ensemble des équations initial plus l'ensemble des assertions forment un système de réécriture confluent. Pour contourner cet inconvénient, la réduction des assertions permet de détecter les inconsistances quand elles existent, sinon leurs formes normales sont utilisées, elles aussi, pour réduire la partie conclusion.

Par ailleurs, nous avons vu que le chaînage transitif, simple ou conditionnel, avec simplification, était un outil en plus de la réécriture pour prouver la validité des équations contenant des relations d'ordre.

Tous les résultats ci-dessus ont été implantés dans le système de preuve *PAUSE*. Nous nous sommes efforcé de lui donner une interface agréable pour permettre à l'utilisateur de

comprendre le déroulement de la preuve; et dans le cas ou elle échoue de lui fournir les outils pour lui permettre de voir pourquoi et de corriger sa spécification ou de reformuler sa propriété à prouver.

Pour appliquer notre méthode de preuve à un problème donné, qu'il soit séquentiel ou parallèle, il faut regarder si un tel problème peut être modélisé par une spécification algébrique, ensuite il faut regarder si la propriété à prouver peut être modéliser avec une formule équationnelle. Le chapitre 7 a permis de donner un exemple d'application dans le cadre de la programmation parallèle.
Il s'agissait de prouver des relations d'asynchronisme entre les modules qui s'occupent de la communication dans un système parallèle. Nous avons vu que ces relations sont utilisées pour appliquer certaines modifications sur les systèmes parallèles.

Une perspective envisageable à ce travail est d'étendre plus la classe de formules prouvables. En effet, nous avons vu dans le chapitre 2 que la contrainte principale à cette extension est la présence de négations d'équations dans une a_équation.
Pour lever cette contrainte il faut essayer de donner une sémantique aux spécifications contenant des négations d'équations, puis une méthode de preuve associée; cela nous permettra de résoudre le problème de présence d'une négation d'une équation dans la partie assertion. Nous devons aussi trouver une méthode générale pour prouver la validité de la négation d'une équation dans une spécification; cela nous permettra de résoudre problème de présence d'une négation d'une équation dans la partie conclusion. L'extension des travaux sur l'unification et la disunification modulo un ensemble d'équations quelconque [CL88,Bur88] est une solution possible à ce problème.

Nous pourrons aussi envisager d'étudier les propriétés d'autres programmes parallèles écrits en d'autres langages d'expression du parallélisme et de la communication qui pourront s'adapter au raisonnement équationnel.

Partie IV

Annexes

Annexe A

Spécification des entiers

```
1 : x = x == vrai.
2 : x < x == faux.
3 : x =< x == vrai.
4 : 0 + x == x.
5 : x + 0 == x.
6 : (x + ( y + z )) == (x + y + z).
7 : ((x + y) + z) == (x + y + z).
8 : ((x - y) + z) == (x + z - y).
9 : ((x + y) - z) == (x + y -z) .
10 : (x + ( y - z)) == (x + y - z).
11 : (x - (y - z)) == (x + z - y) .
12 : (x - (y + z)) == (x - y - z) .
13 : (x + y + x) == (x + x + y).
14 : x < (y + x ) == 0 < y .
15 : x + y < y + x == faux .
16 : x + y < z + x == y < z .
17 : x + y + z < k + y == x + z < k.
18 : x + y < z + y + k == x < z + k.
19 : (x + y - y) == x.
20 : (x - x + y) == y.
21 : (x + y - x) == y.
22 : (x + y + z - k - y) == (x + z - k).
23 : (x + y) =< (z + y - k) == x =< z -k.
24 : (x - y) < z == x < (z + y).
25 : (x - y + z ) < k == (x + z) < (k + y) .
26 : (x - y) =< z == x =< (z + y).
27 : (x - y + z ) =< k == (x + z) =< (k + y) .
28 : (x - y + z) == (x + z - y) .
29 : x < (y - z) == (x + z) < y .
30 : x =< (y - z) == (x + z) =< y .
31 : x < 0 == faux
32 : (x + y) =< (x + z) == y =< z.
33 : x =< (y + x ) == 0 =< y .
34 : x + y + z < k + y == x + z < k.
```

```
35 : x + y < z + y + k == x < z + k.
36 : (x + y - y) == x.
37 : (x - x + y) == y.
38 : (x + y - x) == y.
39 : (x + y + z - k - y) == (x + z - k).
40 : (x + y) =< (z + y - k) == x =< z -k.
41 : (x - y) < z == x < (z + y).
42 : (x - y + z ) < k == (x + z) < (k + y) .
43 : (x - y) =< z == x =< (z + y).
44 : (x - y + z ) =< k == (x + z) =< (k + y) .
45 : (x - y + z) == (x + z - y) .
47 : x < (y - z) == (x + z) < y .
48 : x =< (y - z) == (x + z) =< y .
49 : x < 0 == faux.
50 : (x + y) =< (x + z) == y =< z.
51 : (x + y) = (z + y) == x = z.
52 : (x + y) < (x + z) == y < z .
53 : (y + x) =< (z + x) == y =< z .
54 : (y + x) < (z + x) == y < z.
55 : x < (x + y) == vrai.
57 : (x + y) > x == vrai.
58 : (x + y) >= y == vrai .
59 : (y + x) >= y == vrai .
60 : x =< (y + x) == vrai.
61 : x =< (x + y) == vrai.
62 : (x - y) < x == vrai.
63 : (x - y) =< x == vrai.
64 : (x - x) == 0 .
65 : x + y =< z + x == y =< z .
66 : x < (y + x) == 0 < y .
67 : (x + y) =< x == y =< 0.
68 : (x + y) < x == y < 0 .
69 : (y + x) < x == x < 0 .
70 : 0 =< x == vrai.
71 : x < 0 == faux.
72 : x >= 0 == vrai.
73 : 0 > x == faux.
74 : 1000 > x  == vrai.
75 : x >= 1000 == faux.
76 : x < 1000 == vrai.
77 : 1000 =< x == faux
faxomes
```


Annexe B

Syntaxe concrète du langage d'expression du parallélisme

La déscription de cette syntaxe est donnée par une grammaire à contexte libre. L'axiome initial est SYSTEME; les non terminaux sont en majuscules, les mots clés du langage sont en gras. La notation $X\|Y$ désigne une alternance entre X et Y. La notation $\{X\}^*$ désigne une répétition de 0 ou n fois. Les autres symboles sont des symboles du langage.

SYSTEME ⟶ **programme** IDENT :: SYST
IDENT est le nom du programme

SYST ⟶ {COMMUNICATION}* PROC-COMMIC
{PROC-COMMUNIC}* COMPOSITION

PROC-COMMUNIC ⟶ **processus** IDENT :: PROCESS
‖ **modèle processus** IDENT L-PARAM$^{0/1}$ :: PROCESS
déclaration d'un processus ou d'un modèle (éventuellement paramétré)
de processus. Toutes les instances de processus *sont activé*
simultanément de manière statique dans la partie composition

COMMUNICATION ⟶ {**typcom** ID-TYPCOM(elt) :: DESCR-TYPCOM}*
port LISTE-PORTS; {LISTE-PORTS;}*
ID-TYPCOM est le nom du type de communication
elt est le type de la donnée communiqué
DESCR-TYPCOM est la définition algébrique d'un type de communication

PROCESS ⟶ PROC-SEQ ‖ SYST

PROC-SEQ ⟶ INTERFACE DECLARATION CORPS

COMPOSITION ⟶ **composition** (ID-PROC)$^{0/1}$ {// {ID-PROC ‖ VECT-ID-PROC}}*
fin IDENT;
création et activation parallèle de processus ou d'un vecteur
d'instances d'un modèle de processus précédemment défini
exemple : $P_1//P_2//(i:1..3, C_i :: cons)$
$C//(i:1..5 P_i :: PROD(p_i))$
p_i est le nom de port précédemment déclaré

L-PARAM ⟶ (PARAM {, PARAM}*)

PARAM ⟶ ID-PARAM : ID-TYPE

ID-PARAM est le nom du paramètre formel
ID-TYPE est le type du paramètre

ID-PROC ⟶ IDENT ‖ IDENT1 :: IDENT2 L-ARGT$^{0/1}$
IDENT est le nom d'un processus précédemment défini
IDENT1 est le nom d'un processus instance du modèle
IDENT2 pour les paramètres effectifs contenus dans L-ARGT
exemple PROD
C :: CONS(x,y) où x et y sont des noms de port

VECT-ID-PROC ⟶ indice: 1 .. nb, IDENTindice :: IDENT2(L-ARGT)$^{0/1}$
vecteur de nb processus de modèle IDENT2
exemple (// i : 1..3, Pi :: PP)

L-ARGT ⟶ (ID {,ID}*)
ID est le nom de variable ou de port ou constante

INTERFACE ⟶ ENTREE ‖ SORTIE ‖ ENTREE SORTIE

CORPS ⟶ **corps** ENONCES **FIN** ident;

ENTREE ⟶ **entrée** LISTE-ENTREES;

SORTIE ⟶ **sortie** LISTE-SORTIES;

LISTE-PORTS ⟶ LISTE-ID-PORT : ID-TYPCOM(ID-TYPE);
ID-PORT est le nom de port
ID-TYPE est le nom du type des données communiquées par le port

LISTE-ID-PORTS ⟶ ID-PORT{,{ID-PORT}* ‖ {ID-PORTindice, indice 1.. nb}}*

LISTE-ENTREES ⟶ ID-DONNEE : TYPE **via** ID-PORT;
{ID-DONNEE :TYPE **via** ID-PORT;}*
ID-DONNEE est le nom de donnée communiquée

LISTE-SORTIES ⟶ ID-DONNEE : TYPE **vers** LISTE-ID-PORTS;
{ID-DONNEE :TYPE **vers** LISTE-ID-PORTS;}*
une donnée peut être produite vers plusieurs ports
les liste d'entrées et de sorties sont disjointes

DECLARATION ⟶ *déclaration classique de types, variables simples, structures ...*

ENONCES ⟶ ENONCE ; {ENONCE}*
mise en séquence d'énoncés par ;

ENONCE ⟶ VIDE ‖ AFFECTATION ‖ OP-COMMUNICAT ‖
CHOIX-INDETER ‖ conditionnelle et itérations classiques

VIDE ⟶

AFFECTATION ⟶ ID := EXP

OP-COMMUNICAT ⟶ **produire**(ID-DONNEE) ‖ **consommer**(ID-DONNEE)

CHOIX-INDETER ⟶ **select** GARDE ⟶ ENONCES
{**OU** GARDE ⟶ ENONCES}* **fin select**

GARDE ⟶ EXP-BOOL

Annexe C

Exemple de preuve de relation entre les types de communication

Il s'agit de démontrer que *égalité as< aléatoire.*
Les opérations du type *égalité* sont terminées par _e, ceux du type *aléatoire* sont terminées par _a.

```
Theoreme a demontre par induction :
(
  (
    pre_cons_e(tc1)
       =>
    pre_cons_a(tc2)
  )
     et
  (
    (
      pre_cons_e(tc1)
         et
      pre_cons_a(tc2)
    )
       =>
    (ind_cons_e(tc1) =< ind_cons_a(tc2))
  )
)


Le schema d'induction est

(
  P( init_com_e, init_com_a )
     et
  (
    (
      P( tc3, tc4 )
         =>
      P( produire_e(tc3 , v), produire_a(tc4 , v) )
    )
       et
    (
```

```
      (
        P( tc3, tc4 )
           et
        (
          pre_cons_e(tc3)
             et
          pre_cons_a(tc4)
        )
      )
        =>
      P( consommer_e(tc3), consommer_a(tc4) )
    )
  )
)
```

Generation de noeuds :

```
***   noeud numero 0
***   assertions
***   conclusion
(
  (
    pre_cons_e(tc1)
      =>
    pre_cons_a(tc2)
  )
     et
  (
    (
      pre_cons_e(tc1)
         et
      pre_cons_a(tc2)
    )
       =>
    (ind_cons_e(tc1) =< ind_cons_a(tc2))
  )
)


***   noeud numero 1
***   assertions
***   conclusion
(
  pre_cons_e(tc1)
     =>
  pre_cons_a(tc2)
)


***   noeud numero 1
***   assertions
```



```
***   conclusion
(
  (
    pre_cons_e(init_com_e)
       =>
    pre_cons_a(init_com_a)
  )
     et
  (
    (
      (
        pre_cons_e(tc3)
           =>
        pre_cons_a(tc4)
      )
         =>
      (
        pre_cons_e(produire_e(tc3 , v))
           =>
        pre_cons_a(produire_a(tc4 , v))
      )
    )
       et
    (
      (
        (
          pre_cons_e(tc3)
             =>
          pre_cons_a(tc4)
        )
           et
        (
          pre_cons_e(tc3)
             et
          pre_cons_a(tc4)
        )
      )
        =>
      (
        pre_cons_e(consommer_e(tc3))
           =>
        pre_cons_a(consommer_a(tc4))
      )
    )
  )
)


***   noeud numero 2
***   assertions
***   conclusion
(
  (
    pre_cons_e(tc1)
```

```
      et
    pre_cons_a(tc2)
  )
    =>
  (ind_cons_e(tc1) =< ind_cons_a(tc2))
)


***   noeud numero 2
***   assertions
***   conclusion
(
  (
    (
      pre_cons_e(init_com_e)
         et
      pre_cons_a(init_com_a)
    )
     =>
    (ind_cons_e(init_com_e) =< ind_cons_a(init_com_a))
  )
   et
  (
    (
      (
        (
          pre_cons_e(tc3)
             et
          pre_cons_a(tc4)
        )
          =>
        (ind_cons_e(tc3) =< ind_cons_a(tc4))
      )
        =>
      (
        (
          pre_cons_e(produire_e(tc3 , v))
             et
          pre_cons_a(produire_a(tc4 , v))
        )
          =>
        (ind_cons_e(produire_e(tc3 , v)) =< ind_cons_a(produire_a(tc4 , v)))
      )
    )
     et
    (
      (
        (
          (
            pre_cons_e(tc3)
               et
            pre_cons_a(tc4)
          )
            =>
```



```
          (ind_cons_e(tc3) =< ind_cons_a(tc4))
        )
          et
        (
          pre_cons_e(tc3)
             et
          pre_cons_a(tc4)
        )
      )
       =>
      (
        (
          pre_cons_e(consommer_e(tc3))
             et
          pre_cons_a(consommer_a(tc4))
        )
          =>
        (ind_cons_e(consommer_e(tc3)) =< ind_cons_a(consommer_a(tc4)))
      )
    )
  )
)

Debut de la preuve

***   noeud numero 11
***   assertions
pre_cons_e(init_com_e) ==  vrai
***   conclusion
pre_cons_a(init_com_a)


***   noeud numero 111
***   assertions
pre_cons_e(init_com_e) ==  vrai
pre_cons_e(init_com_e) ==  faux
***   conclusion
Theoreme demontre par suite d'hypotheses incoherentes


***   noeud numero 12
***   assertions
pre_cons_e(produire_e(tc3 , v)) ==  vrai
pre_cons_e(tc3) ==  vrai
pre_cons_a(tc4) ==  vrai
***   conclusion
pre_cons_a(produire_a(tc4 , v))


***   noeud numero 121
***   assertions
pre_cons_a(tc4) ==  vrai
pre_cons_e(tc3) ==  vrai
pre_cons_e(produire_e(tc3 , v)) ==  vrai
```

```
estvide(domaine(consommable_e(tc3))) ==  faux
non(estvide(domaine(consommable_e(tc3)))) ==  vrai
non(estvide(domaine(consommable_a(tc4)))) ==  vrai
estvide(domaine(consommable_a(tc4))) ==  faux
***   conclusion
Le theoreme est demontre dans ce cas


***   noeud numero 13
***   assertions
pre_cons_e(produire_e(tc3 , v)) ==  vrai
pre_cons_e(tc3) ==  faux
***   conclusion
pre_cons_a(produire_a(tc4 , v))


***   noeud numero 131
***   assertions
pre_cons_e(tc3) ==  faux
pre_cons_e(produire_e(tc3 , v)) ==  vrai
non(estvide(domaine(consommable_e(tc3)))) ==  faux
estvide(domaine(consommable_e(tc3))) ==  vrai
non(vrai) ==  faux
***   conclusion
Le theoreme est demontre dans ce cas


***   noeud numero 14
***   assertions
pre_cons_e(consommer_e(tc3)) ==  vrai
pre_cons_e(tc3) ==  vrai
pre_cons_a(tc4) ==  vrai
***   conclusion
pre_cons_a(consommer_a(tc4))


***   noeud numero 141
***   assertions
pre_cons_a(tc4) ==  vrai
pre_cons_e(tc3) ==  vrai
pre_cons_e(consommer_e(tc3)) ==  vrai
estvide(domaine(enlever(consommable_e(tc3),
 min(domaine(consommable_e(tc3)))))) ==  faux
non(estvide(domaine(enlever(consommable_e(tc3),
 min(domaine(consommable_e(tc3))))))) ==  vrai
estvide(domaine(consommable_e(tc3))) ==  faux
non(estvide(domaine(consommable_e(tc3)))) ==  vrai
non(estvide(domaine(consommable_a(tc4)))) ==  vrai
estvide(domaine(consommable_a(tc4))) ==  faux
***   conclusion
Le theoreme est demontre dans ce cas

** Le noeud 1 est entierement prouve
```



```
***   noeud numero 21
***   assertions
pre_cons_e(init_com_e) ==  vrai
pre_cons_a(init_com_a) ==  vrai
***   conclusion
(ind_cons_e(init_com_e) =< ind_cons_a(init_com_a))


***   noeud numero 211
***   assertions
pre_cons_a(init_com_a) ==  vrai
pre_cons_e(init_com_e) ==  vrai
pre_cons_e(init_com_e) ==  faux
***   conclusion
Theoreme demontre par suite d'hypotheses incoherentes


***   noeud numero 22
***   assertions
pre_cons_a(produire_a(tc4 , v)) ==  vrai
pre_cons_e(produire_e(tc3 , v)) ==  vrai
pre_cons_a(tc4) ==  vrai
pre_cons_e(tc3) ==  vrai
ind_cons_e(tc3) =< ind_cons_a(tc4) ==  vrai
***   conclusion
(ind_cons_e(produire_e(tc3 , v)) =< ind_cons_a(produire_a(tc4 , v)))

***   noeud numero 221
***   assertions
ind_cons_e(tc3) =< ind_cons_a(tc4) ==  vrai
pre_cons_e(tc3) ==  vrai
pre_cons_a(tc4) ==  vrai
pre_cons_e(produire_e(tc3 , v)) ==  vrai
pre_cons_a(produire_a(tc4 , v)) ==  vrai
estvide(domaine(consommable_a(tc4))) ==  faux
non(estvide(domaine(consommable_a(tc4)))) ==  vrai
estvide(domaine(consommable_e(tc3))) ==  faux
non(estvide(domaine(consommable_e(tc3)))) ==  vrai
min(domaine(consommable_e(tc3))) =< max(domaine(ensprod_a(tc4))) ==  vrai
***   conclusion
Le theoreme est demontre dans ce cas


***   noeud numero 23
***   assertions
pre_cons_a(produire_a(tc4 , v)) ==  vrai
pre_cons_e(produire_e(tc3 , v)) ==  vrai
pre_cons_e(tc3) ==  faux
***   conclusion
(ind_cons_e(produire_e(tc3 , v)) =< ind_cons_a(produire_a(tc4 , v)))

***   noeud numero 231
```

```
***   assertions
pre_cons_e(tc3) ==  faux
pre_cons_e(produire_e(tc3 , v)) ==  vrai
pre_cons_a(produire_a(tc4 , v)) ==  vrai
non(estvide(domaine(consommable_e(tc3)))) ==  faux
estvide(domaine(consommable_e(tc3))) ==  vrai
non(vrai) ==  faux
***   conclusion
Le theoreme est demontre dans ce cas


***   noeud numero 24
***   assertions
pre_cons_a(produire_a(tc4 , v)) ==  vrai
pre_cons_e(produire_e(tc3 , v)) ==  vrai
pre_cons_a(tc4) ==  faux
***   conclusion
(ind_cons_e(produire_e(tc3 , v)) =< ind_cons_a(produire_a(tc4 , v)))

***   noeud numero 241
***   assertions
max(domaine(ensprod_a(tc4))) + 1 < min(domaine(consommable_e(tc3))) ==  vrai
max(domaine(ensprod_e(tc3))) + 1 < min(domaine(consommable_e(tc3))) ==  vrai
pre_cons_a(tc4) ==  faux
pre_cons_e(produire_e(tc3 , v)) ==  vrai
pre_cons_a(produire_a(tc4 , v)) ==  vrai
non(estvide(domaine(consommable_a(tc4)))) ==  faux
estvide(domaine(consommable_a(tc4))) ==  vrai
non(vrai) ==  faux
***   conclusion
Le theoreme est demontre dans ce cas


***   noeud numero 242
***   assertions
min(domaine(consommable_e(tc3))) =< max(domaine(ensprod_a(tc4))) + 1 ==  vrai
min(domaine(consommable_e(tc3))) =< max(domaine(ensprod_e(tc3))) + 1 ==  vrai
pre_cons_a(tc4) ==  faux
pre_cons_e(produire_e(tc3 , v)) ==  vrai
pre_cons_a(produire_a(tc4 , v)) ==  vrai
non(estvide(domaine(consommable_a(tc4)))) ==  faux
estvide(domaine(consommable_a(tc4))) ==  vrai
non(vrai) ==  faux
***   conclusion
Le theoreme est demontre dans ce cas


***   noeud numero 25
***   assertions
pre_cons_a(consommer_a(tc4)) ==  vrai
pre_cons_e(consommer_e(tc3)) ==  vrai
ind_cons_e(tc3) =< ind_cons_a(tc4) ==  vrai
pre_cons_e(tc3) ==  vrai
pre_cons_a(tc4) ==  vrai
```



```
***   conclusion
(ind_cons_e(consommer_e(tc3)) =< ind_cons_a(consommer_a(tc4)))

***   noeud numero 251
***   assertions
pre_cons_a(tc4) ==  vrai
pre_cons_e(tc3) ==  vrai
ind_cons_e(tc3) =< ind_cons_a(tc4) ==  vrai
pre_cons_e(consommer_e(tc3)) ==  vrai
pre_cons_a(consommer_a(tc4)) ==  vrai
estvide(domaine(enlever(consommable_e(tc3),
 min(domaine(consommable_e(tc3)))))) ==  faux
non(estvide(domaine(enlever(consommable_e(tc3),
 min(domaine(consommable_e(tc3))))))) ==  vrai
min(domaine(consommable_e(tc3))) =< max(domaine(ensprod_a(tc4))) ==  vrai
estvide(domaine(consommable_e(tc3))) ==  faux
non(estvide(domaine(consommable_e(tc3)))) ==  vrai
non(estvide(domaine(consommable_a(tc4)))) ==  vrai
estvide(domaine(consommable_a(tc4))) ==  faux
***   conclusion
Le theoreme est demontre dans ce cas

** Le noeud 2 est entierement prouve
```

Index

Bibliographie

[Ada80] Ada. *Formal definition of Ada programing language.* Rapport, Honewell Inc. Cii Honewell Bull, 1980.

[Aub79] R. Aubin. Mechanizing structural induction. In *Theoretical Computer Science 9*, pages 329–362, 1979.

[BDJ79] D. Brand, J. Darringer, and J. Joyner. Completeness of conditional reductions. *Proceedings Symposium on Automatic Deduction*, 1979.

[Ber85] G. Berthomieu. Le langage lcs et son interprète. une implantation expérimentale de ccs. In *Actes du premier colloque C^3, Angoulême*, 1985.

[Bir35] G. Birkhoff. On the structure of abstract algebras. In *Proceedings Cambridge Phil. Soc. 31*, pages 433–454, 1935.

[BKS85] W. W. Bledsoe, K. Kunen, and R. Shostak. Completeness result for inequality provers. *Artificial inteligence*, 27:255–288, 1985.

[BM79] R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

[Bou87] Gerard Boudol. *Communication is an Abstraction.* Rapports de Recherche 636, INRIA, Sophia Antipolis, mars 1987.

[Bou88a] W. Bousdira. *A completion procedure for hierarchical conditional equations.* Technical Report, 88-R-021 Centre de Recherche en Informatique de Nancy, 1988.

[Bou88b] W. Bousdira. A completion procedure for hierarchical conditional rewriting systems. In P. Lescanne J. Grabowski and W. Wechler, editors, *Proc. International Workshop on Algebraic and Logic Programming*, pages 93–107, Akademie-Verlag, Berlin, DDR, 1988. To be published by Springer Verlag.

[BR87a] W. Bousdira and J.L. Rémy. Complétion des systèmes de réécriture conditionnelle. In *revue BIGRE+GLOBULE*, Aix-en-Provence, 1987. Actes des journées GROSPLAN.

[BR87b] W. Bousdira and J.L. Rémy. Reveur4 : a laboratory for conditional rewriting. In F.J. Branderburg, G. Vidal-Naquet, and M. Wirsing, editors, *Lecture Notes in Computer Science*, pages 472–473, 4th Symposium on Theoretical Aspects of Computer Science, Springer-Verlag, Passau RFA, 1987.

[BR88] W. Bousdira and J.L. Rémy. Hierarchical contextual rewriting with several levels. In R. Cori and M. Wirsing, editors, *5th Annual Symposium on Theoretical Aspects*

of Computer Science, Bordeaux, pages 193–206, Springer-Verlag, Bordeaux (France), 1988.

[Bur69] R-M. Burstall. Proving properties of programs by structural induction. In *Computer Journal 12*, pages 41–48, 1969.

[Bur88] H-J. Bürckert. Solving disequations in equational theories. In Springer-Verlag, editor, *LNCS 310*, 1988.

[CL88] H. Comon and P. Lescanne. *Equational problems and disunification.* Technical Report 88-R-026, Centre de Recherche en Informatique de Nancy, 1988. To appear in the Journal of Symbolic Computation, special issue on unification, 89.

[Com88] H. Comon. Unification et disunification. Théories et applications. Thèse de l'Institut Polytechnique de Grenoble, 1988.

[CPH*85] G. Cousineau, L. Paulson, G. Huet, R. Milner, M. Gordon, and C. Wadsworth. *The ML Handbook.* INRIA, Rocquencourt, May 1985.

[Der82] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.

[EJ89] M.C. Eglin and J. Julliand. Interprétation parallèle asynchrone de systèmes d' équations. In $10^{ième}$ *séminaire tuniso-français*, AFCET-Université de Tunis, May 1989. Rapport crin 88-R-152.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1. Equations and initial semantics.* Volume 6 of *EATCS Monographs on Theorical Computer Science*, Springer-Verlag, 1985.

[Fin79] J.P. Finance. Etude de la construction des programmes: méthode et langages de spécification et de résolution de problèmes. Thèse de Docteur es sciences mathématiques, 1979.

[Fri86] L. Fribourg. A strong restriction of the inductive completion procedure. In *Proceedings 13th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 226*, pages 105–115, 1986.

[Gal86] J. H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving.* Volume 5 of *Computer Science and Technology Series*, Harper & Row, New-York, 1986.

[Gan87] H. Ganzinger. Ground term confluence in parametric conditional equational specifications. In F. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *STACS 87*, pages 286–298, Springer-Verlag, 1987.

[Gen55] Gerhard Gentzen. *La déduction logique, Traduction et commentaire par Feys Robert et Ladrière jean.* Presses universitaires de France, 1955.

[GG88] S.J. Garland and J.V. Guttag. Inductive methods for reasoning about abstract data types. In *Proceedings of the 15th Symposium on Principles of Programming Languages*, pages 219–228, Association for Computing Machinery, San Diego (USA), 1988.

[GG89] S.J. Garland and J.V. Guttag. An overview of LP, the Larch Prover. In N. Dershowitz, editor, *Proceedings of the 3rd Conference on Rewriting Techniques and Applications, Chapel Hill, North Carolina, USA*, pages 137–151, Springer-Verlag, April 1989. Lecture Notes in Computer Science, volume 355.

[GGS88] S. Garland, J. Guttag, and J. Staunstrup. Verification of vlsi circuits using LP. In *Proceeding of International Workshop on design for Behavioral Verification*, Jyly 1988.

[Gog88] J.A. Goguen. *OBJ as a Theorem Prover.* Technical report SRI-CSL-88-4, SRI International CSL, 1988.

[Gor87] M. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In *Formal Aspects of VLSI Design, North-Holland*, George Milne and P. A. Subrahmanyam, editors, 1987.

[Gri88] E. Pascal Gribomont. From synchronous to asynchronous communication. In *Workshop BCS-FACS on Specification and Verification of Concurrent Systems (Stirling, Scotland, 1988)*, 1988.

[GTW78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In Yeh R., editor, *Current Trends in Programming methodology IV: Data structuring*, pages 80–144, Prentice Hall, 1978.

[Gut78] J.V. Guttag. Abstract data types and software validation. *Communications of the Association for Computing Machinery*, 21:1048–1064, 1978.

[Hee86] J. Heering. Partial evaluation and ω-completeness of algebraic specifications. *Theoretical Computer Science*, 43:149–167, 1986.

[HH82] G. Huet and J-M. Hullot. Proofs by induction in equational theories with constructors. *Journal of Computer and System Sciences*, 25(2):239–266, October 1982. Preliminary version in Proceedings 21st Symposium on Foundations of Computer Science, IEEE, 1980.

[HO80] G. Huet and D. Oppen. Equations and rewrite rules: a survey. In R. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405, Academic Press, New-York, 1980.

[Hoa78] C. Hoare. Communicating sequential process. *Communications of the Association for Computing Machinery*, (21), 1978.

[HS86] J.R. Hindley and J.P. Seldin. *Introduction to combinators and λ-calculus.* Cambridge University Press, 1986.

[Hue73] G. Huet. A unification algorithm for typed lambda calculus. *Theorical Computer Science*, 1(1):27–57, 1973.

[Hue76] G. Huet. Résolution d'equations dans les langages d'ordre 1,2, ...,ω. Thèse d'état de l'Université de Paris 7, 1976.

[Hue80] G. Huet. Confluent reductions : abstract properties and applications to term rewriting systems. *Journal of the Association for Computing Machinery*, 27(4):797–821, October 1980. Preliminary version in 18th Symposium on Foundations Of Computer Science, IEEE, 1977.

[Hul80] J-M. Hullot. Compilation de formes canoniques dans les théories équationelles. Thèse de 3ième cycle, Université de Paris Sud, Orsay, France, 1980.

[JK86] J-P. Jouannaud and E. Kounalis. *Automatic Proofs by Induction in Theories without Constructors.* Technical Report 295, Universite de Paris-Sud, Centre d'Orsay, septembre 1986.

[JLR82] J. P. Jouannaud, P. Lescanne, and F. Reinig. Recursive decomposition ordering. In Bjørner D., editor, *Formal Description of Programming Concepts 2*, pages 331–348, North Holland, Garmisch-Partenkirchen, RFA, 1982.

[JM86] J. Julliand and M. Marmonier. Comedie : outils de développement de systèmes parallèles. In *Actes 3ème Colloque de Génie Logiciel*, Versailles, France, Mai 1986.

[JP85] J. Julliand and G.R. Perrin. Towards a methodology for concurrent programming. In *Proceedings International Conference Parallel Computing 85*, Berlin , RFA, Septembre 1985.

[Kap84] S. Kaplan. *Fair Conditional Term Rewriting Systems: Unification, Termination, and Confluence.* Technical Report, University of Paris Sud, Orsay, Orsay, 1984.

[Kap87] S. Kaplan. Simplifying conditional term rewriting systems : unification, termination and confluence. *Journal of Symbolic Computation*, (4(3)):295–334, 1987.

[KB70] D.E. Knuth and P.B. Bendix. *Simple Word Problems in Universal Algebras*, pages 263–297. J. Leech, Oxford, U.K., pergamon press edition, 1970. Computational Problems in Abstract Algebra.

[KFJP88] A. Koukam, J.P. Finance, J. Julliand, and G.R. Perrin. Spécification et réalisation des types de communication. In *In proceedings of International Symposium on Software Engineering*, Oran - Algérie, 1988.

[KK70] R. Kowalski and D. Kuehner. Linear resolution with selection function. In *Metamathematics Unit*, Edinburgh University Scotland, 1970.

[KNZ87] D. Kapur, P. Narendran, and H. Zhang. On sufficient completeness and related properties of term rewriting systems. *Acta Informatica*, 24:395–415, 1987.

[KR87a] S. Kaplan and J-L. Rémy. Completion algorithms for conditional rewriting systems. In H. Ait-Kaci and M. Nivat, editors, *Colloquium on the Resolution of Equations in Algebraic Structures*, pages 141–170, MCC and INRIA, Austin, (USA), 1987.

[KR87b] E. Kounalis and M. Rusinowitch. On word problem in Horn logic. In J-P. Jouannaud and S. Kaplan, editors, *Proceedings on the first international workshop on Conditional Term Rewriting*, Orsay (France), 1987.

[Lau70] H. Lauchli. An abstract notion of realizability for which intuitionistic predicate calculus is complete. In J. Myhill, A. Kino, and R.E. Vesley, editors, *Intuitionism and Proof Theory*, pages 227–234, North-Holland, Amsterdam, 1970.

[Laz86] N. Lazrak. *Spécification temporelle de relations de communication.* Rapport de D.E.A, Centre de Recherche en Informatique de Nancy, 1986.

[Laz88] A. Lazrek. Etude et réalisation de méthodes de preuve par récurrence en logique équationnelle. Thèse de l'Institut National Polytéchnique de Lorraine, 1988.

[Laz89] N. Lazrak. PAUSE: a theorem prover for elementary properties in equational specifications. In *Proceeding of the Workshop on Computer-aided Development of Proofs, Theories, Programs, and Circuits. Madrid*, july 1989.

[Laz90a] N. Lazrak. *Equational validity of equational formulas.* Rapport C.R.I.N 90-R-44, Centre de Recherche en Informatique de Nancy, 1990.

[Laz90b] N. Lazrak. PAUSE; how to prove equational formulas in equational specifications. In *Rapport C.R.I.N 90-R-43*, System demonstration in 7-th Symposium on Theoretical Aspects of Computer Science, Rouen France, Febraury 1990.

[LC86] P. Le Certen. Conception et mise en œuvre d'un langage impératif pour la programmation parallèle. Thèse de Doctorat de l'univercité de Rennes 1, 1986.

[Les83] P. Lescanne. Computer experiments with the REVE term rewriting systems generator. In *Proceedings, 10th ACM Symposium on Principles of Programming Languages*, pages 99–108, ACM, 1983.

[LLT86] A. Lazrek, P. Lescanne, and J-J. Thiel. *Proving inductive equalities, Algorithms and Implementation.* Technical Report, Internal Report CRIN 86-R-087, 1986.

[Lov70] D. W. Loveland. A linear format for resolution. In *Proc. IRIA Symp. Automatic Demonstration*, Springer-Verlag, 1970.

[LPF89] N. Lazrak, G-R Perrin, and J-P Finance. *Verification of elementary properties in abstract data types.* Rapport interne CRIN 89-R-042, Centre de Recherche en Informatique de Nancy, 1989.

[Mar73] P. Martin-Löf. An intuitionistic theory of types: predicative part. In E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, pages 73–118, North-Holland, Amsterdam, 1973.

[Mar86] P. Marchand. Cour de logique de dea. Université de Nancy I, 1986.

[Mar89] M. Marmonier. Spécification et validation de systèmes de processus communicants. Thèse de l'Université de Nancy I, 1989.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[Mus80] D.L. Musser. On proving inductive properties of abstract data types. In *Proceedings 7th ACM Symp. on Principles of Programming Languages*, pages 154–162, Association for Computing Machinery, 1980.

[Nav87] M. Navarro. *Técnicas de reescritura para especificationes condicionales.* PhD thesis, University of Barcelona (Spain), 1987.

[Nil71] N. J. Nilsson. *Problem solving Methods in Artificial intelligence.* McGraw-Hill New York, 1971.

[Nip88] T. Nipkow. *Equational Reasoning in Isabelle.* Technical Report, Departement of Computer Science, University of Manchester, 1988.

[NW83] T. Nipkow and G. Weikum. A decidability result about sufficient completeness of axiomatically specified abstract data types. In *6th GI Conference*, pages 257–268, Springer-Verlag, Lecture Notes in Computer Science, 1983.

[Pau84] E. Paul. Proof by induction in equational theories with relations between constructors. In B. Courcelle, editor, *Proceedings 9th Colloquium on Trees in Algebra and Programming*, pages 210–225, Cambridge University Press, 1984.

[Pau85] E. Paul. *Manuel OASIS.* Technical Report R.G. 6. 85, Greco programmation, 1985.

[Pau86] L. Paulson. Natural deduction proof as higher–order resolution. *Journal of Logic Programming*, vol. 3:pages 237–258, 1986.

[Pau87] L. C. Paulson. *The foundation of a generic theorem prover.* Technical Report Report 130, Computer Lab. Univ. of Cambridge, 1987.

[Pau88] L. Paulson. *A Preliminary Users's Manual For ISABELLE.* Technical Report 133, University of Cambridge, Computer Laboratory, 1988.

[Per85] G. R. Perrin. La communication: un outil pour la spécification, la construction et la vérification de systèmes parallèles. Thèse de Docteur es sciences mathématiques, 1985.

[Pla78] D. Plaisted. A recursively defined ordering for proving termination of term rewriting sytems. *Dept. of Computer Science Report 78-943*, 1978.

[Rem82] J-L. Rémy. Etude des systèmes de réécriture conditionnels et applications aux types abstraits algébriques. Thèse d'Etat de l'Institut National Polytechnique de Lorraine, 1982.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12, 1965.

[Rus85] M. Rusinowitch. Path of subterm ordring and recursive decomposition ordering revisited. In *Proceedings 1st Conference Rewriting Technique and Applications, Lecture Notes in Computer Science 202*, pages 225–240, Springer Verlag, Dijon France, 1985.

[Rus87] M. Rusinowitch. Démonstration automatique par des techniques de réécriture. Thèse d'Etat de l'Université de Nancy I, 1987.

[RZ84] J.L. Rémy and H. Zhang. Reveur4 : a system for validating conditional algebraic specifications data types. In T. O'Shea, editor, *Proceedings of the European Conference on Artificial Intelligence*, ECAI, Pisa, Italy, 1984.

[SBGP87] J. Staunstrup, F. Barrett, M. Greenstreet, and Moller-Nielsen P. The design of a problem-mesh. In *Proceeding from Cmp-Euro 87, VLSI and Computers, IEEE*, May 1987.

[Sho67] J. R. Shoenfield. *Mathematical logic.* Addison-Wesley. Reading. Masschusetts, 1967.

[Sif79] J. Sifakis. Le contrôle des systèmes asynchrones: concepts, propriétés, analyse statique. Thèse d'Etat, Université de Grenoble, 1979.

[Sla71] J. R. Slagle. *Artificial intelligence, the Heuristic Programming Approch.* McGraw-Hill New York, 1971.

[Sla72] J. R. Slagle. Automatic theorem proving with built-in theories inclluding equality, partial ordering and sets. *Journal of the Association for Computing Machinery*, 19:120–134, January 1972.

[SN73a] J. R. Slagle and L. Norton. Automated theorem-proving for the theories of partial and total ordering. *The Computer Journal*, 18:49–54, 1973.

[SN73b] J. R. Slagle and L. Norton. Experiments with an automatic theorem prover having partial ordering rules. *Communications of the Association for Computing Machinery*, 16:683–688, 1973.

[STE*82] C.A. Sunshine, D.H. Thompson, R.W Erickson, S.L. Gerhart, and D. Schwabe. Specification and verification of communication protocols in AFFIRM using state transition models. *IEEE Transactions on Software Engineering*, SE-8:460–488, September 1982.

[TW82] J.W. Thatcher and E.G. Wagner. Data type specification : parametrization and the power of specification techniques. In *ACM TOPLAS 4,4*, pages 711–732, 1982.

[Zha84] H. Zhang. Reveur4 : etude et mise en œuvre de la réécriture conditionnelle. 1984.

[Zha88] H. Zhang. *Reduction, Superposition and Induction: Automated Reasoning in an Equational Logic.* PhD thesis, Rensselaer Polytechnic Institute, Department of Computer Science, Troy, NY, 1988.

[ZKK88] H. Zhang, D. Kapur, and M.S. Krishnamoorthy. A mechanizable induction principle for equational specifications. In E. Lusk and R. Overbeek, editors, *Proceedings 9th International Conference on Automated Deduction*, pages 162–181, Springer-Verlag, 1988.

[ZR85] H.T. Zhang and J-L. Rémy. Contextual rewriting. In J.P. Jouannaud, editor, *First Conference on Rewriting Techniques and Applications*, pages 46–62, Springer-Verlag, Dijon (France), 1985.

NOM DE L'ETUDIANT : LAZRAK Nourreddine

NATURE DE LA THESE : Doctorat de l'Université de NANCY I en Informatique

VU, APPROUVE ET PERMIS D'IMPRIMER

NANCY, le 0 1 FEV. 1990 n°264

LE PRESIDENT DE L'UNIVERSITE DE NANCY I

UNIVERSITE NANCY I
LE PRESIDENT
M. BOULANGE

Notre objectif dans cette thèse est de vérifier la validité d'une *formule équationnelle* dans une spécification algébrique axiomatisée par un ensemble d'équations.

Une formule équationnelle est une formule universellement quantifiée d'un langage du premier ordre dont les formules atomiques sont des équations simples.

Nous proposons de généraliser et d'adapter les méthodes utilisées dans le cas des équations simples pour prouver la validité des formules équationnelles. Ainsi, dans le cas de la validité inductive nous avons utilisé l'induction structurelle basée sur la récurrence classique. Nous avons proposé une méthode pour la génération automatique des schémas d'induction dans certains cas particuliers. Dans le cas de la validité équationnelle nous avons proposé une méthode basée sur la transformation et la décomposition de la formule initiale pour déterminer un ensemble représentatif d'équations simples. La validité équationnelle de ces équations est équivalente à celle de la formule initiale.

Un autre mécanisme de raisonnement appelé le chaînage transitif est proposé pour tester la validité des équations contenant des relations d'ordre.

L'étude théorique des principes proposés dans cette thèse a abouti à la réalisation du système de preuve *PAUSE* que nous avons utilisé pour vérifier certaines relations d'asynchronisme entre les composantes qui gèrent la communication dans un système parallèle.

Mots-clés :
Preuve automatique, formule équationnelle, induction structurelle, décomposition, réécriture, raisonnement par cas, chaînage transitif, communication.