# Vérification formelle des systèmes parallèles décrits en UNITY à l'aide d'un outil de démonstration automatique

Boutheïna Chetali

Université Henri Poincaré – Nancy I

Département de Formation Doctorale en Informatique — École Doctorale IAE + M

# Vérification Formelle des Systèmes Parallèles décrits en UNITY à l'aide d'un outil de Démonstration Automatique

## THÈSE

présentée et soutenue publiquement le 28 Mai 1996

pour l'obtention du

**Doctorat de l'Université Henri Poincaré – Nancy I**

**(Spécialité Informatique)**

par

Boutheïna CHETALI

**Composition du jury**

| | |
|---|---|
| *Président :* | Jean-Pierre FINANCE |
| *Rapporteurs :* | Max DAUCHET<br>Stephen J. GARLAND<br>Jean-Pierre THOMESSE |
| *Examinateurs :* | Dominique BOLIGNANO<br>Pierre LESCANNE<br>Joseph SIFAKIS |

Centre de Recherche en Informatique de Nancy
INRIA–Lorraine

# Remerciements

*Je voudrais tout d'abord exprimer ma profonde gratitude à Pierre Lescanne qui m'a dirigée et conseillée avec une rare gentillesse et compréhension durant ces années de recherche et sans qui ce travail n'aurait pu voir le jour. Sa compétence, sa permanente disponibilité et ses encouragements m'ont été des plus précieux.*

*Je souhaite également remercier chaleureusement ceux qui ont bien voulu prendre part à ce jury :*

- *Jean-Pierre Finance qui, d'abord en tant que Directeur du Centre de Recherche en Informatique de Nancy puis en tant que Président de l'Université Henri Poincaré, m'a permis de mener cette thèse dans d'excellentes conditions et m'a fait l'honneur d'en présider le jury.*
- *Stephen J. Garland, malgré ses engagements et responsabilités de Directeur de recherche au laboratoire d'Informatique du* MIT, *a accepté d'examiner ce travail et d'écrire un rapport malgré l'obstacle de la langue.*
- *Max Dauchet, Professeur à l'Université des Sciences et Technologies de Lille, qui s'est prêté à une lecture avisée et intéressée de ce document.*
- *Dominique Bolignano, Responsable de l'équipe de méthodes formelles à* BULL, *a consacré beaucoup de son temps à l'étude de ce travail, en acceptant la lourde tâche de rapporteur. Ses conseils et critiques durant ces années de recherche m'ont été très utiles et m'ont permis d'apercevoir un premier impact de ce travail dans le milieu industriel.*
- *Joseph Sifakis, Directeur de Recherche au CNRS, a bien voulu s'intéresser à ce travail et je le remercie vivement de l'intérêt qui lui a porté. Ses commentaires et critiques furent très enrichissants.*
- *Jean-Pierre Thomesse, Professeur à l'Institut National Polytechnique de Lorraine, qui en tant que rapporteur interne a accepté d'examiner ce document. Les discussions qui en ont découlées m'ont été extrêmement bénéfiques.*

*Je tiens à remercier par ailleurs tous les membres des équipes EURECA et PROTHEO, en particulier Eric Domenjoud pour ses conseils techniques judicieux et Laurent Vigneron pour son aide précieuse en période de rédaction.*

*Je suis très reconnaissante à Patrice Bonhomme pour toute l'attention, l'aide, le soutien et les encouragements qu'il m'a prodigués, et à Laurent Romary qui s'est porté à mon secours à de nombreuses reprises, me faisant profiter de son expérience et de ses conseils judicieux.*

*Enfin, je remercie tous ceux qui m'ont encouragée et soutenue durant ces années, en particulier mes compagnons de galère..., Narges Berregeb, Slim Ben Attalah, Riadh Robbana, Farid Ajili et Barbara Heyd.*

# Résumé

Cette thèse est consacrée à l'utilisation des méthodes formelles de spécification et de vérification dans le cadre des techniques déductives basées sur la preuve de théorèmes. En particulier, nous nous intéressons à la spécification et à la vérification mécanique de programmes parallèles décrits en UNITY à l'aide du démonstrateur du LARCH, LP. Nous décrivons la formalisation et la mécanisation de la logique et de la méthodologie d'UNITY à l'aide d'un outil de démonstration automatique du premier ordre et à large spectre tel que LP et à leur mise en oeuvre dans des exemples utiles et conséquents. Nous formalisons dans un premier temps la syntaxe et la sémantique d'UNITY dans l'environnement de LP en choisissant comme outil formel la plus faible pré-condition introduite par Dijkstra. Cette modélisation comprend la représentation syntaxique concrète des objets prédicats, de la notation de programmation et des prédicats temporels de UNITY dans une logique du premier ordre. Nous décrivons la construction et la validation d'une base de faits basée sur l'approche des spécifications LSL. Nous proposons une méthodologie de preuve incrémentale basée sur l'utilisation d'un démonstrateur pour la vérification mécanisée dans le but à la fois d'aider à la mise au point des preuves et à la réutilisation des preuves.

Nous illustrons l'approche proposée à l'aide de trois études de cas. La vérification formelle mécanique d'un *protocole de communication* à travers des canaux défectueux met en évidence la méthodologie utilisée pour montrer des propriétés de sûreté et de vivacité et comment un démonstrateur peut être effectivement utilisé pour détecter des failles dans la spécification. La vérification du problème des *lecteurs rédacteurs* illustre un aspect important dans l'utilisation des démonstrateurs, à savoir la réutilisation et la mécanisation des preuves. Enfin, la vérification d'un protocole de *contrôle d'un ascenseur* permet de comparer notre approche à celle utilisée avec le démonstrateur d'ordre supérieur HOL.

**Mots-clés:** Vérification formelle, Vérification mécanique, Méthodologie de vérification, Preuves de Théorèmes, Protocole de communication, Unity

# Abstract

This thesis is an approach to the formal specification and verification of distributed systems and in particular to the computer assisted verification. In this work, we use the LARCH Prover to verify concurrent programs and the chosen specification mechanism is "UNITY logic". We describe the mechanization of the syntax and the semantic of UNITY in LP, and how we can use the theorem proving methodology to prove safety and liveness properties. We choose the calculus of the weakest pre-condition of Dijkstra as the formal tool to reason about UNITY programs. We choose a concrete syntactical representation of state dependent predicates and the programming notation in first order logic. To ensure the soundness of the encoding, we use an incremental method for the specifications and standard techniques for the verification.

To illustrate the feasibility of our approach, we present three case studies of different complexities. The first one describes the formal verification of a *communication protocol over faulty channels*, where we show how we can use the theorem proving methodology to prove safety and liveness of a communication protocol and how a theorem prover can be actually used to detect flaws in a system specification. The second study presents the mechanical proof of the correctness of the *reader-writer* problem. We use this small example to illustrate the level of details needed to mechanize a structured hand proof. The last study describes the proof of a *lift-control* protocol. As the original proof was developed with the HOL theorem prover, we use this example for a first comparison with the HOL proofs.

**Keywords:** Formal Verification, Theorem proving, Theorem Prover methodology, Computer checked proof, Communication Protocol, Unity

*A mes parents*

# Table des matières

# Table des figures

# Introduction

L'importance des méthodes formelles pour la spécification et la vérification des systèmes informatiques est un fait établi, reconnu par les industriels, après l'avoir été par la communauté académique. Des applications mettant en jeu des vies humaines (avions, centrales nucléaires, médecine) aux programmes de jeux pour enfants, en passant par les applications stratégiques (bourse, armée), le logiciel est omniprésent. Disposer de méthodes de validation et de vérification efficaces est donc nécessaire pour la fiabilité de systèmes de plus en plus complexes. Les récents problèmes du "bug" du Pentium, et du système SOCRATE sont là pour nous le rappeler.

L'architecture des logiciels modernes intègre de plus en plus de parallélisme, du fait notamment de l'importance que prennent les architectures clients-serveurs et de manière générale les architectures distribuées. Or ces systèmes, où différents composants interagissent, sont difficiles à appréhender, à maîtriser, et surtout à mettre au point. Par conséquent, compte tenu de la faillite des techniques classiques de conception, de débogage et de test dans ces systèmes, les méthodes de vérification formelle deviennent les seuls outils dans ce contexte.

Plusieurs techniques de vérification mécanique sont utilisées ou combinées, à savoir les techniques dites *déductives* et les techniques par *évaluation de modèles.* Ces dernières sont automatiques à l'inverse des techniques déductives mais sont d'un usage limité aux systèmes finis. Pour les logiciels distribués, pour lesquels les outils traditionnels de test ne sont plus utilisables, à cause notamment de la non-reproductibilité des anomalies et de l'explosion combinatoire, l'utilisation des méthodes déductives semble être une alternative prometteuse. Bien qu'une grande diversité de formalismes existe et que de nombreux outils de validation performants soient disponibles, l'utilisation des méthodes déductives semble être confinée aux milieux académiques[1]. Ceci est probablement dû d'une part, à l'absence de méthodologie dans l'utilisation des outils de validation, et d'autre part au manque d'exemples convaincants pour illustrer l'application des méthodes déductives.

Par ailleurs, les méthodes formelles sont souvent associées à la notion de *vérification* en occultant celle de *spécification.* Or les spécifications formelles, outre leur nécessité théorique, sont un moyen de communication puissant dans une communauté d'experts qui collaborent au développement d'un logiciel. Elles permettent l'utilisation d'un langage non ambigu pour communiquer et pour cerner les besoins afin de produire une spécification formelle à partir des spécifications informelles souvent vagues et contradictoires.

Dans ce contexte, notre objectif consiste à étudier l'utilisation d'un logiciel de démonstration automatique pour la spécification et la vérification des systèmes parallèles. En particulier, nous nous sommes intéressés à la vérification mécanisée des programmes UNITY [Cha88] à l'aide du démonstrateur de théorèmes du LARCH, à savoir LP [Gar91]. Ces deux outils choisis, l'un comme environnement de spécification et l'autre comme outil de vérification, nous permettrons

1. Il est possible que certaines industries utilisent ces méthodes (comme par exemple INTEL) mais n'en font pas une grande publicité pour conserver un avantage technologique.

d'illustrer certains aspects importants liés aux méthodes formelles, à savoir la nécessité de spécifier rigoureusement les objets sur lesquels porte le raisonnement et la fragilité des preuves informelles généralement basées sur des intuitions dont un outil de validation ne peut se satisfaire.

## Spécification et vérification formelle

Le problème de fiabilité et de correction d'un système a pris les proportions d'une crise dans le domaine de développement du logiciel. Ce problème n'est pas nouveau, il est apparu dans les années soixante comme étant *la crise du logiciel* et 30 ans après, il est encore présent.

Les approches courantes de développement de logiciel sont primitives et *ad-hoc*. Les programmes sont développés avec un processus d'essai et d'erreur. Étant donné un problème, le programmeur utilise son intuition pour développer un programme supposé le résoudre. Le programmeur estime que son travail est correct lorsque le programme conçu s'exécute correctement sur un sous ensemble d'entrées qu'il considère comme typiques. Dans un logiciel complexe, plusieurs de ces programmes (possiblement erronés) interagissent, et les erreurs subtiles et insidieuses peuvent provenir de cette interaction. Localiser des erreurs dans un tel système revient à trouver une aiguille dans une botte de foin : c'est un processus long, coûteux en efforts et en temps et pas toujours couronné de succès.

Cette approche *ad-hoc* pour le développement de logiciel a un certain nombre de conséquences néfastes. Tout d'abord, la nature imprévisible du débogage fait qu'il est difficile pour les gestionnaires de logiciel de prévoir des *deadlines* aux projets. De plus, l'absence d'un argument convaincant de correction se reflète par l'absence de garantie des produits logiciels lorsqu'ils sont livrés aux clients. Enfin, ces approches ne peuvent être généralisées à de larges systèmes et sont inacceptables lorsqu'on développe des programmes dont la fiabilité est cruciale pour la sécurité ou la vie d'êtres humains.

Dès les années soixante, R. Floyd, C.A.R Hoare et W. Dijkstra furent les pionniers d'une nouvelle approche pour le développement des programmes. Cette approche révolutionnaire consistait à *vérifier formellement* un programme à partir de son texte sans jamais l'exécuter. Le processus de conception d'un programme fut alors élevé à la preuve de théorèmes mathématiques. R. Floyd [Flo67] introduit l'idée des assertions inductives pour prouver la correction des programmes séquentiels, où un programme est spécifié comme un diagramme et sa vérification est réduite à prouver deux types de propriétés : l'*invariance* (ou correction partielle) et la *terminaison* (ou correction totale). Ceci a été suivi par plusieurs théories pour le développement des programmes séquentiels, notamment les *triplets* de correction partielle de Hoare [Hoa69] et la *plus faible pré-condition* de Dijkstra [Dij76].

Dans les années soixante dix, l'importance du non-déterminisme et l'apparition du parallélisme ont poussé les chercheurs à étendre ces techniques de vérification pour prendre en compte les programmes dits *parallèles* et/ou *concurrents*. Les programmes parallèles sont des objets complexes à concevoir et à analyser et le parallélisme ajoute une dimension supplémentaire à la difficulté intrinsèque de l'activité de programmation. Cette complexité rend d'autant plus nécessaire une approche formelle pour la spécification et la vérification. Plusieurs approches furent alors développées pour la spécification et la vérification des programmes parallèles, notamment CSP [Hoa85] de Hoare, CCS [Mil80] de Milner, celles basées sur la logique temporelle, [Lam91, Man91, Man82], et enfin les approches axiomatiques [Owi76, Lam82]. Contrairement à l'approche séquentielle, aucune approche ne fait l'unanimité en ce qui concerne le parallélisme.

A la fin des années 80, Chandy et Misra développent UNITY [Cha88], un environnement pour la spécification et la vérification des programmes parallèles. Le traitement du parallélisme est

alors ramené au strict nécessaire (à l'essentiel), où tous les aspects du langage de programmation et ceux liés à l'architecture cible sont occultés.

## Deux approches pour la vérification mécanique

La méthode de *vérification de modèles* (*model checking*) [Que81] [Cla86] [Arn89] a été utilisée avec succès pour vérifier de façon automatique des propriétés d'un système à comportement fini, avec une interaction minime de l'utilisateur. Le système est représenté par un modèle de transition d'état et les propriétés sont spécifiées dans une logique temporelle. Cependant, le problème de complexité (de graphe d'états) limite souvent l'application de cette méthode.

La *preuve de théorèmes* est une alternative prometteuse à la vérification automatique de modèles, où la vérification formelle est basée sur l'utilisation d'un démonstrateur automatique. En effet, compte tenu de la complexité de la vérification et des objets manipulés, il est essentiel de pouvoir automatiser la conduite des preuves ou tout au moins d'en assurer la validité par l'utilisation d'un assistant de preuve.

## Pourquoi Unity?

"Because its simplicity and elegance, especially with respect to its treatment of progress properties, UNITY may be better suited as a tool for the design of correct software."

"The sheer number and diversity of examples that Chandy and Misra tackle, and their complexity, comprises a convincing argument for the practical significance of this approach"

Ces appréciations d'Edgar Knapp [Kna94] et de Rob Gerth et d'Amir Pnueli [Ger89] à propos d'UNITY résument les deux principales raisons pour le choix d'UNITY comme environnement de spécification et de vérification de programmes parallèles et distribués. Chandy et Misra se sont basés sur le principe que la plupart des difficultés proviennent du fait que l'on n'arrive pas à séparer l'algorithme abstrait de son implémentation. Par conséquent, ils ont développé un environnement de spécification et de vérification de programmes parallèles, où les notions de contrôle de flux de données et d'architectures sont occultés. Un programme UNITY n'est autre qu'un ensemble d'instructions d'affectations conditionnelles multiples. Une exécution d'un programme débute dans un état satisfaisant les conditions initiales et se poursuit par exécution des affectations sélectionnées suivant la règle d'équité *inconditionnelle* suivante : chaque instruction est choisie *infiniment souvent*. L'exécution se poursuit indéfiniment jusqu'à ce que le programme atteigne son point fixe, c'est à dire un état tel que toute affectation exécutée à cet état ne modifie pas les variables du programme.

La logique d'UNITY est un fragment de logique temporelle dans le sens où elle comprend la logique du premier ordre avec les règles usuelles du calcul des prédicats, enrichie avec des modalités temporelles pour exprimer les propriétés qu'un programme parallèle doit vérifier. Malgré sa simplicité, UNITY a été utilisé pour dériver des algorithmes pour une large variété de problèmes rencontrés dans la programmation parallèle [Sta88, Kna90] et dans l'industrie [Sta92, Sta93].

Le modèle proposé regroupe plusieurs notions essentielles de programmation telles que : synchrone, car il permet de modifier la valeur de plusieurs variables par une seule instruction (atomique); asynchrone en spécifiant peu sur l'ordre dans lequel seront exécutées les différentes instructions du programme; communication de processus par partage de variables, par envoi de messages, etc.

Plus précisément, UNITY fournit des prédicats pour la spécification, des règles pour dériver des

spécifications directement à partir du texte du programme et un système de preuve pour vérifier les propriétés importantes des programmes concurrents, à savoir *la sûreté* et *la vivacité.*

La caractéristique importante d'UNITY est la séparation entre l'algorithme spécifiant le problème et l'architecture sur laquelle doit être implanté l'algorithme. Il définit une sémantique générale pour les programmes concurrents qui permet des raffinements du programme *général* (indépendant de l'architecture) à un programme spécifique à une architecture particulière.

Malgré la relative simplicité du modèle proposé, la vérification formelle d'un programme spécifié en UNITY nécessite une analyse précise et minutieuse mettant en jeu un nombre considérable de détails cruciaux pour la preuve. Par conséquent, l'utilisation d'un démonstrateur de théorèmes s'avère nécessaire pour automatiser certaines étapes de vérification afin de valider la correction des preuves effectuées et donc du programme.

Plusieurs outils ont été utilisés pour la vérification des programmes UNITY, tels que le système NQTHM de Boyer et Moore [Boy88], le démonstrateur d'ordre supérieur HOL [Gor93] et plus récemment, la méthode B [Bro93].

## Pourquoi le démonstrateur du Larch?

UNITY offre une logique et une notation permettant de décrire le calcul et les raffinements successifs du programme par une spécification abstraite, sans faire référence à la séquence d'exécution du programme. Cette vue syntaxique du programme rappelle le style des spécifications du projet LARCH [Gut93], qui mettent l'accent sur la clarté plutôt que sur l'exécutabilité. Ce projet a donné lieu à un outil d'aide à la vérification, le LARCH PROVER, qui est d'un abord assez simple tout en fournissant de nombreuses méthodes de preuve.

LP est considéré comme un assistant de preuve et non comme un démonstrateur automatique du fait qu'il requiert une certaine dose d'interactivité. Nous considérons cette dernière comme une caractéristique primordiale dans le sens où le but que nous lui assignons consiste à détecter des erreurs de conception, ce qui perturbe généralement gravement les outils très automatisés aptes à prouver élégamment mais inaptes à échouer proprement. Or notre recherche s'inscrit plutôt dans la certification et la mise au point des preuves que dans l'automatisation des preuves.

En effet, dans le contexte de développement de logiciel, vérifier une étape de conception ne se ramène pas à répondre par un "oui" ou par un "non", mais plutôt à faire un diagnostic de la preuve afin de déceler les causes des erreurs de conception pour permettre leur correction. Dans un tel contexte, nous considérons LP comme un assistant de preuve ou un *correcteur de preuve*, car il n'effectue pas de façon automatique des preuves difficiles et il n'utilise pas d'heuristiques, comme le système NQTHM de Boyer&Moore [Boy88], pour générer des conjectures utiles à la preuve. Par contre, il offre un ensemble d'outils et de méthodes pour la mise au point des preuves (expressivité, différents niveaux de traces) tout en offrant un degré d'automatisation suffisant pour mener à bien et automatiquement les étapes fastidieuses et répétitives des preuves.

## Les objectifs

Dans cette thèse, nous nous intéressons à la formalisation et à la mécanisation de la logique et de la méthodologie d'UNITY à l'aide d'un outil de démonstration automatique du premier ordre et à large spectre tel que LP [Gar91] et à sa mise en oeuvre dans des exemples utiles et conséquents . Le but de cette recherche étant de spécifier et de vérifier des propriétés de *sûreté*

et de *vivacité* des programmes concurrents non triviaux spécifiés dans l'environnement UNITY, de proposer une méthodologie de preuve à l'aide d'un outil de vérification mécanique et enfin de valider cette approche à travers des études de cas réalistes.

Nous nous intéressons particulièrement à l'utilisation des méthodes formelles de spécification et de vérification dans le cadre des techniques déductives basées sur la preuve de théorèmes. Pour cela, nous explorons la formalisation de la logique sous jacente à UNITY dans la logique du premier ordre de LP. Cette formalisation consiste d'une part à choisir un modèle sémantique pour les programmes UNITY et d'autre part, à définir les concepts d'UNITY dans ce modèle.

La sémantique d'un programme UNITY peut être vue comme celle d'un modèle de transition d'états, où les variables du programme caractérisent l'ensemble d'états et où les actions du programme représentent les transitions. Une exécution d'un programme UNITY débute dans un état satisfaisant la condition initiale et se poursuit indéfiniment en choisissant une instruction à exécuter. La sélection est sujette à la contrainte d'*équité inconditionnelle*: Pour une exécution infinie, chaque instruction est choisie infiniment souvent. Les règles de preuves d'UNITY pour montrer les propriétés de progrès sont construites en utilisant cette contrainte d'équité. Dans un tel modèle, un programme UNITY peut être considéré comme un ensemble de séquences d'exécution équitables. Par contre, alors que dans un modèle de calcul basé sur les états, le raisonnement porte sur une séquence d'exécution (à la fois), les opérateurs temporels de UNITY expriment des propriétés sur les programmes, considérés comme des ensembles de séquences d'exécution équitables.

Nous avons choisi pour modèle sémantique le calcul de la *plus faible pré-condition* de Dijkstra, en raison de la simplicité des constructions UNITY qui ne sont que des affectations. Nous avons alors formalisé la logique d'UNITY et son système de preuve. Cela consiste à définir une base de faits représentant les données de base sur lesquelles porte le raisonnement. Cette base de fait définissant les entiers, les listes, les expressions booléennes, est utilisée dans tous les programmes UNITY et plus généralement, elle est indépendante du modèle UNITY. Par la suite, nous avons formalisé le calcul des prédicats nécessaire pour définir les trois relations temporelles qui sont à la base de la logique d'UNITY et son système de preuve constitué par des règles d'inférence dérivées de ces relations temporelles.

La deuxième partie consiste à appliquer cette formalisation à des programmes UNITY afin de vérifier leur correction, et ceci à l'aide du démonstrateur LP. Le but de ces études de cas est d'une part, de prendre en compte des problèmes de tailles croissantes et d'autre part, d'illustrer l'utilisation des méthodes formelles de vérification lorsqu'on s'intéresse à des problèmes complexes, pour la détection des erreurs dans la spécification initiale ou dans la preuve informelle. En effet, LP fournit un langage de spécification et un système interactif de vérification basé essentiellement sur la réécriture. Néanmoins, il ne fournit aucune indication sur la méthode à utiliser pour une application pratique, c'est à dire qu'il ne fournit pas de méthodologie. Or pour un application pratique, la question décisive est de savoir comment organiser le travail, *quand* spécifier *quoi* et à quel niveau de détail, et *quand* prouver *quoi*.

Notre première expérience, et non la moindre, fut de vérifier la correction d'un protocole de communication *à travers des canaux défectueux*. Cette application, dont les résultats sont conséquents nous, a permis de traiter ensuite le problème des *lecteurs-rédacteurs* ainsi qu'un protocole de *contrôle d'un ascenseur*.

## Plan de la thèse

Ce document est organisé en deux parties. La première partie est consacrée à la formalisation de la base de faits, des objets manipulés par les programmes, tels que les entiers naturels ou les listes, et celle d'UNITY. Nous commençons par une présentation du démonstrateur LP et de ses concepts fondamentaux. Nous décrivons, dans le chapitre 2, notre approche pour la formalisation d'UNITY, qui consiste à définir le méta-langage des prédicats nécessaire à la définition du calcul des *wp* ainsi que les substitutions textuelles nécessaires à la formalisation des affectations. Cela nous permet de définir les opérateurs temporels en LP et les règles d'inférence du système de preuve associé à UNITY. Nous présentons par la suite la spécification et la vérification de la base de faits qui comprend les objets de base manipulés par les programmes UNITY. Enfin, nous proposons, dans le dernier chapitre de cette première partie, une méthodologie de spécification et de vérification à l'aide d'un outil de démonstration automatique. En particulier, nous proposons une méthode incrémentale pour la spécification d'une base de travail ainsi que des principes de gestion et d'organisation des preuves complexes.

La seconde partie de cette thèse est dédiée aux études de cas. En particulier, nous décrivons la vérification de trois études de cas de différentes complexités, à savoir la vérification d'un protocole de communication à travers des canaux défectueux, celle du problème des lecteurs rédacteurs et enfin la vérification d'un protocole de contrôle d'ascenseur.

Nous concluons ce document par l'étude des travaux connexes, de la portée de cette recherche, et enfin l'analyse des perspectives de ce travail.

# Première partie

# Formalisation et Preuve

# Chapitre 1

# Le démonstrateur de théorème de Larch

## 1.1 Introduction

Les méthodes formelles de spécification et de vérification ont été proposées pour réaliser l'impossible, à savoir obtenir des programmes et des systèmes fiables prouvés corrects. Néanmoins, jusqu'ici ces méthodes ont été considérées difficiles et coûteuses en pratique. Par conséquent, certains doutes ont été formulés quant à l'utilisation des méthodes formelles en pratique ainsi que sur leur généralisation. Néanmoins, depuis un certain nombre d'années, quelques expériences réussies ont été menées dans la vérification formelle des circuits matériels, dont on peut citer la plus connue, la vérification d'un microprocesseur par Hunt [Hun86]. Cette vérification ne s'est pas arrêtée aux circuits matériels mais à toute forme de logiciels, comme par exemple la vérification d'un algorithme de récupération de mémoire (*Garbage Collector*) concurrent par Doligez et Gonthier [Dol94].

Les méthodes formelles se sont avérées nécessaires pour améliorer la fiabilité des logiciels. Dans ses débuts, la vérification formelle concernait des programmes finis (vérification *post-mortem*), mais on s'est vite aperçu que cette vérification serait plus profitable si elle était réalisée en cours de conception du programme, en partant d'une spécification formelle du problème donné et en construisant le programme et sa preuve pas à pas. C'est en particulier les méthodes de transformation de programmes qui permettent, avec une approche descendante mécanisée, de construire des programmes corrects.

Néanmoins, même les méthodes de vérification les plus avancées ne peuvent résoudre aussi facilement le problème de la fiabilité de logiciel. La fiabilité d'un programme *correct* (vérifié formellement) dépend de façon cruciale de la correction de sa spécification formelle. En effet, la vérification n'a de sens que si les besoins sont formulés correctement. En fait, dans une application pratique, c'est une tâche importante dans le développement que de cerner les besoins et de donner une spécification formelle à partir des spécifications informelles, souvent vagues et contradictoires. De plus, il s'avère souvent nécessaire de modifier et de raffiner la spécification initiale pendant le processus de développement, pour prendre en compte de nouveaux faits et pour avoir des solutions élégantes et plus efficaces. Par conséquent, produire les spécifications des besoins a un rôle crucial dans le développement. L'ingénierie de le spécification est le domaine d'application le plus important des méthodes formelles, où les techniques de spécifications, incluant les techniques de vérification, ont un impact crucial . Néanmoins, ces méthodes nécessitent un nouveau type de développement et par conséquent des outils différents.

Plusieurs projets dans le domaine de l'ingénierie du logiciel ont élaborés des modèles et des théories. Mais ces domaines ne sont pas toujours formalisés, ce qui crée de nouveaux problèmes

pour la programmation et les méthodes formelles. En effet, ces domaines doivent être connus par l'ingénieur et doivent être mis dans une certaine forme de façon à ce qu'ils puissent être utilisés pour les spécifications et leurs représentations en un programme. En particulier, l'ingénieur ne doit pas refaire toutes les preuves théoriques d'un domaine spécifique quand il applique ou utilise des méthodes formelles. Il faut que la théorie et les théorèmes de l'application soient dans un formalisme adéquat pour des développements ultérieurs et dans lequel la validité de la formalisation peut être testée aussi facilement que possible. Ce dont on a besoin est donc une interface entre le monde des applications et celui de la programmation. Les spécifications axiomatiques, en particulier les spécifications du LARCH, peuvent être cet interface.

## 1.2 Les spécifications Larch

### 1.2.1 Historique

Le projet LARCH a débuté avec la thèse de John Guttag en 1975, mais dès 1974, Jim Horning et lui constatent qu'une approche purement algébrique des spécifications n'était pas "pratique". Ils proposent alors une approche qui combine les spécifications algébriques et les spécifications opérationnelles qu'ils appelèrent les spécifications "dyadiques" [Gut74]. En 1980, ils développent la base des spécifications à deux niveaux (*two-tiered*), appellation introduite par Jeannette Wing dans sa thèse en 1983 au MIT [Win83]. En 1983, une première version du langage partagé du Larch fut publiée [Gut83], puis raffinée en 1985 [Gut86]. En 1990, certains outils pour LARCH sont disponibles pour vérifier et corriger des spécifications [Bou92] [Lau92], dont le plus important est le démonstrateur du LARCH, ou LARCH-PROVER, développé essentiellement par Stephen Garland et John Guttag au DEC System Research Center et au MIT. En 1992 a eu lieu le premier Workshop sur le LARCH, où nous avons présenté notre première expérience avec LP [Che92b]. Enfin, en 1993, l'essentiel de cette recherche du projet LARCH fut concentré dans [Gut93].

### 1.2.2 Deux niveaux de spécifications

Une spécification doit être d'une part, assez rigide pour exclure des implémentations non acceptables, et d'autre part assez souple pour permettre une implémentation élégante et efficace. Par conséquent, une spécification doit avoir suffisamment de contraintes mais ces contraintes doivent être indépendantes de l'implémentation. Une spécification "déclarative" liste les propriétés que doit avoir l'implémentation, par opposition à une spécification opérationnelle, qui produit une solution ayant les propriétés désirées sans forcement les décrire (Elle décrit une fonction en montrant comment on la calcule). Les avantages et les inconvénients des unes et des autres sont discutables en terme d'exécution, de rapidité, de modularité, de réutilisation, etc. Néanmoins, il est difficile de distinguer dans une spécification opérationnelle les propriétés essentielles de celles qui ne le sont pas, ceci dépend généralement du contexte où on évalue la spécification. Par contre, on peut toujours enrichir une spécification déclarative (axiomatique) afin d'exprimer de nouveaux besoins, et plus généralement de nouvelles propriétés.

*Larch* est une famille de langages de spécifications axiomatiques (ou langage de spécifications algébriques). Ces spécifications utilisent un langage formel, qui comprend une notation, une sémantique et une relation de satisfaction. Chaque spécification comprend une composante écrite dans un langage d'interface, LARCH INTERFACE LANGUAGE, dédié à un langage de programmation particulier et une composante écrite dans un langage formel, LARCH SHARED LANGUAGE, indépendant de toute implémentation. Dans ce document, nous utilisons exclusivement les

spécifications LSL.

Les éléments de base des spécifications LSL sont les *sortes* et les *opérateurs*. Les sortes sont des objets syntaxiques qui peuvent être des *types*, à la différence près qu'un type est généralement à la fois un ensemble de données et un ensemble de fonctions opérant sur ces données[2]. Les sortes sont des ensembles non vides disjoints de valeurs et les opérateurs sont des fonctions totales sur ces sortes. Le *Trait* est l'entité de base d'une spécification LSL.

**Exemple 1.1** *Le trait suivant est une axiomatisation des ensembles finis :*

```
Ens(Elem,Set):trait
introduces
 {}     :          -> Set
 insert : Elem,Set -> Set
 __∈__  : Elem,Set -> Bool

assert
 Set generated by {},insert
 Set partitioned  by ∈
 ∀ s:Set, e,e1,e2:Elem
    (e ∈ {})=false;
    e1 ∈ insert(e2,s) <=> (e1 = e2) \/ (e1 ∈ s)

implies
 insertgenerated ({} for empty)
 ∀ e,e1,e2:Elem, s:Set
    insert(e,s) ≠ {};
    insert(e,insert(e,s)) = insert(e,s);
    insert(e1,insert(e2,s)) = insert(e2,insert(e1,s))
converts ∈
```

*où* `insertgenerated(Elem,Set)` *est le trait suivant:*

```
insertgeneragted(Elem,Set):trait

introduces
 empty :         -> Set
 insert:Elem,Set -> Set
assert Set generated by empty,insert
```

Le mot clé `introduces` déclare une liste de fonctions en spécifiant leurs signatures. `assert` introduit un ensemble d'axiomes (équations) qui définissent certaines propriétés des opérateurs introduits. Une spécification LSL ne correspond pas forcément à un type de donnée abstrait dans le sens où l'ensemble des axiomes ne définit pas *toutes* les opérations qui caractérise le type[3].
Les constructeurs `generated by` et `partitioned by` permettent d'enrichir la théorie équationnelle du trait. La primitive `generated by` déclare une liste d'opérateurs, *Lops*, comme étant un ensemble complets de *générateurs* de la sorte. Chaque valeur de la sorte est équivalente à un terme construit en appliquant un nombre fini de fois les opérateurs appartenant à *Lops*. Cela définit donc un schéma d'induction très utilisé pour prouver des théorèmes par induction.
La primitive `partitioned by` déclare un ensemble complet d'*observateurs*[4] pour la sorte. Deux valeurs de la sorte peuvent être identifiées de façon unique par ces opérateurs. Par exemple, deux ensembles contenant les mêmes éléments sont égaux.

2. On peut associer à une sorte plusieurs types différents selon les fonctionnalités considérées
3. D'où la différence entre un trait et une spécification algébrique.
4. Ce sont des opérateurs dont le domaine est le sort identifié et le co-domaine est une autre sorte.

On peut aussi combiner deux traits. La théorie résultante est l'union des deux théories, c'est à dire l'union des parties `introduces` et `assert`.
Le trait `Ens(Elem,Set)` inclue le trait `insertgenerated(Elem,Set)`, où l'opérateur `empty` est "renommé" en {}. Ce dernier est un abstraction des propriétés usuelles des structures de données qui contiennent des éléments (ensemble, files, piles, etc), où la clause `generated by` spécifie que toute valeur de S peut être construite à partir de `empty` auquel on applique un nombre fini de fois `insert`.

Chaque trait définit une *théorie* dans la logique du premier ordre multi-sortée, une théorie étant un ensemble de formules et leurs conséquences logiques (équationnelles). Cette théorie est définie par les assertions du trait, les axiomes usuels de la logique du premier ordre et tout ce qui en découle mais rien de plus. Chaque formule apparaissant dans le trait est une formule de la théorie du trait. Cette interprétation *classe de modèles*[5] de la sémantique garantit que la validité d'une formule dans la théorie est due à la présence d'axiomes dans le trait mais jamais à leur absence. Ce qui est en opposition avec les langages de spécifications algébriques basés sur les algèbres *initiaux* ou *terminaux*. Ceci assure que tout les théorèmes prouvés pour une spécification incomplète restent valides si elle est étendue.

La notion de "correction" d'une spécification est une notion qui n'est pas complètement formalisable. En effet, la question de savoir si un système décrit par sa spécification possède le comportement désiré ne peut être formalisable que dans la mesure où l'on dispose d'une description précise (formelle) des propriétés attendues du système considéré. S'il semble aussi problématique de réussir à formaliser la notion de correction d'une spécification, il est néanmoins souhaitable de savoir au moins donner des critères de correction d'une spécification. Deux critères correspondent à des interrogations naturelles sur une spécification, la **consistance** et la **complétude**. Intuitivement, la complétude d'une spécification correspond à la question de savoir si "tout est spécifié" ou si les axiomes de la spécification "suffisent" à décrire le système, et la consistance correspond, quand à elle, à la question de savoir si les axiomes ne sont pas contradictoires.

La consistance est difficile à prouver et est indécidable en général. Par contre, l'inconsistance est plus facile à détecter. Chaque trait LSL doit être *consistant*: il ne doit pas définir une théorie contenant l'équation `vrai = faux`. Pour assurer un minimum de correction d'une spécification LSL, on peut spécifier des propriétés que l'on pense être vérifiées par cette spécification, à l'aide de la clause `implies`. Par exemple, le trait `Ens(Elem,Set)` "implique" (induit) qu'un ensemble dans lequel on a inséré un élément est nécessairement non vide. De façon générale, la clause `implies` permet donc d'exprimer ce qu'on pense être une conséquence de l'ensemble d'axiomes. En d'autres termes, cela permet de confirmer l'intuition de ce que la spécification doit vérifier. Cette clause est importante pour la vérification d'une spécification.

Un trait ne définit pas une théorie complète, puisque il ne ne définit pas nécessairement tous les opérateurs et que la plupart des théories que nous manipulons ne sont pas complètes. Néanmoins, la clause `converts` permet de vérifier une certaine notion de complétude, à savoir qu'un ensemble d'opérateurs est bien défini. Par exemple, le trait `Ens(Elem,Set)` affirme que $\in$ est complètement défini relativement à {} et à `insert`, c'est à dire que si l'on fixe l'interprétation de ces opérateurs, il existe une interprétation unique de $\in$ qui satisfait les axiomes.

Même si on ne peut prouver la correction d'une spécification, nous pouvons tout de même la vérifier, c'est à dire vérifier qu'elle possède de bonnes propriétés, en utilisant le démonstrateur du LARCH qui permet de vérifier une spécification en cours de conception. Nous verrons, dans le chapitre consacré aux preuves, comment vérifier, à l'aide du démonstrateur, les propriétés que doit satisfaire une spécification LSL, à savoir qu'elle n'est pas inconsistante (qu'elle n'est pas

5. Cette traduction de *loose* est empruntée à Michel Bidoit.

contradictoire), la théorie "containment" (qu'elle induit les bonnes propriétés) et la complétude relative (qu'un ensemble d'opérateurs est bien défini).

## 1.3 Le démonstrateur

**LP**, descendant de REVE [Les83], est un démonstrateur de théorème interactif basé sur la réécriture équationnelle dans le cadre de la logique du premier ordre [Gar91]. La conception et le développement de LARCH démonstrateur ont été initialement motivés par la vérification des spécifications LSL [Gar90] [Gar88a], mais son utilisation s'est étendue aux preuves de circuits matériels ou d'algorithmes mettant en jeu la concurrence [Sta89], [Gar88b], [Che92b].

Ses concepteurs le définissent plutôt comme un assistant de preuve ou un *correcteur de preuve*, dû fait qu'il n'effectue pas de façon automatique des preuves difficiles et qu'il n'utilise pas non plus d'heuristiques, comme le système NQTHM de Boyer & Moore [Boy88], pour générer des conjectures utiles à la preuve. LARCH et LP sont conçus suivant le principe que tout doit être aussi simple que possible et que la première tentative pour effectuer une preuve échoue dans la plupart des cas. Dans les premiers temps de la conception de LP [Gut86], une des questions que se posaient John Guttag et Jim Horning était de savoir sous quelle forme un diagnostic sur la preuve doit être fourni aux utilisateurs qui ne sont pas familiers avec la structure interne du démonstrateur ni avec la théorie sous jacente.

LP ne fournit pas de procédure de décision, mais plutôt une notation et des techniques axiomatiques pour définir et utiliser les règles de réécriture. Toutes les preuves sont effectuées en appliquant les règles de réécriture, les preuves par cas, l'induction, la contradiction et l'application des règles de déduction. De même, les dérivations ou les conséquences logiques sont obtenues en appliquant les règles de réécriture ou les règles de déduction. Par conséquent, l'utilisateur de LP doit guider le système, même si LP peut effectuer de façon automatique des preuves triviales mais fastidieuses.

En LP, une preuve est basée sur un *système logique* ou *théorie*. Une système logique est un ensemble d'équations, de règles de réécriture, d'opérateurs, de règles d'induction et de règles de déduction, formulés dans une logique du premier ordre multi-sortée avec égalité. Une théorie LP correspond à une spécification LSL, mis à part que les axiomes en LP ont, en plus d'un contenu sémantique, un contenu opérationnel.

Le moteur d'inférence de base du démonstrateur est la réécriture équationnelle [Jou87]. Les axiomes introduits sont orientés en règles de réécriture, en utilisant soit un ordre pré-défini de LP, soit un ordre défini explicitement, tel un ordre polynômial. LP fournit des règles de réécriture pré-définies pour simplifier les formules construites avec les opérateurs booléens pré-définis `=>` , `\/` , `/\`, etc.

### 1.3.1 Le système logique

Par analogie à un langage de programmation, mis à part la sorte `Bool` et les opérateurs logiques associés, `=>`, `<=>` , `/\`, `\/` et `~`, tout ce qu'on utilise dans un script LP doit être déclaré : les sortes, les opérateurs et les variables.

**Exemple 1.2** *La déclaration suivante définit deux sortes* `Elem` *et* `Set` *et quelques opérateurs avec leurs signatures :*

```
declare sorts Elem, Set
declare variables e, e1, e2: Elem, x, y, z: Set
declare operators
```

```
{}    :            -> Set
{__}  : Elem       -> Set
insert: Elem, Set -> Set
__\u__ : Set, Set -> Set
__\in__:Elem, Set -> Bool
__\<__:Set, Set -> Bool
..
```

Les axiomes sont introduits sous forme d'*équations* ou de *formules*. Une équation n'est autre qu'une formule avec deux termes reliés par un symbole d'égalité $t_1 = t_2$ et une formule n'est autre qu'une abréviation d'un équation dont le membre droit est *true* ($F$ est logiquement équivalente à `F <=> true`). Les axiomes, introduits sous forme d'équations (ou de formules), sont orientés en règles de réécriture, qui seront utilisées pour réécrire tout les faits (règles) **non immunisés** présents dans le système courant.

**Exemple 1.3** *Les axiomes suivants sont une axiomatisation des ensembles finis :*

```
set name Ensemble
 assert
   sort Set generated by {}, insert;
   {e} = insert(e, {});
   ~ (e \in {});
   e \in insert(e1, x) <=> e = e1 \/ e \in x;
   {} \< x;
   insert(e, x) \< y <=> e \in y /\ x \< y;
   e \in (x \u y) <=> e \in x \/ e \in y
   ..
```

**Remarque 1.1** *Les axiomes portent une étiquette formée par le nom sous lequel ils ont été introduits suivi d'un numéro correspondant à l'ordre d'introduction dans le système. Les axiomes précédents portent l'étiquette* `Ensemble.1`,`Ensemble.2`,...,`Ensemble.7`.

**Inconsistance** LP considère la formule `false = true` comme étant inconsistante et plus généralement une formule de la forme `~(x=t)`, où `t` est un terme ne contenant pas la variable `x` (ou bien `b = true`, où `b` est une variable de la sorte `Bool`). Par conséquent, toute sorte LP contient au moins un élément et la sorte `Bool` des booléens contient deux éléments distincts `true` et `false`.

### 1.3.2 Quantification

L'une des extensions fondamentales entre la précédente version du démonstrateur (2.4) et l'actuelle (3.1) est l'introduction des quantificateurs. En effet, les quantificateurs universels et surtout existentiels augmentent d'une façon considérable la puissance d'expression du démonstrateur.
Les termes LP sont définis de façon usuelle :

- Toute variable et toute constante est un terme.
- Pour tout symbole de fonction d'arité $n > 0$, $f(t_1, \ldots, t_n)$ est un terme.

Les termes peuvent être de différentes formes syntaxiques : fonctionnels (`suc(x)`), infixés (`x + y`), préfixés (`-x`), postfixés (`x.last`), etc.
De même, une formule est définie comme suit :

- Si $P$ est un symbole de prédicat (dont le co-domaine est `Bool`) d'arité $n \geq 0$, $P(t_1, \ldots, t_n)$ est une formule.

- Si $F_1$ et $F_2$ sont des formules, alors ($F_1$ /\ $F_2$), ($F_1$ \/ $F_2$), ($F_1$ => $F_2$), ($F_1$ <=> $F_2$), ~$F_1$ sont des formules.

- Pour toute variable $x$ et toute formule $F$, \E $x$ $F$ et \A $x$ $F$ sont des formules.

LP fournit un ensemble de règles pré-définies de simplificationn des formules quantifiées (manipulation des variables libres et mise en forme *prenex*).

**Exemple 1.4** *Nous pouvons introduire deux axiomes supplémentaires à la spécification* `Ensemble`*:*

```
set name Ax_suppl
 assert
   \A e (e \in x <=> e \in y) => x = y                    %Ax_suppl.1
   \E x \A e (e \in x <=> e = e1 \/ e = e2)               %Ax_suppl.2
```

*pour exprimer les propriétés suivantes: "deux ensembles dont les éléments sont identiques sont égaux" et "pour tout élément e1 et pour tout élément e2, il existe une ensemble qui contient uniquement ces deux éléments".*

Les variables quantifiées doivent être préalablement déclarées. Par ailleurs, LP fournit des règles de manipulation des quantificateurs, pour l'*élimination* et l'*introduction.*

- $\forall$*−elimination* : `instantiate` $x$ `by` $t$ `in` *nom*, permet d'éliminer les quantificateurs universels dans les formules *nom* en substituant à toutes les occurrences de $x$ (liées par un de ces quantificateurs) le terme $t$. Cela correspond à la règle d'inférence suivante, où $P$ ne contient pas de quantificateur :

$$\frac{\forall x P(x)}{P(t)}$$

- $\exists$ *− elimination* : `fix` $x$ `as` $t$ `in` *nom*, permet d'éliminer le quantificateur existentiel des formules *nom* en substituant à toutes les occurrences de la variable quantifiée $x$ le terme $t$, où $t$ est de la forme `sk(x1,...,xn)` avec `sk` un nouveau symbole de fonction et `x1,...,xn` sont les variables liées par le quantificateur existentiel.

$$\frac{\exists x P(x, x_1, \ldots, x_n)}{P(sk(x1, \ldots, xn), x_1, \ldots, x_n)}$$

Nous verrons dans la suite de ce chapitre, les méthodes de preuves associées aux quantificateurs, qui correspondent aux règles d'*introduction*, la *généralisation* ($\forall$*−introduction*) et la *spécialisation* ($\exists$ *− introduction*).

### 1.3.3 Axiomes particuliers

Certains axiomes permettent d'enrichir les théories que l'on définit en LP, sans alourdir pour autant les spécifications. Ces axiomes, qui peuvent être donc prouvés comme tout autre axiome, permettent d'écrire des spécifications concises et claires, ce qui très utile lorsqu'on manipule des bases de règles de taille importante.

### Les théories associatives et commutatives

LP permet de définir des théories associatives et/ou commutatives en déclarant des opérateurs comme étant associatifs et/ou commutatifs, ce qui permet à LP de réécrire les termes modulo la commutativité ou l'associativité. L'assertion `assert ac +` définit + comme étant un opérateur associatif et commutatif. Cette assertion est logiquement équivalente aux deux équations usuelles définissant l'associativité et la commutativité.
Les opérateurs pré-définis de LP, `/\`, `\/`, et `<=>` , sont définis comme étant des opérateurs associatifs et commutatifs. LP définit l'opérateur, `=:S,S -> Bool`, et le déclare automatiquement pour chaque sorte `S` comme étant commutatif si la sorte `S` est différente de la sorte `Bool`.

### Les règles d'induction

**Induction structurelle** Un règle d'induction (schéma d'induction) est logiquement équivalente à un ensemble infini de formules, utilisée par LP pour effectuer des preuves par induction. On peut définir plusieurs schémas d'induction pour la même sorte.

**Exemple 1.5** *On peut considérer les ensembles finis comme étant construits à l'aide de l'ensemble vide et de l'opérateur d'insertion (axiome* `Ensemble.1`*), ou bien à l'aide de l'ensemble vide, du singleton et de l'union :*

```
set name Ens_ind2
 sort Set generated by {}, {__}, \u
```

De façon générale, un schéma d'induction structurelle de la forme `Sort` $S$ `generated by` $g_1$, $g_2$, ...,$g_n$ induit que les générateurs de la liste ont $S$ pour co-domaine. Si aucun des domaines d'un $g_i$ n'est $S$, $g_i$ est alors un générateur de base (`{}`,`{__}` dans le schéma précédent), sinon c'est un générateur inductif.
Si on spécifie l'option `freely`, on ajoute alors un ensemble de formules qui spécifient que les opérateurs ont des co-domaines disjoints deux à deux.

**Exemple 1.6** *La règle suivante définit un schéma d'induction pour les entiers :*

```
set name ind_nat1
sort Nat generated freely by 0, s
```

*Ce schéma est équivalent à l'ensemble infini de formules*

$$P(0) \wedge \forall x(P(x) \Rightarrow P(s(x))) \Rightarrow \forall x P(x)$$

*enrichi des deux axiomes*[6] *:*

$$s(x) = s(y) \equiv x = y$$
$$\neg(0 = s(x))$$

*Un exemple simple de l'utilité des ces axiomes est la preuve de* $\neg(x = s(x))$. *Supposons qu'on ait les déclarations suivantes :*

```
set name Nat
declare sort Nat
```

6. Dans la version (2.4) de LP, ces axiomes devaient être explicitement introduits.

```
declare variables x, y, z,i,j,n,m: Nat
declare operators
  0, 1, 2                    :            -> Nat
  s                          : Nat        -> Nat
  __+__                      : Nat, Nat -> Nat
  __<=__,__<__,__>=__,__>__: Nat, Nat -> Bool
  ..
assert
  ac +;
  sort Nat generated  by 0, s;
  i + 0 = i;
  i + s(j) = s(i + j);
  ..
```

*La preuve de* $\neg(x = s(x))$ *par induction nécessite pour le cas de base* `~(s(0) = 0)`, *l'axiome* $\neg(0 = s(x))$, *et pour le pas d'induction* `~(s(s(xc)) = s(xc))`, *l'axiome* $\neg(x = s(x))$.

**Induction bien fondée** Un deuxième type d'induction peut être défini en LP, à savoir une induction basée sur une relation binaire bien fondée $R$ (nothérienne). Cette règle d'induction est utilisée dans les preuves par "induction bien fondée".

```
assert well-founded R
```

L'opérateur `R` doit avoir la signature `S,S -> Bool` pour une sorte `S`. Les règles d'induction bien fondée sont logiquement équivalentes à un ensemble infini de formules de la forme

$$\forall x(\forall y(R(y,x) \Rightarrow P(y)) \Rightarrow P(x)) \Rightarrow \forall x P(x)$$

où $P$ est une formule quelconque du premier ordre. Si $R$ est la relation $<$ sur les entiers, ce schéma permet de montrer $\neg P(x)$ s'il existe une valeur $y$ inférieure à $x$ qui ne vérifie pas la propriété.

Nous verrons dans les chapitres suivants l'utilisation de ces deux types d'induction pour les preuves. Enfin, il faut noter que l'on peut prouver les schémas d'induction.

### Les règles de déduction

Une règle de déduction est un objet de la forme `when` $h_1,h_2,\ldots,h_n$ `yield` $c_1,c_2,\ldots,c_m$, logiquement équivalente à la formule $(h_1 \wedge h_2 \wedge \ldots \wedge h_n) \Rightarrow (c_1 \wedge c_2 \wedge \ldots \wedge c_m)$.
Du point de vue opérationnel, une règle de déduction diffère d'une formule dans le sens où elle est utilisée par LP pour inférer de nouvelles équations à partir de celles présentes dans le système.

**Exemple 1.7** *L'axiome* `Ax_suppl.1` *de la spécification* `Ensemble` *peut s'écrire :*

```
when \A e (e \in x <=> e \in y) yield x = y
```

*Si on applique cette règle à la formule:*

```
e1 \in x <=> e1 \in (x \u x)
```

*on obtient la formule* `x = x \u x`.

**Cas particulier** Les règles de déduction permettent de définir en LP un ensemble d'*observateurs* pour une sorte `S` (cf. paragraphe1.2.2). La règle `sort` *S* `partitioned by` *liste_op* permet d'exprimer que deux objets de la sorte *S* sont égaux si on peut les comparer avec les opérateurs de *liste_op*.

**Exemple 1.8** *Pour une axiomatisation du type de donnée* pile, *nous avons la règle suivante :*

```
sort pile partitioned by estvide, top, pop
```

*traduite par* LP *en une règle de déduction :*

```
when estvide(p1) = estvide(p2), top(p1) = top(p2), pop(p1) = pop(p2)
yield p1 = p2
```

### 1.3.4 Les ordres de réécriture

En LP, si l'option `automatic-ordering` est positionnée, toute équation introduite est automatiquement orientée en une règle de réécriture, sinon l'orientation doit être demandée explicitement avec la commande `order` ou avec la complétion. L'ordre utilisé est soit l'ordre par défaut `noeq-dsmpos` ou un ordre choisi par l'utilisateur, `dsmpos`, `polynomial` (basé sur une interprétation des symboles de fonctions fournie par l'utilisateur [BC87], [BC86]), ou bien un ordre "brutal" pour lequel la terminaison du système n'est pas garantie, tel que `either-way` (les équations sont orientées de gauche à droite si possible, sinon de droite à gauche si possible, sinon elles demeurent sous formes d'équations), `left-to-right`(de gauche à droite si possible sinon il les garde sous formes d'équations), `manual` (ordre interactif où LP demande à l'utilisateur pour chaque équation le sens d'orientation, et ceci si l'option `automatic-registry` n'est pas positionnée).

Les ordres `dsmpos` et `noeq-dsmpos` (ce dernier ne permet pas que deux symboles aient la même précédence) sont deux ordres "enregistrés" (`registred-ordering`) basés sur les ordres *Rpo* [Der92]. Ils garantissent la terminaison en l'absence d'opérateurs associatifs-commutatifs. Ces ordres utilisent des informations contenues dans un "enregistrement"[7] : la *précédence* et le *statut* des symboles de fonction. Ce registre est étendu par LP en cours de traitement lorsqu'il oriente une formule. Si l'option `automatic-registry` n'est pas positionnée, LP fait des suggestions à l'utilisateur qui doit choisir une extension possible. L'utilisateur a aussi la possibilité d'étendre le registre en proposant des informations sur le statut ou la précédence. Ces informations sont acceptées et enregistrées si elles ne contredisent pas les informations présentes dans le registre.

### 1.3.5 L'immunité et la passivité

L'immunité et la passivité permettent à l'utilisateur de contrôler l'application de certaines règles problématiques, pour effectuer des preuves complexes. En effet, tout nouveau fait introduit sera réécrit (normalisé) en utilisant le système de réécriture courant, puis orienté en une règle de réécriture avec l'ordre courant. Cette nouvelle règle sera alors utilisée pour simplifier toutes les autres règles de réécriture (axiome, théorème, règle de déduction). Parfois, il est utile de "protéger" certains faits pour qu'ils ne soient pas simplifiés, c'est à dire les immuniser contre la simplification. De même, il est parfois utile de ne pas considérer certaines règles du système dans le processus de normalisation, auquel cas, on peut les "dés-activer".

7. Nous utiliseront "registre" plutôt qu"enregistrement" comme traduction de `registry`

**Immunité** Dès que l'option `immunity` est positionnée, l'immunité est appliquée aux faits introduits comme axiomes, aux conjectures à prouver, aux paires critiques calculées, etc, et en général à tout nouveau fait déduit à partir de cet instant là.

LP simplifie (réécrit) de façon automatique tous les termes à leur forme normale[8], dans les formules, les règles de réécriture, ou bien dans les règles de déduction, et applique toutes les règles de déduction *actives* aux formules et règles de réécriture non immunisées. Par contre, toute formule ou règle immunisée ne pourra être simplifiée que par une commande explicite `normalize` ou `rewrite` et ne sera utilisée pour la simplification qu'avec la commande explicite `apply`.
Les formules introduites sous l'option, `set immunity on`, conservent leur immunité après leur orientation en règle de réécriture. Lorsqu'on essaye de prouver un théorème sous l'option d'immunité, LP le réécrit et le simplifie pour les besoins de la preuve mais dès la preuve terminée, le théorème est conservé immunisé sous sa forme originale.
Lorsque l'option d'immunité n'est pas positionnée, on peut immuniser explicitement des faits présents dans le système avec la commande `make immune nom`, où `nom` peut être le nom d'une seule règle, où bien d'un ensemble de règles.

L'immunité est très utile lorsqu'on veut conserver certains faits dans le système sous leur forme originale pour la lisibilité, ou bien, lors d'un calcul de paires critiques, certaines règles de réécriture peuvent être immunisées pour qu'elles ne soient pas simplifiées (par exemple, réduites à des identités). De plus, on peut immuniser certains faits que l'on sait irréductibles, pour améliorer les performances du système, car LP n'essayera pas de les simplifier avec les nouvelles règles qui seront introduites.

Néanmoins, cette possibilité doit être utilisée de façon rigoureuse car elle peut engendrer des systèmes non confluents et/ou ralentir le système avec un nombre considérable de règles de réécriture redondantes, et diminuer ainsi les performances du système à cause du processus de normalisation.

**Passivité** Tous les faits introduits dans le système sont par défaut *actifs* (LP utilise toute règle active pour réduire les termes à leur forme normale (cf note 8) et applique toute règle de déduction active pour déduire des conséquences d'une formule), à moins que l'option `activity` ne soit positionnée à `off`. Dans ce cas, les axiomes introduits et les théorèmes prouvés seront "passifs", non utilisés pour toute réécriture ultérieure, ou déduction. On peut changer l'activité d'une règle ou d'un ensemble de règles avec les commandes `make passive` *nom* (désactiver) et `make passive` *nom* (activer). L'utilisation d'une règle de réécriture (respectivement règle de déduction) passive est alors explicitement invoquée avec les commandes `normalize` ou `rewrite` (respectivement `apply`). Les faits demeurent passifs quand ils sont normalisés ainsi que les formules lorsqu'elles sont orientées en règles de réécriture.

La passivité est utile pour améliorer les performances du système lorsqu'on sait que certains faits ne peuvent pas être appliqués. Par exemple, les règles de réécriture passives ne seront pas utilisées pour réécrire tout fait introduit.

## 1.4 Les méthodes de preuve

Les deux principaux mécanismes de preuve en LP sont l'inférence *en avant*, qui consiste à produire des conséquences du système logique afin de simplifier la conjecture[9] et l'inférence

8. si elle existe par rapport au système de réecriture courant

9. En LP, une conjecture à prouver est sous forme d'équation $t_1 = t_2$, une formule $F$ étant équivalente à l'équation $F = true$.

*arrière* (guidée par le but), qui produit un ensemble de buts à satisfaire. Pour chaque type de mécanisme, il fournit un ensemble de méthodes. La description qui suit n'est pas exhaustive, et n'entend pas être un manuel utilisateur pour LP. Nous renvoyons le lecteur à [Gar89] pour une description succincte et à [Gar91] pour une description détaillée. Nous allons décrire, et éventuellement détailler, certaines méthodes de LP qui nous semblent intéressantes ou que nous utilisons fréquemment dans le suite de ce document.

Les mécanismes d'inférence en avant sont les méthodes liées au moteur d'inférence du démonstrateur. LP, basé sur les systèmes de réécriture, fournit de façon explicite les mécanismes de réécriture, normalisation, calcul des paires critiques, et complétion.

Le mécanisme d'inférence arrière correspond à la méthode usuelle utilisée pour prouver une conjecture, c'est à dire à un raisonnement guidé par le but avec la commande `prove` *assertion* `by` *methode*. LP génère alors des sous buts suffisants pour prouver la conjecture. Quand la méthode n'est pas spécifiée, LP utilise la (ou les) méthode(s) de la liste des méthodes à utiliser par défaut, positionnée avec la commande `set Proof-methods` *list-meth*. Si cette liste n'est pas positionnée, la méthode par défaut est la normalisation.

Il existe plusieurs types de méthodes d'inférence arrière en LP :

- Les méthodes standards telles que *la preuve par cas*, *la preuve par contradiction*, et *la preuve par induction*.
- Les méthodes qui dépendent de la forme syntaxique de la conjecture, telles que l'*implication*, l'*équivalence*, et la *conjonction*.
- Les méthodes pour manipuler les quantificateurs, à savoir la *généralisation* et la *spécialisation*.

Nous allons décrire en premier certaines méthodes d'inférence en avant car elles sont utilisées dans les méthodes d'inférence arrière.

### 1.4.1 Les mécanismes de réécriture

- La *réécriture* explicite permet de contrôler manuellement l'application des règles de réécriture. On peut réécrire explicitement une règle (ou bien une formule, une règle de déduction, une conjecture) en utilisant une règle particulière ou un ensemble de règles. Cette méthode est très utile dans deux contextes particuliers : lorsqu'on utilise l'immunité et l'activité des règles, pour prouver un théorème particulier pour "déplier" sa définition en utilisant des règles dés-activées, ou bien pour réécrire un fait en utilisant une règle de réécriture comme étant orientée dans le sens inverse (`rewrite` nom1 `with reversed` nom2).
  La normalisation explicite est similaire à la réécriture (qui permet de faire un seul pas de réécriture), sauf que l'on utilise une ou un ensemble de règles, pour réécrire une ou plusieurs règles afin d'obtenir la forme normale respectivement à l'ensemble de règles utilisé.
- Le *calcul de paires critiques* permet de prouver une conjecture en la normalisant à une identité. On peut préciser les règles dont on veut calculer les paires critiques, LP oriente les paires non triviales en règles de réécriture, les ajoute à sa base de règles courantes pour réécrire la conjecture. Le processus s'arrête dès que la conjecture est normalisée en une identité. Les paires critiques non triviales calculées dans une preuve sont éliminées dès que la preuve est achevée. De plus, le calcul de paires critiques est effectué par LP pour la procédure de complétion.

- *Application de règles de déduction*: de même que pour la réécriture et la normalisation explicite, on peut appliquer explicitement une règle de déduction (éventuellement désactivée) à des formules ou à des règles de réécriture.

**Exemple 1.9** *Ayant les règles suivantes,*

```
Ensemble.7: e \in (x \u y) -> e \in x \/ e \in y
Ax_suppl.1: when \A e (e \in x <=> e \in y) yield x = y
```

*on peut prouver* `x \u x = x` *en appliquant la règle* `Ax_suppl.1`

### 1.4.2 Les méthodes standards

#### La preuve par induction

**Induction structurelle** La commande `prove` $F$ `by induction on` $x$ `using` $R$ permet à LP d'effectuer une preuve de la formule $F$ par induction structurelle sur la variable $x$ d'une sorte $S$ en utilisant la règle d'induction $R$. LP construit alors un ensemble de sous-buts correspondants au cas de base et un ensemble de buts pour le pas d'induction.
Le cas de base consiste à prouver un ensemble de formules $F\prime$, obtenues en remplaçant dans $F$ la variable $x$ par un générateur de base. Par conséquent, on a autant de sous-buts de base que de générateurs de base définis par la règle d'induction $R$. LP renomme les variables de $F$ appartenant à une autre sorte que $S$. Une fois prouvé le(s) sous but(s) de base, LP introduit de nouvelles constantes de la sorte de la variable d'induction, ajoute les hypothèses d'induction à son système logique courant (contexte de la preuve), où la variable d'induction a été remplacée par une constante, et essaye de prouver une formule $F\prime$, obtenue en remplaçant dans $F$ la variable $x$ par les générateurs non basiques appliqués à ces constantes.

**Exemple 1.10** *Pour montrer le théorème* $x \subseteq (x \cup y)$ *en utilisant la spécification* `Ensemble`, *les principales étapes sont les suivantes:*

```
prove x \< (x \u y) by induction on x using Ens_ind2
```

*Cette commande introduit la conjecture et la méthode de preuve. En réponse,* LP *construit les sous buts de base:*

```
{} \< ({} \u y), {e} \< ({e} \u y)
```

*introduit deux nouvelles constantes* `xc`, `xc1`, *et les hypothèses d'induction:*

```
xc \< (xc \u y), xc1 \< (xc1 \u y)
```

*et essaye de prouver le but correspondant au pas d'induction:*

```
(xc \u xc1) \< (xc \u xc1 \u y)
```

**Profondeur d'une preuve par induction** Parfois, il est nécessaire d'utiliser plusieurs niveaux d'induction. La commande `prove` $F$ `by induction on` $x$ `depth` $n$ `using` $IR$, permet à LP de prouver la formule $F$ en utilisant la règle $IR$ avec une induction structurelle de profondeur $n$. Quand la profondeur n'est pas spécifiée, elle est égale à 1. En général, LP construit les sous buts de base et le pas d'induction en utilisant l'ensemble de générateurs défini par la règle $IR$.

**Exemple 1.11** *Prouver une formule* `F(x)` *avec une induction de profondeur 2 en utilisant la règle* `ensemble.1` *revient à prouver deux sous buts de base:*

```
F({})
F(insert(x,{})
```

*puis à poser deux hypothèses d'induction :*

```
F(xc)
F(insert(x,xc))
```

*pour prouver le but*

```
F(insert(x,insert(y,xc)))
```

**Induction bien fondée** L'induction bien fondée utilise une relation binaire $R$, qui a été prouvée ou simplement définie comme étant bien fondée sur une sorte S. Cette induction utilise une seule hypothèse d'induction $R(x, xc) \Rightarrow F(x)$, pour prouver le but $F(xc)$, où $xc$ est une nouvelle constante de la sorte S.

**Exemple 1.12** *Pour les entiers naturels, définis par la spécification* `Nat` *(cf. exemple 1.6), on pose l'axiome suivant :*

```
set name ind_nat2
 assert well founded <
```

*Pour prouver la formule suivante* `0 + x = x` *en utilisant cette règle,* LP *construit l'hypothèse d'induction* `x < xc => 0 + x = x` *et le but* `0 + xc = xc`.

### La preuve par cas

La preuve par cas consiste à simplifier la preuve d'une formule en procédant par cas. Si pour chaque cas, la formule se simplifie à `true`, alors c'est un théorème. Une analyse par cas permet de contourner l'incomplétude de la définition d'un certain type. En effet, si la preuve réussit, alors les axiomes de définition étaient incomplets, d'où le traitement par cas. Si la preuve échoue, alors l'analyse par cas permet de détecter le cas critique, et de comprendre la raison de l'échec.

`prove` $F$ `by cases` $F_1, \ldots, F_n$

La liste des cas $F_1, \ldots, F_n$ doit être exhaustive, elle doit couvrir tous les cas possibles. Par conséquent, la preuve comprend $(n + 1)$ sous-buts, le premier sous-but généré par LP est $F_1 \backslash/ F_2 \backslash/ \ldots \backslash/ F_n$. Si $n = 1$ alors les deux sous-buts générés sont $F_1$ et $\sim F1$.
Pour chaque cas $i$, LP produit un but $F\prime$ et une hypothèse $F_i\prime$, en substituant les variables libres de $F$ et de $Fi$ par des nouvelles constantes[10]. Si une inconsistance est déduite lorsqu'une hypothèse $F_j\prime$ est ajoutée au système logique courant, alors ce cas est considéré comme impossible et le but correspondant est trivialement *vrai*.

**Remarque 1.2** *Les hypothèses sont toujours nommées* < *nom* >`CaseHyp.`< *numero* >, *où nom est le nom du système courant, c'est à dire le nom sous lequel la preuve est effectuée. Nous verrons dans le chapitre consacré aux preuves, l'utilité de ces objets.*

### La preuve par contradiction

Prouver une formule $F$ par contradiction (ou par l'absurde) consiste à ajouter $\neg F\prime$ au système logique courant pour déduire un inconsistance, c'est à dire la formule *false*. $F\prime$ est déduite de $F$ en substituant aux variables libres de $F$ des nouvelles constantes (ce qui revient à remplacer les

10. Le remplacement par de nouvelles constantes est dû au fait que les variables apparaissant dans les formules $F_i\prime$ ne sont pas quantifiées de façon universelle mais représentent les mêmes variables que celles qui apparaissent dans les $F_i$. Or les variables libres dans une formule LP sont implicitement quantifiées de façon universelle [Gar91]

variables implicitement quantifiées universellement par des variables existentielles). Le nom associé à l'hypothèse de contradiction suit la même syntaxe que précédemment, le mot clé `ContraHyp` remplaçant `CaseHyp`.

### La preuve par généralisation

La preuve par généralisation permet de montrer une conjecture qui contient un quantificateur universel. Prouver une formule $F$ par généralisation consiste à éliminer le quantificateur universel de la formule pour prouver une instance obtenue en substituant toutes les occurrences de la variable quantifiée par un "nouveau" terme $t$ (qui n'apparaît dans aucun fait de la base). Le choix du terme $t$ obéit aux mêmes restrictions que pour l'élimination de quantificateur existentiel par `fix` (cf. paragraphe 1.3.2). Ce traitement correspond à la règle suivante,

$$\frac{P(t)}{\forall x P(x)}$$

**Exemple 1.13** *Pour une constante* `c` *donnée, la commande*

```
prove \A x (f(x) = c) by generalizing x from d
```

*est possible lorsque* `d` *est une nouvelle constante et le but à montrer est alors*

```
f(d) = c
```

### La preuve par spécialisation

La preuve par spécialisation permet de montrer une conjecture qui contient un quantificateur existentiel. Prouver une formule $F$ par spécialisation consiste à éliminer un quantificateur existentiel (accessible) de la formule en substituant toutes les occurrences de la variable quantifiée par un terme $t$.

**Exemple 1.14** *Avec la spécification des ensembles, nous pouvons prouver l'axiome* `Ax_suppl.2` *(cf. paragraphe 1.3.2) :*

```
prove \E x \A e (e \in x <=> e = e1 \/ e = e2)
 resume by specializing x to insert(e2, {e1})
```

*où nous avons "choisi" pour $x$, l'ensemble $\{e1, e2\}$.*

*De même, soit la spécification des entiers naturels* `Nat` *où nous avons les deux schémas d'induction* `ind_nat1` *et* `ind_nat2`*, et où nous définissons* `<=` *(respectivement* `<`*) comme un ordre total (respectivement ordre total strict). Nous pouvons montrer que $\forall n \exists m(n < m)$, en prenant $m$ comme étant $s(n)$.*

```
prove \A i \E j (i <= j)
  resume by specializing j to s(i)
qed
```

### 1.4.3 Les méthodes syntaxiques

Les méthodes syntaxiques sont celles qui sont liées à la forme syntaxique de la formule à prouver.

- Preuve d'une *conjonction*: cette méthode permet de réduire le coût de la réécriture équationnelle des termes lorsque la conjecture est de la forme $t_1/\backslash t_2/\backslash \ldots/\backslash t_n$. Elle consiste à prouver séparément chaque sous-but $t_i$.
  Nous verrons que cette méthode ne doit pas être utilisée de façon systématique, en particulier si tous les sous-buts admettent une même structure de preuve.

- Preuve d'une *implication*: une implication $A \Rightarrow B$ étant équivalente à $\neg A \vee B$, la preuve d'une implication peut se ramener à une preuve par cas $A$ et $\neg A$. Par conséquent, LP prouve une implication de la forme `t1 => t2` en ajoutant l'hypothèse `t1'` pour montrer le but `t2'`, où `t1'` et `t2'` sont obtenus à partir de `t1` et `t2` comme pour une preuve par cas. L'hypothèse est nommée comme précédemment avec le mot clé `ImpliesHyp`.

- Preuve d'une *équivalence*: la preuve se ramène à prouver une double implication en utilisant la méthode précédente.

- Preuve d'une *conditionnelle*: LP permet de définir des axiomes conditionnels de la forme
  `if` $t_1$ `then` $t_2$ `else` $t_3$.
  La preuve d'une règle conditionnelle se ramène à une preuve par cas, où les deux cas sont $t_1$ et $\neg t_1$. LP ajoute l'hypothèse $t_1'$ pour prouver $t_2'$ puis $\neg t_1'$ pour prouver $t_3'$. Les $t_i'$ sont obtenus à partir des $t_i$ suivant le même principe que pour la preuve par cas. Les conjectures auxquelles ont peut appliquer cette méthode sont de la forme
  `if` $t_1$ `then` $t_2$ `else` $t_3$ ou bien `if` $t_1$ `then` $t_2$ `else` $t_3$ `=` $t_4$,
  si le symbole de tête du terme $t_4$ n'est pas `if`. Les hypothèses sont nommées comme précédemment avec le mot clé `IfHyp`.

**Exemple 1.15 (récapitulatif)** *Soit la spécification des ensembles finis* `Ensemble`*, et soit le théorème* $e \in x \wedge x \subseteq y \Rightarrow e \in y$*. La preuve de ce théorème utilise une induction sur* $x$*, l'implication, une preuve par cas et enfin une complétion. Nous avons les axiomes (2..7), orientés par* LP *en règles de réécriture en utilisant l'ordre prédéfinis* `noeq-dsmpos`*, et l'axiome* `Ax_suppl.1`*:*

```
Induction rules:
Ensemble.1: sort Set generated by {}, insert

Rewrite rules:
Ax_suppl.1: \A e (e \in x <=> e \in y) => x = y -> true
ensemble.2: {e} -> insert(e, {})
ensemble.3: e \in {} -> false
ensemble.4: e \in insert(e1, x) -> e = e1 \/ e \in x
ensemble.5: {} \< x -> true
ensemble.6: insert(e, x) \< y -> e \in y /\ x \< y
ensemble.7: e \in (x \u y) -> e \in x \/ e \in y
```

*La preuve débute par les commandes:*

```
set name theoreme
prove e \in x /\ x \< y => e \in y by induction on x
```

*En réponse,* LP *produit le sous-but de base,*

```
Subgoal 1:e \in {} /\ {} \< y => e \in y
```

*introduit une nouvelle constante* `xc`, *construit l'hypothèse d'induction*

```
theoremeInductHyp.1: e \in xc /\ xc \< y => e \in y
```

*et le but*

```
Subgoal 2: e \in insert(e1, xc) /\ insert(e1, xc) \< y => e \in y
```

*Intuitivement, on veut montrer* `e \in y` *avec les hypothèses* `e \in insert(e1, xc)` *et* `insert(e1, xc) \< y`. *Il suffit alors de poursuivre par implication,* `resume by =>-m`. LP *introduit alors trois constantes* `e1c, ec, yc`, *posent les trois hypothèses*

```
theoremeImpliesHyp.1.1: ec = e1c \/ ec \in xc
theoremeImpliesHyp.1.2: e1c \in yc
theoremeImpliesHyp.1.3: xc \< yc
```

*et le but* `ec \in yc`, *où la deuxième hypothèse résulte de la réécriture de* `insert(e1, xc) \< y` *avec la règle* `Ensemble.6`.
*Intuitivement, comme* `ec` *appartient à* `insert(e1c,xc)`, *il faut prouver le but, suivant que* `ec = e1c` *ou bien que* `ec` *appartient à* `xc`. *Nous poursuivons alors par cas,* `resume by case ec = e1c`. *Il suffit de donner un seul cas, le deuxième sera généré automatiquement par* LP. *En effet, il posera* `~(ec = e1c)`, *et simplifiera l'hypothèse* `theoremeImpliesHyp.1.1` *en* `(false \/ ec \in xc)` *soit* `(ec \in xc)`. *La preuve par cas produit alors les deux hypothèses pour le même but* `(ec \in yc)`:

```
theoremeCaseHyp.1.1: ec = e1c
theoremeCaseHyp.1.2: ¬(ec = e1c)
```

*Le premier cas est résolu automatiquement par* LP *en utilisant l'hypothèse* `theoremeImpliesHyp.1.2` *(Il oriente l'hypothèse en* `ec -> e1c` *pour réécrire le but en* `(e1c \in yc)` *simplifié à* `true`*).*
*Le deuxième cas correspond au contexte suivant:*

```
theoremeCaseHyp.1.2: ec = e1c -> false
theoremeImpliesHyp.1.1: ec \in xc -> true
theoremeImpliesHyp.1.2: e1c \in yc -> true
theoremeImpliesHyp.1.3: xc \< yc -> true
theoremeInductHyp.1: e \in xc /\ xc \< y => e \in y -> true
```

*L'état de la preuve est alors (obtenu avec la commande* `display conjecture`*):*

```
Level 4 conjecture theoreme.1: e \in x /\ x \< y => e \in y

  Level 4 subgoal 2 (induction step) for proof by induction on x:
    e \in insert(e1, xc) /\ insert(e1, xc) \< y => e \in y
    Current subgoal:
      (e = e1 \/ e \in xc) /\ e1 \in y /\ xc \< y => e \in y

    Level 4 subgoal for proof of =>: ec \in yc

      Level 4 subgoal for case 2 (out of 2): ec \in yc
```

*Comme nous ne savons pas exactement quelles hypothèses utiliser et quels axiomes de la base, on demande une complétion avec la commande* `complete`. *Ceci permettra à* LP *d'utiliser les règles de son système logique pour dériver de nouveaux faits. En particulier, il calcule les paires critiques non triviales pour essayer de simplifier le but à* `true`. *Il déduit la paire critique non triviale entre les deux hypothèses,* `theoremeImpliesHyp.1.3` *et* `theoremeInductHyp.1`

```
theoreme.2: e \in xc => e \in yc
```

*qu'il utilise pour calculer les paires critiques avec les hypothèses. En particulier, avec l'hypothèse* `theoremeImpliesHyp.1.1`, *il obtient la paire critique*

```
theoreme.3: ec \in yc
```

*Ce qui est précisément le but recherché.*

### 1.4.4 Preuve de schéma d'induction

LP permet de prouver des schémas d'induction, c'est à dire des assertions de la forme `Sort` $S$ `generated by` $g_1, g_2, \ldots, g_n$.

**Exemple 1.16** *Aux commandes suivantes :*

```
set name Ens_ind2
prove sort Ensemble generated by {},{_},\u
```

LP *génère les hypothèses :*

```
  Ens_ind2GenHyp.1: isGenerated({})
  Ens_ind2GenHyp.2: isGenerated({e})
  Ens_ind2GenHyp.3:
     isGenerated(s2) /\ isGenerated(s3) => isGenerated((s2 \u s3))
```

*et le but*

```
  isGenerated(s2)
```

*A la commande*

```
prove Sort Nat generated by 0,1,+
```

LP *génère les hypothèses et le but :*

```
Induction subgoal hypotheses:
  natGenHyp.1: isGenerated(0)
  natGenHyp.2: isGenerated(1)
  natGenHyp.3: isGenerated(n) /\ isGenerated(n1) => isGenerated(n + n1)
Induction subgoal:
  isGenerated(n)
```

*Pour une règle d'induction bien fondée de la forme*

```
  sort Nat generated freely by 0:-> Nat, f:Nat-> Nat, g:Nat-> Nat
```

LP *génère en plus du but* `isGenerated(n)`, *les sous-buts suivants :*

```
f(n) ¬= 0          f(n) = f(n1) <=> n = n1          f(n) ¬= g(n1)
g(n) ¬= 0          g(n) = g(n1) <=> n = n1
```

*Pour prouver la règle d'induction bien fondée* `well founded` $<$ , LP *crée un seul sous-but* `isgenerated(n)` *avec les hypothèses*

```
  ∀ n1 (n1 < n => isGenerated(n1)) => isGenerated(n)
```

### 1.4.5 Preuve de règles de déduction

La preuve d'une règle de déduction se ramène à une preuve d'une implication, car une règle de déduction est logiquement équivalente à une implication(cf. p.17). Ainsi, pour prouver une règle de déduction de la forme `when` $h_1, h_2, \ldots, h_n$ `yield` $c_1, c_2, \ldots, c_m$, LP la transforme en une implication $(h_1 \wedge h_2 \wedge \ldots \wedge h_n) \Rightarrow (c_1 \wedge c_2 \wedge \ldots \wedge c_m)$. Par conséquent, les règles d'extension `partitioned-by` peuvent être prouvées, puisque'elles sont transformées par LP en règle de déduction.

**Remarque 1.3** *La transformation par* LP *d'une règle de déduction en une implication est très utile. En effet, les règles de déduction sont un moyen puissant pour exprimer l'évolution du système à définir, ou bien son comportement suivant l'état de ses différentes composantes. Il est donc*

*intéressant de prouver les règles de déduction. Dans la version (2.4) de* LP*, on pouvait prouver une règle de déduction mais sans qu'elle soit transformée explicitement en une implication. Or comme sa sémantique était celle d'une implication,* LP *commençait la preuve directement comme une preuve d'une implication et par conséquent remplaçait les variables libres par des constantes, ce qui réduisait les méthodes à utiliser par la suite pour achever la preuve.*

**Exemple 1.17** *Soit la preuve suivante :*

```
set name expl
prove when R(i,j),R(j,n) yield R(i,n)
```

*Avec la version précédente,* LP *générait les faits suivants :*

```
New constants: ic, jc, nc
Hypotheses:
  explWhenHyp.1: R(ic,jc) = true
  explWhenHyp.2: R(jc,nc) = true
Subgoal:
  expl.1.1: R(ic,nc) = true
```

*Par conséquent, si la preuve nécessite ensuite une induction (par exemple, sur i), on était bloqués puisque nous n'avions plus que des constantes [Che92a].*

### Preuve d'une théorie

En LP, nous pouvons définir des théories associatives et/ou commutatives mais nous pouvons aussi prouver que l'ensemble d'axiomes spécifie une théorie commutative et/ou associative. Prouver qu'un opérateur est associatif et/ou commutatif revient à montrer les règles usuelles de l'associativité et/ou de la commutativité.

## 1.5 Consistance

Une approche standard en logique pour prouver la consistance d'une théorie revient à la définir à l'aide d'une autre théorie dont la consistance est supposée ou a été vérifiée. En LP, cela revient à prouver que l'interprétation satisfait les axiomes du trait. Si nous ne pouvons pas montrer la consistance, nous pouvons tout de même montrer l'inconsistance d'une spécification avec un calcul de paires critiques. En effet, pour des traits équationnels, si le calcul de paires critiques s'arrête sans générer la paire `true = false` alors le trait est consistant. Cette stratégie de preuve ne réussit pas toujours du fait que plusieurs théories équationnelles ne peuvent être complétées, ou bien ne peuvent l'être en un temps raisonnable. Néanmoins, cette stratégie "a fait ses preuves" pour les traits équationnels et non équationnels. Même si cela ne certifie pas que la spécification est consistante, un début de complétion donne au moins une idée de la cohérence de l'ensemble des axiomes, car on peut espérer que si une inconsistance existe, elle sera découverte assez tôt.

## 1.6 Remarques générales

LP propose un ensemble assez large de méthodes de preuves, mais il ne fournit aucune indication sur quelle méthode utiliser pour telle conjecture. Or, il est important de bien choisir l'ordre dans lequel utiliser les différentes méthodes, en particulier lorsque LP remplace les différentes

variables libres par des constantes, on ne peut plus faire d'induction. D'où l'importance de concevoir une preuve avant de la corriger à l'aide du démonstrateur. En effet, l'échec d'une preuve (de sa mécanisation) peut être dû à la stratégie utilisée, c'est à dire les différentes méthodes LP qu'elle requière et l'ordre dans lequel on les applique. Vue la multitude des méthodes proposées, on comprend aisément le choix des auteurs de concevoir un système semi-automatique capable de prendre en charge les étapes pénibles de la preuve tout en laissant à l'utilisateur une liberté suffisante dans la stratégie de preuve.

Les concepts suivants ne sont pas fournis par LP, mais peuvent être utiles pour certaines spécifications, en particulier pour spécifier des systèmes complexes.

- Une logique avec des fonctions partielles : pour la spécification d'un logiciel ou d'une application, on peut être en présence de fonctions partiellement définies. En LP, les fonctions doivent être totales, il faut alors utiliser un mécanisme de sous spécification, où les valeurs critiques d'une fonction ne sont pas explicités. Par exemple, pour la spécification d'une pile, on ne spécifie pas la valeur de `top(nil)`.

- Ordre supérieur : on ne peut pas raisonner sur des éléments de type fonction, ce qui restreint la puissance d'expression du langage.

- Une théorie de point fixe et les objets infinis : Il est parfois utile de définir des éléments par récursion, c'est à dire des points fixes de fonctions monotones ou continues. Ceci fait appel à des objets infinis et à une théorie de point fixe.

- Le polymorphisme : Pour la spécification des listes, le type des éléments n'est pas pertinent pour les fonctions de manipulation. Par exemple, la fonction qui retourne le premier élément de la liste ne prend pas en compte le type des éléments. Ceci permet d'avoir des spécifications plus concises et ré-utilisables. En LP, ce traitement n'est pas possible mais on peut utiliser un même nom pour différentes fonctions ce qui permet de simuler ce polymorphisme, mais les noms des sortes doivent être distincts. Par conséquent les axiomes, et éventuellement les preuves, doivent être dupliquées. Nous verrons dans les chapitres suivants que l'absence de polymorphisme n'est pas un obstacle pour la ré-utilisation des définitions et des preuves.

- Héritage et sous types : dans certains contextes, il est parfois utile de définir une sorte comme étant incluse dans une autre, ce qui permet à la sorte incluse d'hériter des propriétés de la sorte parente. Ceci n'est pas possible en LP, mais on peut contourner le problème par des fonctions de conversions.

Néanmoins, l'objectif de notre travail étant la vérification mécanique des programmes UNITY, avec un aspect de détection d'erreurs, nous attachons plus d'importance aux techniques fournies par LP pour la construction des preuves, et au raisonnement interactif qui permet le retour arrière et la mise au point des preuves. De plus, ces aspects compliquent le langage ainsi que le calcul et les rend plus difficiles à apprendre et à utiliser.

# Chapitre 2

# Formalisation et Mécanisation d'Unity

## 2.1 Introduction

Vérifier qu'un programme satisfait les propriétés requises de correction nécessite d'une part, une représentation formelle du programme et des propriétés et d'autre part, un ensemble de méthodes de vérification adéquat. Le but du travail décrit dans ce document consiste à développer un système de preuve adapté à la vérification mécanisée des programmes parallèles et plus particulièrement à celle des programmes décrits enUNITY.

UNITY est une théorie pour spécifier et vérifier les propriétés des programmes parallèles. Cette théorie comprend un langage de programmation pour écrire (spécifier) des programmes, un fragment de logique temporelle pour exprimer les propriétés de *sûreté* et de *vivacité* [Lam77] que le programme doit satisfaire et enfin un système de preuve pour vérifier ces propriétés.

Un programme UNITY comprend une déclaration de variables, des conditions initiales portant sur ces variables et un ensemble fini d'instructions (affectations). La logique d'UNITY comprend, outre la logique prédicative du premier ordre, des opérateurs "temporels" ou *modalités*, à savoir l'opérateur, *unless*, pour spécifier la sûreté, et deux opérateurs, *ensures* et *leads_to* pour exprimer le progrès (sous ensemble des propriétés de vivacité). Ces opérateurs ou *relations* ont pour argument des prédicats (non modaux) et par conséquent sont des fonctions d'ordre supérieur.

La théorie d'UNITY développée dans ce travail est définie de telle sorte que les propriétés à vérifier le soient pour tout état défini par le domaine des données du programme. Les définitions formelles des opérateurs temporels d'UNITY sont données en terme de relations entre les transitions d'états du programme en utilisant des pré- et post-conditions. Ces définitions sont basées sur la représentation du programme comme un système de transitions (d'états à états). Dans ce modèle, il est possible de définir des propriétés modales en termes de pré- et post-condition [Hoa69], entre toute transition possible du programme représenté comme un ensemble de transitions. En utilisant cette définition, tout état vérifiant la pré-condition (un prédicat) d'une propriété doit mener à un état vérifiant la post-condition de la propriété. Par conséquent, les prédicats sont quantifiés par rapport à tous les états possibles définis par le domaine des données du programme.

Bien que les définitions formelles des opérateurs soient données en terme de pré- et post-condition, leur sémantique, telle qu'elle a été définie dans [Cha88], utilise des relations entre les états des chemins d'exécution du programme. Ceci peut créer une ambiguïté lors de la formalisation car les pré- et post-conditions doivent être vérifiées par tous les états possibles alors qu'avec un raisonnement sur les traces, seuls les états *accessibles*[11] sont pris en compte.

Dans ce qui qui suit, les définitions formelles et la sémantique de *unless* et *ensures* sont données

11. Un état accessible est un état atteint par une séquence d'actions à partir d'un état initial.

en termes de pré et post-conditions comme étant des relations entre les états de transition du programme. Chaque opérateur exprime une relation entre des ensembles d'états définis par un prédicat.

## 2.2 La logique d'UNITY

La logique d'UNITY comprend la logique du premier ordre avec les règles usuelles du calcul des prédicats, enrichie par des modalités temporelles pour exprimer les propriétés qu'un programme parallèle doit vérifier.

• Les propriétés de **sûreté** spécifient "ce qu'un programme ne doit pas faire". Ce sont des propriétés qui doivent être satisfaites par tous les états d'un programme. Elles sont définies à l'aide des opérateurs *unless*, *stable* et *invariant*.

Pour un programme $P$:

$$p \ unless \ q \stackrel{def}{=} \langle \forall s : s \in P :: \{p \wedge \neg q\} s \{p \vee q\} \rangle$$

La sémantique opérationnelle de cette définition est la suivante: si $p$ est valide et $q$ ne l'est pas à une étape de l'exécution, alors après l'exécution d'une instruction, soit $p$ reste valide, soit $q$ devient valide. Si on utilise les pré- et post-conditions (transitions d'un état à l'autre), alors $p \ unless \ q$ est vrai pour un programme *pgm* si "dès lors que *pgm* atteint un état dans lequel $p$ est vrai, toutes les transitions doivent préserver $p$ ou bien rendre $q$ valide".

À partir de *unless*, on définit deux cas particuliers, la *stabilité* et l'*invariance*.

$$\begin{aligned} stable \ p &\stackrel{def}{=} p \ unless \ false \\ &\stackrel{def}{=} \langle \forall s : s \in P :: \{p\} s \{p\} \rangle \end{aligned}$$

Un prédicat $p$ est *stable* si lorsqu'il est valide dans un certain état, toutes les transitions le préserve.

$$invariant \ p \stackrel{def}{=} (condition \ \ initiale \Rightarrow p) \wedge stable \ p$$

Un invariant est un prédicat *stable* valide à l'état initial.

• Les propriétés de **progrès** spécifient "ce qu'un programme doit faire". UNITY comprend deux opérateurs pour exprimer les propriétés de progrès, *ensures* et *leads_to*:

$$p \ ensures \ q \stackrel{def}{=} (p \ unless \ q) \wedge \langle \exists s : s \in P :: \{p \wedge \neg q\} s \{q\} \rangle$$

si $p$ est valide et $q$ ne l'est pas dans un certain état alors toutes les transitions doivent préserver $p$ jusqu'a ce que $q$ devienne vrai. De plus, il *doit* exister une instruction dans le programme $P$ qui établit la validité de $q$. Le prédicat *leads_to* est alors défini comme la fermeture transitive et disjonctive de la relation **ensures**, ou bien la plus petite relation qui vérifie les règles d'inférence suivantes:

$$\frac{p \ ensures \ q}{p \ leads\_to \ q}$$

$$\frac{p \ \ leads\_to \ \ r, r \ \ leads\_to \ \ q}{p \ \ leads\_to \ \ q}$$

$$Pour \ \ tout \ \ W, \ \ \frac{\forall m \in W : p(m) \ \ leads\_to \ \ q}{\exists m \in W : p(m) \ \ leads\_to \ \ q}$$

La modélisation d'UNITY consiste donc d'une part à formaliser la notion de programme (initialisation, affectation, etc) et d'autre part, à formaliser les opérateurs temporels de la logique et le système de preuve associé. Nous considérons alors la logique UNITY comme étant composée de trois niveaux: le calcul des prédicats, le calcul associé aux programmes et enfin les opérateurs temporels. Le succès de UNITY provient du fait qu'il offre un environnement unique pour écrire un programme, pour spécifier les propriétés pertinentes et pour vérifier la correction du programme relativement à ces propriétés. Dans la même philosophie, notre objectif consiste à utiliser un même outil pour écrire des spécifications formelles et pour pour valider les preuves de correction à l'aide d'un certain nombre de techniques de vérification.

Nous avons choisi comme outil formel pour la preuve des programmes UNITY, le calcul de *la plus faible pré-condition* dû à Dijkstra [Dij89]. En effet, les preuves en UNITY sont basées sur des assertions du type $\{P\}s\{Q\}$, où $s$ représente une instruction d'affectation d'UNITY, $P$ une pré-condition qui doit être vraie avant l'exécution de $s$ et $Q$ une post-condition qui résulte de l'exécution de $s$. Cette notation est équivalente à $P \Rightarrow wlp(s, Q)$, où $wlp$ est la plus faible pré-condition de $Q$.

Cette modélisation (et sa mécanisation) est indépendante du modèle d'UNITY, et pourrait être appliquée (et bien sur, étendue) à un autre langage de programmation, avec une extension qui prendrait en compte d'autres constructions. Pour cela, cette extension compléterait la définition sémantique des instructions du langage considéré, sans modifier la formalisation et la mécanisation du calcul des prédicats. Avec une sémantique opérationnelle utilisant la notion explicite d'états et de traces d'exécutions, la formalisation doit prendre en compte des références explicites aux états, gérer la valeur des variables d'états avant et après les transitions, définir des opérateurs de transition d'un état à l'autre, et surtout définir des fonctions de transitions vérifiant les conditions d'*équité* pour le choix d'instruction à exécuter.

Nous allons décrire dans ce qui suit la formalisation des trois composantes de la logique d'UNITY à l'aide du démonstrateur de théorème LP. En particulier, nous allons étudier comment formaliser le calcul des prédicats, et le calcul des *wp* afin de raisonner sur les programmes et enfin définir les opérateurs temporels d'UNITY à l'aide d'un démonstrateur du premier ordre tel que LP.

## 2.3 Le langage des structures booléennes

### 2.3.1 Préliminaires

Comme nous l'avons décrit précédemment, le but de notre formalisation est de prouver des formules du type $unless(P, Q, prog)$. L'opérateur *unless* (et *ensures*) a donc comme argument deux prédicats et une variable *prog* qui représente le programme considéré. Par conséquent *unless* est une fonction d'ordre supérieur, manipulant des objets dont la sémantique est celle des prédicats.

Définir ces opérateurs dans la logique de premier ordre sous jacente à LP nécessite la formalisation de la notion de "prédicat", afin que les variables $P$ et $Q$ soient des variables LP d'un

certain type mais dont la sémantique est celle des prédicats. Il faut noter que cette formalisation doit être indépendante des programmes à prouver, c'est à dire des types de données manipulés.

Afin de présenter cette formalisation, nous commençons par rappeler certaines notions de la sémantique des programmes basées sur les travaux de Dijkstra [Dij89].

**Espace d'État** Pour un programme qui opère sur $n$ variables, l'espace d'états est visualisé par un espace de dimension $n$ dans lequel les $n$ variables du programme sont les $n$ coordonnées cartésiennes. La notion d'*espace d'état* est une métaphore pour représenter la correspondance point à point entre les points de l'espace et toutes les combinaisons possibles de valeurs des $n$ variables. Puisque à chaque combinaison des valeurs de ces variables correspond un état de la *mémoire*, il y a une correspondance directe entre un état de la mémoire et un point de l'espace d'état. Par conséquent, l'expression "pour toute combinaison des valeurs des variables du programme" peut se réduire à "en tout point de l'espace d'états" ou "partout dans l'espace d'états", ou "partout".

Pour ne pas confiner la notion d'"expression" à "une combinaison possible de valeurs de $n$ variables d'un certain espace de coordonnées", les variables de la théorie ne sont pas considérées de type "expression" mais de type "structure".

La notion de structure est donc une abstraction de la notion "expression contenant des variables du programme" dans le sens où les notions d'espace d'état et de dimension ont été éliminées. Néanmoins, les expressions comprenant des variables de programmes ont un type, ce sont des expressions *entières* (arithmétiques) ou des expressions *booléennes*. Par conséquent, les structures peuvent être des structures booléennes ou arithmétiques. Pour les structures arithmétiques, *1*, *2*, ... sont des constantes, appelés *scalaires*. Par analogie, on considère deux structures booléennes particulières, *vrai* et *faux* appelées *scalaires booléens*.

**Égalité** Le problème de l'égalité provient du fait que $A = B$ n'est pas en général considéré comme une expression booléenne mais plutôt comme le fait que $A$ et $B$ sont égaux "partout" (ou bien représentant la même quantité). Par exemple, si $A$ et $B$ sont deux matrices, $A = B$ est généralement vu comme le fait que les deux matrices sont "égales" point à point et non comme une matrice de booléens (un objet construit avec une comparaison point à point), alors que pour deux expressions réelles $A$ et $B$, $A = B$ représente bien une expression booléenne, puisque la comparaison point à point n'est plus possible. Par conséquent, une expression où deux structures booléennes connectées par un opérateur relationnel, tel que $A{=}B$ ou $A < B$, n'est pas interprétée comme étant un fait établi mais comme une expression booléenne pouvant être aussi générale que les opérandes.

Une fonction appelée "partout" (*everywhere*) est introduite, notée [], qui appliquée à une structure booléenne retourne un scalaire booléen *vrai* ou *faux*. Appliquée à un scalaire booléen, elle opère comme la fonction identité, $[vrai] = vrai, [faux] = faux$ car *vrai* ou *faux* sont les deux seuls scalaires booléens. Le prix d'une telle formalisation est l'ajout de [] chaque fois qu'on veut exprimer une égalité complète entre deux opérandes qui peuvent être des structures. La règle de Leibnitz $x = y \Rightarrow f(x) = f(y)$ est exprimée par la formule $[x = y] \Rightarrow [f(x) = f(y)]$. Par conséquent, pour représenter l'égalité entre deux structures booléennes, on utilise $\equiv$ qui est associatif et non $=$.

**Format de preuve** La sémantique d'un énoncé de théorème est une expression booléenne et celle de sa preuve est un calcul évaluant cette expression booléenne à "vrai". Dans ce contexte, le but de notre formalisation est donc de manipuler des structures booléennes et arithmétiques pour

prouver une " expression" de la forme $[P \Rightarrow wp(s,Q)] \equiv vrai$. Partant de $[P \Rightarrow wp(s,Q)]$, nous devons utiliser un certain nombre de règles de calcul pour obtenir la valeur *vrai*. Pour ce faire, nous devons considérer deux aspects : la formalisation de l'opérateur "partout" et la formalisation de l'égalité.

### 2.3.2 Le langage objet

Les pré- et post-conditions utilisées dans la définition des opérateurs temporels sont des prédicats d'*état*, des fonctions booléennes dont le domaine est l'espace d'état du programme considéré et le co-domaine des valeurs booléennes. Un état du programme est défini par la valeur courante des variables du programme. Plus simplement, les prédicats sont des expressions contenant des variables du programme qui peuvent être évalués à *vrai* ou $faux$ suivant la valeur courante des variables.

Un *prédicat* du langage *objet* est défini comme étant une *structure booléenne*, c'est à dire deux structures connectées par un opérateur. Une *structure* est une expression (ou abstraction) qui peut contenir des variables (de programme).
**Exemple:**

```
x: Nat
y: Nat
b: bool
*corps du prog*
```

$x < y$ est une structure booléenne, $x + y$ est une structure arithmétique et $x < y \Rightarrow x \leq y$ est une structure booléenne. De même, $vrai, faux$, $b$ sont des structures booléennes, $x$ et $y$ sont des structures arithmétiques. Par contre,

```
[b]
[x + y = 0]
[x < y ⇒ x ≤ y]
```

sont des expressions de type *scalaire booléen* "égales" à *vrai* ou $faux$. L'opérateur [] joue ainsi le rôle d'interprétation du *langage objet* vers les booléens de LP, donc vers le méta-langage de LP. Par exemple, l'expression

```
[x ≤ y ≡ (x < y) ∨ (x=y)]
```

est traduite en

```
(x ≤ y ≡ (x < y) ∨ (x=y)) = T
```

L'évaluation de cette égalité dans la théorie de LP est soit `true` soit `false`.

Nous définissons une sorte LP, `Bexp`, représentant les structures booléennes, où *les expressions de prédicat*. Sémantiquement, `Bexp` est un ensemble de prédicats. Puis, nous avons besoin de définir les opérateurs usuels tels que l'implication, la disjonction et la conjonction. Nous définissons `\=>`, `\or`, `^`, dont la signature est : `Bexp,Bexp -> Bexp`, `T` et `F`, différents des opérateurs prédéfinis de LP, `=>` , `\/` , `/\`, `true`, `false`, dont la signature est : `Bool,Bool -> Bool`. De même, nous définissons l'opérateur de négation, `nnot`, différent de l'opérateur de négation `~` de LP. Les premiers sont des opérateurs du langage objet et les seconds des opérateurs de la méta-théorie. Nous avons ainsi promu les opérateurs logiques opérant avec des variables booléennes à des opérateurs logiques opérant avec des "prédicats".

Qu'en est-il de l'égalité? L'opérateur = de LP est automatiquement déclaré pour chaque sorte $S$, de signature `S,S -> Bool`. Cet opérateur est commutatif, et associatif si la sorte est celle des booléens, considéré alors par LP comme un synonyme de `<=>`.

La solution consiste à considérer le symbole d'égalité de LP comme l'opérateur "partout" et un opérateur `\<=>`, dont la signature est `Bexp,Bexp -> Bexp`, pour l'égalité entre deux structures booléennes. En d'autres termes, nous traduisons l'expression $[A \equiv B]$ du type *true* ou *false* en `(A \<=> B) = T`, du type `true` ou `false` dans la théorie de LP.

### Le calcul

Nous allons, à présent, définir les propriétés des opérateurs logiques du langage objet, qui en LP, sont des fonctions totales. Il est certain que nous pouvons introduire directement les axiomes que nous supposons utiles pour notre formalisation. Néanmoins, deux points sont essentiels , d'une part la consistance (la cohérence) du système obtenu et d'autre part, l'utilisation effective de ce système.

- A la différence de NQTHM [Boy88] et de HOL [Gor93], LP ne contient aucune théorie prédéfinie dans laquelle définir une nouvelle théorie. Par conséquent, nous n'avons aucun moyen explicite de vérifier la consistance des théories que nous définissons. Néanmoins, nous allons utiliser une démarche en trois étapes: *Enrichir*, *vérifier* et *corriger*. Cette démarche consiste à introduire un minimum d'axiomes, puis à enrichir la spécification par des théorèmes. Les théorèmes sont de même nature que les axiomes sauf qu'il faut les démontrer pour qu'ils puissent être ajoutés au système.

- On ne peut dissocier la spécification de son utilisation. En effet, notre but ne se résume pas à formaliser les "prédicats" mais il faut pouvoir les utiliser pour les preuves que nous nous proposons d'effectuer dans le cadre des programmes parallèles. En d'autres termes, l'utilisation de cette formalisation est importante pour la mécanisation du système et pour le degré d'automatisation. Pour cela, nous allons expliciter dans ce qui suit la démarche suivie en étudiant quelques exemples afin de justifier les choix de formalisation.

Pour construire notre système, nous allons suivre la démarche explicitée dans [Dij89], tout en utilisant la démarche *enrichir-vérifier-corriger*.

Soit le système suivant où les axiomes (4,5,6) définissent la réflexivité de `\<=>`, l'idempotence et la distributivité de `\or` par rapport à `\<=>` (ces deux opérateurs sont préalablement définis comme des opérateurs associatifs et commutatifs):

```
bool_str.3: p \<=> T -> p
bool_str.4: p \<=> p -> T
bool_str.5: (p \or p) \<=> p -> T
bool_str.6: (p \or (q \<=> c)) \<=> (p \or c) \<=> (p \or q) -> T
```

Le théorème suivant:

```
(c \or p \or q) \<=> (c \or p \or p \or q) = T
```

peut être prouvé en utilisant l'idempotence. Mais telle qu'elle est formalisée par la règle `bool_str.5`, elle ne peut pas être utilisée directement. Ce théorème est alors prouvé à l'aide d'un calcul de paires critiques (cf. paragraphe 1.4) entre les deux règles `bool_str.4` et `bool_str.5`[12].

12. si `\<=>` n'était pas ac, ce calcul ne donnerait pas de paires critiques.

Ce calcul produit la règle `(p \or p) -> p`, qui permet de réécrire le théorème en `(c \or p \or q) \<=> (c \or p \or q) = T`, puis en `T = T` avec la règle `bool_str.4`.

Le théorème est alors introduit dans la base sous la forme :

```
bool_str.7: (c \or p \or q) \<=> (c \or p \or p \or q) -> T
```

Le terme `p \or p` n'est pas réécrit en `p` car la paire critique correspondante a été calculée dans le contexte d'une preuve et par conséquent son existence est liée à ce contexte.

Nous allons à présent montrer que `T` est l'élément absorbant de `\or`, en utilisant `\<=>` :

```
((p \or T) \<=> T) = T
```

Ce théorème est réécrit `p \or T = T` à l'aide de la règle `bool_str.3`, puis un calcul de paires critiques donne la règle `b \or T -> T` (règles `bool_str.3` et `bool_str.6`) avec laquelle le théorème est normalisé à `T=T`[13]. Le système obtenu est alors :

```
bool_str.3: p \<=> T -> p
bool_str.4: p \<=> p -> T
bool_str.5:(p \or p) \<=> p -> T
bool_str.6:(p \or (q \<=> c)) \<=> (p \or c) \<=> (p \or q) -> T
bool_str.7:(c \or p \or q) \<=> (c \or p \or p \or q) -> T
bool_str.8: p \or T -> T
```

où la propriété d'idempotence a été introduite sous sa forme normale, `bool_str.8`. Cette forme est plus utile comme règle de simplification que la forme originale.

Pour vérifier la cohérence de notre système, nous pouvons calculer les paires critiques de ce système. Ce calcul s'arrête et produit le système suivant

```
bool_str.3: p \<=> T -> p
bool_str.4: p \<=> p -> T
bool_str.8: p \or T -> T
bool_str.9: b \or b -> b
bool_str.41:p \or (q \<=> c) -> (p \or c) \<=> (p \or q)
```

Nous pouvons remarquer que la règle `bool_str.5` a été éliminée, normalisée avec les règles `bool_str.9` ( obtenue par calcul de paires critiques) et `bool_str.4`.

Ce processus correspond au passage du langage objet vers le méta-langage de LP, pour que l'égalité dans le langage objet soit la réécriture dans la méta-théorie :

$$[p \vee p \overset{objet}{\equiv} p] \overset{Theorie}{\equiv} true \quad \rightsquigarrow \quad \texttt{(p \or p)} \overset{objet}{\texttt{\<=>}} \texttt{p} \overset{LP}{=} \texttt{T} \quad \rightsquigarrow \quad \texttt{(p \or p} \overset{LP}{=} \texttt{p)}$$

• **La conjonction** comme opération dérivée de la disjonction et de l'équivalence : soit l'axiome suivant, appelé *règle d'Or*, qui permet de définir la conjonction par rapport à la disjonction et à l'équivalence.[14] Cet axiome contient 3 occurrences du connecteur $\equiv$ :

$$[X \wedge Y \equiv X \vee Y \equiv X \equiv Y] \tag{2.1}$$

13. Une preuve plus simple mais moins automatique aurait été d'instancier c par q dans la règle 6.
14. Par symétrie, il permet aussi de définir la disjonction par rapport à la conjonction et l'équivalence.

Comment s'"assurer" que la conjonction ainsi définie est bien la conjonction que nous connaissons? Pour cela, on peut montrer que *true* est l'élément neutre de la conjonction :

$$[X \wedge true \equiv X] \tag{2.2}$$

**Preuve :**

$X \wedge true$
= { Règle d'Or avec $Y = true$}
$X \vee true \equiv X \equiv true$
= { *true* est l'élément absorbant de $\vee$}
$X \equiv true \equiv true$
= {réflexivité de $\equiv$ }
$X \equiv true$
= { *true* est l'élément neutre de $\equiv$}
$X$

Plusieurs preuves en LP vont découler de la façon de prendre en compte la *règle d'Or*.
Le base de faits est :

$R_1$: `p \<=> T -> p`
$R_2$: `p \<=> p -> T`
$R_3$: `p \or T -> T`
$R_4$: `b \or b -> b`
$R_5$: `p \or (q \<=> c) -> (p \or c) \<=> (p \or q)`

Premier choix : le premier connecteur $\equiv$ est considéré comme une égalité LP, transformé en réécriture :

$R_6$: `p ^ q -> p \or q \<=> p \<=> q`

Nous avons traduit une occurrence du connecteur $\equiv$ en une égalité, les autres comme équivalences. Cette règle nous permet de "définir" l'opérateur `^` en fonction de `\<=>` et `\or`. La preuve en LP est alors la suivante, où nous passons d'une ligne à l'autre en réécrivant l'un des membres de l'équation.

**LpPreuve 2.1**

```
 p ^ T = p
{R6}
 p \or T \<=> p \<=> T = p
{R1}
 p \or T \<=> p = p
{R3}
  T \<=> p = p
{associativité de \<=>}
 p \<=> T = p
{R1}
 p = p
{LP}
 true
```

Le théorème prouvé est donc $R_7$ : `p ^ T -> p`

Deuxième choix : la deuxième occurrence du connecteur $\equiv$ est transformé en une égalité, ce qui nous permet de donner une autre sémantique à la *règle d'Or*:

$R_{6'}$: `p ^ q \<=> p \or q -> p \<=> q`

Cette règle exprime le fait que si la disjonction de deux prédicats est équivalente à leur conjonction, alors ces deux prédicats sont équivalents. Cette règle nous permet de prouver le théorème 2.2, mais la preuve se déroule de façon différente : Partant de la *règle d'Or*, nous instancions l'une des variables à T. La règle obtenue est alors normalisée et nous obtenons le théorème voulu :

**LpPreuve 2.2**

```
 p ^ T \<=> p \or T -> p \<=> T
{R3}
 p ^ T \<=> T -> p \<=> T
{R1}
 p ^ T \<=> T -> p
{R1}
 p ^ T -> p
```

Troisième choix : le troisième connecteur est transformé en un égalité LP.

$R_{6\prime}$ : `p ^ q \<=> p \or q \<=> p -> q`

Ce choix nous permet de montrer un théorème "différent" de $R_7$. En utilisant les mêmes règles que précédemment, le théorème que l'on montre est

$R_{new7}$ : `p ^ T \<=> p -> T`

Ce théorème diffère du $R_7$ par la traduction de l'équivalence. Nous pouvons néanmoins prouver $R_7$ avec un calcul des paires critiques entre la règle $R_{new7}$ et la *règle d'Or*, $R_{6\prime}$. Le choix entre $R_7$ et $R_{new7}$ dépend de leur utilisation. Pour notre formalisation, la première forme, $R_7$, est plus intéressante comme règle de simplification.

Quatrième choix : la formule entière est supposée égale à *true*, en traduisant la sémantique de [] directement par l'égalité de LP. En effet, dans certains cas, nous n'avons pas un *choix* pour la traduction des axiomes. Il faut alors utiliser d'autres techniques afin de prouver les théorèmes qui nous intéresse. Pour illustrer ce cas de figure, nous introduisons la règle d'Or sous sa forme originale :

```
 (p ^ q) \<=> (p \or q) \<=> p \<=> q = T
```

Cet axiome sera orienté en une règle de réécriture

$R_6$ : `(p \or q) \<=> (p ^ q) \<=> p \<=> q -> T`

Il nous permet de montrer le théorème $R_{new7}$ et l'idempotence de ^ :

```
prove (p ^ p) \<=> p = T
 ins q by p in bool_str.69   %R6
 [] conjecture
prove (p ^ T) \<=> p =T
 cri-p bool_str* with bool_str*
 [] conjecture
qed
```

Conclusion : la choix de la traduction de la *règle d'Or* en un axiome de la théorie de LP dépend de son utilisation dans le calcul. La *règle d'Or* permet de définir le nouvel opérateur ^ en fonction des opérateurs existants \or et \<=>. Ainsi, elle nous permettrait de réécrire un terme de la forme `p ^ q` en `p \or q \<=> p \<=> q`, afin de ramener toute preuve mettant en jeu l'opérateur ^ à une preuve sur les opérateurs \<=> et \or. Suivant la forme syntaxique choisie, nous pouvons

montrer les mêmes théorèmes, mais les structures de preuves peuvent être plus ou moins simples.

En effet, soit le système de réécriture $\{R_1,R_2,R_3,R_4,R_5,R_6\}$. Pour prouver une formule qui contient l'opérateur ^, il faut *éliminer* cet opérateur de la formule considérée, car le système courant ne contient aucun autre axiome mettant en jeu cet opérateur. Or nous ne pouvons pas utiliser directement la *règle d'Or* ($R_6$), puisque l'équivalence du langage objet n'est pas une égalité de la théorie LP. Par contre, nous pouvons montrer la première forme syntaxique de la *règle d'Or*:

$R_8$: `p ^ q = (p \or q) \<=> p \<=> q`

à l'aide de $R_6$.

**LpPreuve 2.3**

```
(p ^ q)
{Introduction avec R1}
= ((p ^ q) \<=> T)
{R6}
= ((p ^ q) \<=> (p ^ q) \<=> (p \or q) \<=> p \<=> q)
{élimination avec R2 }
= (T \<=> (p \or q) \<=> p \<=> q)
{élimination de T avec R3}
= ((p \or q) \<=> p \<=> q)
```

Les deux premières étapes de la preuve consistent à utiliser les règles $R_1$ et $R_6$ comme étant orientées dans le sens inverse. Bien que ce processus soit possible en LP, cette technique est néanmoins dangereuse et plus complexe à manipuler (elle nécessite la passivité et l'immunité des règles). Nous pouvons contourner cette technique en prouvant de façon explicite des lemmes appropriés :

**LpPreuve 2.4**

```
prove p ^ q = (p \or q) \<=> p \<=> q
 make passive bool*          %pour eviter une réécriture trop rapide
   set name lemme
   prove
    (p \or q) \<=> p \<=> q
     =
    (p \or q) \<=> (p ^ q) \<=> p \<=> q \<=> (p \or q) \<=> p \<=> q
   ..
  rewrite conj with bool_str.69   %R6
  rewrite conj with bool_str.3    %R1
  [] conjecture
 normalize lemme with bool_str.4, bool_str.3   %R2 et R3
 [] conjecture
qed
```

Le lemme prouvé correspond à une phase que nous appelons l'*introduction*, substitution de `T` par le partie gauche de la *règle d'Or* $R_6$. Une fois ce lemme prouvé, il suffit de le réécrire en utilisant les deux règles $R_2$ et $R_3$.
En utilisant la symétrie induite par la *règle d'Or*, cette méthode permet de prouver des résultats similaires pour l'opérateur `\or`.

**Exemple**: Supposons qu'on ait à prouver la formule

`(q ^ c \<=> q \<=> c) = (q \or c)`

Il suffit pour cela d'introduire `T` avec $R_1$, de réécrire le membre gauche avec la *règle d'Or*, puis d'éliminer avec la réflexivité de l'équivalence ($R_2$):

**LpPreuve 2.5**

```
(q ^ c \<=> q \<=> c)
{R1}
(q ^ c \<=> q \<=> c \<=> T)
{R6}
(q ^ c \<=> q \<=> c \<=> q ^ c  \<=> q \or c \<=> q \<=> c)
{R2, p ← q ^ c}
(T \<=> q \<=> c \<=> q \or c \<=> q  \<=> c)
{R2, p ← q \<=> c }
(T \<=> T \<=> q \or c)
{R2, p ← T}
(T \<=> q \or c)
{R_1}}
(q \or c)
```

Si nous disposons de la règle $R_8$, la preuve se résume à éliminer `q ^ c`, puis à absorber en utilisant trois fois la règle $R_2$. Ce théorème est prouvé et immunisé pour qu'il ne soit pas simplifié en une identité :

$R_9$: `(q \or c) = q ^ c \<=> q \<=> c`

Cette méthode, permettant d'éliminer dans une preuve un nouvel opérateur en introduisant les opérateurs précédemment définis, sera réutilisée, en particulier pour l'implication. Cette réutilisation consiste à appliquer une même structure de preuve mécanisée.

• **L'implication :**

$$[X \Rightarrow Y \equiv X \vee Y \equiv Y] \tag{2.3}$$

Cet axiome est traduit directement en un axiome de la méta-théorie de LP, où l'implication est définie en fonction des opérateurs `\<=>` et `\or` :

```
(p \=> q) \<=> (p \or q) \<=> q = T
```

Cette règle permet de montrer de façon triviale la réflexivité de `\=>`, à l'aide de l'idempotence de `\or`.

Soit à présent le système :

```
bool_str.3:      p \<=> T -> p
bool_str.4:      p \<=> p -> T
bool_str.8:      p \or T    -> T
bool_str.9:      b \or b    -> b
bool_str.41:     p \or (q \<=> c) -> (p \or c) \<=> (p \or q)
bool_str.72:     p ^ q   -> (p \or q) \<=> p \<=> q
bool_str.73(I): (p ^ q) \<=> p \<=> q -> p \or q
bool_str.74(I): (p ^ (p \or q)) \<=> p  -> T
bool_str.75(I): (p \or (p ^ q)) \<=> p  -> T
bool_str.76(I): (p \or (q ^ c)) \<=> ((p \or q) ^ (p \or c))  -> T
bool_str.77(I): (p ^ (q \or c)) \<=> ((p ^ q) \or (p ^ c)) -> T
bool_str.78(I): (p ^ (q \<=> c)) \<=> (p ^ c) \<=> (p ^ q) \<=> p -> T
bool_str.81:    (p \=> q) \<=> (p \or q) \<=> q -> T
bool_str.82:    p \=> p -> T
```

Les règles `bool_str.74` et `bool_str.75` correspondent aux théorèmes d'*Absorption*, `bool_str.76` et `bool_str.77` représentent la distributivité de `\or` par rapport à `^` et réciproquement. La règle `bool_str.78` permet d'éliminer `\<=>` d'un sous terme.

Par analogie à l'introduction de l'opérateur `^`, nous pouvons réutiliser la même structure de preuve que celle de la preuve 2.4, pour montrer que `p \=> q` se réécrit en `(p \or q) \<=> q`.

**LpPreuve 2.6**

```
prove p \=> q = (p \or q) \<=> q  %bool_str.83
 make passive bool*
 set name lemme
 prove
  (p \or q) \<=> q
   = ( p \=> q) \<=> (p \or q ) \<=> q \<=> (p\or q) \<=> q
 ..
  rewrite conj with bool_str.81
  rewrite conj with bool_str.3
  [] conjecture
 normalize lemme with bool_str.4,bool_str.3
 [] conjecture
qed
```

La stratégie est similaire à celle utilisée pour la preuve 2.4 : le lemme permet la substitution de `T` par la partie gauche de la règle qui définit l'implication (`bool_str.81`), puis réécriture du lemme en utilisant les deux règles de simplification.

Le système précédent auquel a été ajoutée la règle

```
bool_str.83: p \=> q -> (p \or q) \<=> q
```

nous permet de prouver deux théorèmes de façon "automatique":

```
prove (p \=> q) \<=> (p ^ q) \<=> p =T
 [] conjecture
prove ((p \=> q) ^ (q \=> c)) \=> (p \=> c) = T
 [] conjecture
```

La preuve utilise la définition de l'implication, celle de la conjonction (*règle d'Or*), et les règles de simplification. Par exemple, les étapes de réécriture pour le premier théorème sont les suivantes:

**LpPreuve 2.7**

```
 (p \=> q) \<=> (p ^ q) \<=> p
{définition de ^}
= (p \or q) \<=> (p \=> q) \<=> p \<=> p \<=> q
{simplification avec bool_str.4}
= T \<=> (p \or q) \<=> (p \=> q) \<=> q
{simplification avec bool_str.3}
= (p \or q) \<=> (p \=> q) \<=> q
{définition de \=>}
= (p \or q) \<=> (p \or q) \<=> q \<=> q
{simplification avec bool_str.4}
= T \<=> q \<=> q
{simplification avec bool_str.3}
= q \<=> q
{simplification avec bool_str.4}
= T
```

• **La négation** : le deuxième point qui nécessite réflexion est la formalisation de la négation. En effet, une traduction systématique des axiomes peut engendrer des erreurs sémantiques critiques. Ayant introduit l'équivalence, la disjonction, la conjonction et l'implication, nous avons besoin de la négation, définie par l'axiome suivant :

$$[\neg(X \equiv Y) \equiv \neg X \equiv Y] \tag{2.4}$$

traduit en un axiome LP

```
nnot(p \<=> q) \<=> nnot(p) \<=> q = T
```

Cet axiome nous permet de prouver les théorèmes suivants :

```
nnot(p) \<=> p = F
nnot(p) \<=> q \<=> p \<=> nnot(q) = T
```

**Remarque 2.1** *Un amalgame entre le langage objet et le méta-langage, c'est à dire entre la négation* `nnot` *et celle de* LP `~`*, peut donner la traduction suivante de 2.4 :*

```
~ (p = q) = (nnot(p)=q)
```

*Cet axiome permet de montrer* `~(p=T) = (p=F)`*, qui exprime le fait que chaque structure booléenne est soit égale à "T" soit égale à "F". Ce qui est sémantiquement incorrect.*

*Lorsque le fait* `nnot(rc) = T` *est valide, (où* `rc` *une constante de type* `Bexp`*), nous pouvons montrer que* `rc = F` *par calcul de paires critiques entre les règles de la base. Par contre, lorsque le fait* `rc = T` *est faux dans le système courant, c'est à dire dans le méta-langage de* LP *(*`rc = T -> false`*), nous ne pouvons pas montrer que* `rc = F`*. Ce qui est sémantiquement correct. En effet, la sémantique de* `nnot(rc) = T` *est que l'évaluation du prédicat* `nnot(rc)` *donne* vrai *dans le système courant. Ceci implique que l'évaluation du prédicat* `rc` *dans le même système donne* faux*. Par contre, la sémantique de* `rc = T -> false` *est que nous n'avons pas pu évaluer* `rc` *à* true*, cela n'implique pas que son évaluation donnerait* faux*.*

De même que pour les opérateurs `^` et `\=>`, nous pouvons utiliser une même structure de preuve pour plusieurs théorèmes :

La base de faits comprend entre autres, les règles suivantes :

```
bool_str.3: p \<=> T -> p
bool_str.4: p \<=> p -> T
...
bool_str.97:  nnot(p \<=> q) \<=> nnot(p) \<=> q -> T
bool_str.99:  nnot(p) \<=> nnot(q) \<=> p \<=> q -> T
bool_str.100: nnot(nnot(p)) \<=> p -> T
```

Ces règles nous permettent de prouver les théorèmes suivants :

```
nnot(p \<=> q) =  nnot(p) \<=> q
nnot(p) \<=> nnot(q) = p \<=> q
nnot(nnot(p)) = p
```

La structure des preuves est identique à celle de 2.4 et 2.6. Elle utilise les deux règles de simplification `bool_str.3` et `bool_str.4` et l'axiome de définition de la négation `bool_str.97`.

Pour chaque théorème, nous prouvons le lemme correspondant en utilisant la règle de réécriture adéquate (Axiome de définition). La preuve du premier est la suivante :

**LpPreuve 2.8**

```
prove nnot(p \<=> q) =  nnot(p) \<=> q
  set name lemme
  make passive bool*
  prove
    nnot(p) \<=> q
     =
   nnot(p \<=> q) \<=> nnot(p)\<=> q \<=> nnot(p) \<=> q
    rewrite conj with bool_str.97
    rewrite conj with bool_str.3
    [] conjecture
  normalize lemme with bool_str.4,bool_str.3
  [] conjecture
qed
```

Ces théorèmes nous permettent d'exprimer l'équivalence comme une égalité de telle façon que les règles qui en découlent puissent être utilisées à leur tour comme des règles de simplification.

Nous avons ainsi modélisé le calcul des structures booléennes (des expressions de prédicats), en traduisant formellement les axiomes de définition des opérateurs logiques, puis en prouvant les propriétés qu'ils doivent vérifier. Le système de réécriture obtenu comprend 140 règles de réécriture.
L'importance de cette formalisation découle du choix des axiomes à introduire et des théorèmes à prouver. Prenons l'exemple de l'opérateur \=>. Une formalisation *ad-hoc* nécessite de choisir comme définition, l'une des expressions équivalentes suivantes :

$X \vee Y \equiv Y$
$X \wedge Y \equiv X$
$\neg X \vee Y$
$\neg X \wedge \neg Y \equiv \neg Y$
$\neg X \vee \neg Y \equiv \neg X$
$\neg(X \wedge \neg Y)$

Une fois le choix effectué, il faut alors prouver les autres expressions comme des théorèmes. Une autre solution serait d'introduire toutes ces expressions comme axiomes, ce qui non seulement engendrerait des redondances mais aussi des bases inconsistantes, si les axiomes sont mal "écrits". D'où l'importance de la règle `bool_str.81` qui définit l'implication par rapport à \or et \<=>, puis d'exiger et donc de prouver que cette opérateur soit réflexif (\or idempotent), transitif (\or associatif), antisymétrique (\or symétrique), et que \or soit monotone par rapport à cet opérateur `(p \=> q) \=> (p \or r \=> q \or r)`. Pour obtenir les autres expressions possibles de l'opérateur \=>, il suffit de les prouver en tant que théorèmes.

### 2.3.3 Mécanisation

L'une des applications de la formalisation de ce calcul est la preuve des propriétés de sûreté. Ces propriétés sont exprimées en fonction de la plus faible pré-condition *wp*. Plus précisément, dans notre modèle, la plupart des preuves se ramènent à la preuve d'un théorème de la forme :

```
P \=> wp(s,Q) = T
```

Cette équation est normalisée, à l'aide de la règle qui exprime \=> en fonction de \or :

```
nnot(P) \or wp(S,Q) = T
```

L'idée intuitive est de supposer P *vrai* et de prouver `wp(s,Q)`, sachant que c'est trivial quand P est *faux*. Une première recherche nous a mené à identifier dans quelle mesure nous pouvons utiliser les méthodes de preuves disponibles en LP. En particulier, étant donné que les théorèmes à prouver sont sous la forme d'une implication, il serait intéressant d'utiliser la méthode d'implication de LP. En effet, un théorème de la forme `A => B = true` où `A` et `B` sont deux booléens, peut être prouvé par la méthode *implique* (cf. paragraphe 1.4.3).

Pour utiliser les méthodes de preuve de LP liées à la forme syntaxique des conjectures, il faut disposer de règles (ou axiomes) qui nous permettrait de "convertir" nos opérateurs logiques (du langage objet) en opérateurs LP (de la méta-théorie).

$$[X \wedge Y] \equiv [X] \wedge [Y] \tag{2.5}$$

Cet axiome est traduit, en une règle de "conversion" :

$R_{et}$ : `(P ^ Q = T) <=> (P = T /\ Q = T)`

sachant que

$$[X \wedge Y] \overset{theorie}{\equiv} [X] \overset{theorie}{\wedge} [Y]$$

Ainsi, la règle $R_{et}$ nous permet de prouver une formule comprenant un ^ en utilisant la méthode *conjonction* de LP. Par contre, pour l'implication, la propriété est plus faible :

$$[X \Rightarrow Y] \Rightarrow ([X] \Rightarrow [Y]) \tag{2.6}$$

formalisé par la règle :

$R_{impl}$ : `(P \=> Q = T) => (P = T => Q = T) -> T`

La règle $R_{impl}$ ne nous permet donc pas d'utiliser la méthode *implique*.

**Remarque 2.2** *Pour une structure booléenne de la forme* `P \=> Q = T`, `Q` *peut contenir* `P` *(par exemple,* `Q` *peut être* `f(P,R)` *où* `f` *a la signature* `Bexp,Bexp -> Bexp`*) et* `P` *peut contenir des "variables" de programmes. Par conséquent supposer* `P` vrai, *veut dire que* `P` *est* vrai *"partout" et non que la variable* `P` *est "égale" à la constante* `T`.
$([P] \equiv true) \neq ([P \equiv true])$

Les règles $R_{et}$, et $R_{impl}$ traduisent la sémantique des opérateurs logique du langage objet. En effet, pour que la conjonction de deux prédicats soit évaluée à vrai, il est nécessaire que l'évaluation de chaque prédicat donne vrai. Par contre, pour que l'évaluation de l'expression de prédicats `P \=> Q` donne *vrai*, nous avons deux cas; Soit l'évaluation de `P` donne *faux*, soit l'évaluation de `P` donne *vrai* ainsi que celle de `Q`. En d'autres termes, "prouver" une expression de la forme `P \=> Q` du langage objet, revient à prouver la formule LP, `P \=> Q = T`. En utilisant la règle $R_{impl}$, cela revient à montrer `P = F`, ou bien `(P = T) => (Q = T)`.

Les règles $R_{et}$ et $R_{impl}$ nous permettent de prouver :

$R_{ouT}$: `(P = T \/ Q = T) => (P \or Q = T)`
$R_{ouF}$: `(P \or Q = F) <=> (P = F /\ Q = F)`

Comment peut on alors automatiser les preuves des formules mettant en jeu les nouveaux opérateurs définis, afin d'utiliser les méthodes de preuve *syntaxiques* fournies par LP ?

Toute formule du langage objet à prouver est préalablement normalisée, c'est à dire que tout théorème sera de la forme `(A ^ B) = T`. LP utilise alors la règle $R_{et}$ pour normaliser la conjecture à `(A=T) /\ (B=T)`. En utilisant la méthode *conjonction*, nous essayons alors de montrer les sous buts `A = T` et et `B = T`. Les sous buts sont de la forme

$$L_1 \backslash\text{or } L_2 \backslash\text{or } \ldots \backslash\text{or } L_m = T$$

où $L_i$ est une structure booléenne ne comprend aucun connecteur "logique" autre que le "nnot".

Comme nous l'avons décrit précédemment, prouver la correction d'un programme UNITY revient à montrer un théorème de la forme :

`(P \=> Q) = T`

Dans le but d'effectuer des preuves "intuitives", la stratégie consiste à éviter la réécriture en `\or` en immunisant la règle correspondante (cf. p.19), puis à utiliser la règle $R_{impl}$. Nous commençons par "essayer" la preuve de `(P = T) => (Q = T)`, qui nous permet d'utiliser la méthode *implique* de LP. Deux cas sont alors possibles : soit la preuve échoue, auquel cas en utilisant $R_{impl}$, nous en déduisons que `P \=> Q = T` est faux, soit la preuve réussit, auquel cas nous utilisons les informations obtenues pour prouver `P \=> Q`.

Ceci achève l'étude de la formalisation du calcul des structures booléennes, que nous appellerons dans la suite "expressions booléennes" ou "expressions de prédicats". Nous allons à présent nous intéresser à la formalisation liée à l'environnement UNITY à savoir les programmes, les affectations,les variables, etc.

## 2.4 Les programmes

Bien que les auteurs d'UNITY aient choisi une théorie pour la programmation parallèle dont la sémantique est basée sur les systèmes de transition, ils ont préféré une représentation des programmes qui rappelle celle utilisée par les langages de programmation. Selon eux, considérer un programme comme un système de transition , c'est à dire un ensemble d'états, un état initial et un fonction de transition, offre peu de méthodologie pour le développement de programme et par conséquent pour la vérification.

Un programme UNITY est constitué de trois parties : la partie *declare* contient la déclaration des variables du programme, la partie *initially* est un ensemble d'équations d'initialisation des variables. La partie *assign* est un ensemble non vide d'instructions, affectations qui peuvent être simples, multiples ou conditionnelles.

**Exemple 2.1 Program** *{expl}*

**Declare** $x : nat, b : boolean, \ldots$
**initially** $x = 1 \,[]\, b = false$
**assign**
$\{xact\}\ x \ := x + 1 \quad if\ x \leq 10 \wedge \neg b$
$\{bact\}\ b \ := true \quad if\ x > 10$
**end** *{expl}*

Pour modéliser un programme UNITY en LP, nous devons définir les notions de *programme*, de *variable*, d'*affectation*, etc.

Pour formaliser les programmes, nous utilisons le type abstrait *liste* défini par la sorte `Actlist` et nous définissons une sorte `Act` pour représenter les actions. Un programme UNITY est donc représenté par une constante de type `Actlist`, au quelle on affecte la liste des *actions* du programme. Par exemple, pour le programme `expl`, nous avons les déclarations LP suivantes :

```
declare operators
expl :-> Actlist
xact :-> Act
bact :-> Act
..
assert
expl = cons(xact,cons(bact,nil))
```

Cette constante représente en fait le "nom" du programme considéré. Toute propriété que doit satisfaire le programme aura comme comme argument cette constante. Par exemple, si le programme `expl` doit satisfaire une propriété de la forme *p unless q*, elle sera formalisée dans le système par `unless(p,q,expl)`. La constante `expl` sera alors remplacée par la liste des actions du programme.

Pour représenter le fait que l'ordre des affectations (actions) est quelconque, le constructeur du type `Actlist` doit vérifier :

$$cons(a1, cons(a2, l1)) = cons(a2, cons(a1, l1))$$

où $a_1$, $a_2$ sont des actions et $l1$ une liste d'actions. Nous verrons l'importance de cette formule pour la mécanisation des preuves.

### 2.4.1 Les variables

La première partie d'un programme UNITY, *declare*, contient les noms des variables et leurs types, et la partie *initially* définit leurs valeurs initiales.

Dans la déclaration du programme *expl*, $x$ et $b$ sont des noms de variables. Dans un modèle d'états-transitions, les valeurs de ces variables représentent à tout moment du calcul, un état du programme, et peuvent être de différents types.
Dans notre modèle de pré et post-condition, nous avons choisi de ne pas référencer de façon explicite les états. Les pré- et post-conditions sont des expressions de prédicats (structure booléennes) pouvant contenir des variables du programme. Par exemple, pour le programme `expl`, une pré-condition peut être de la forme : $\forall y \in N : x \leq (x + y + 1)$. Or, dans cette formule, $x$ est un *identificateur du programme* de type *nat* et $y$ est un *identificateur de variable* de type *nat*.

Par conséquent, nous considérons deux types de variables UNITY, les variables du programme UNITY et les variables apparaissant dans les obligations de preuves UNITY. Pour ce faire, nous définissons deux sortes LP, `Id` pour le premier type de variables et `Id_of_var` pour le second. La sorte `Id` est celle des identificateurs et `Id_of_var`, celle des identificateurs de variables. Les *identificateurs* apparaissent dans le texte du programme. Ce sont des variables d'état ou *variables dynamiques* car elles représentent des quantités qui peuvent changer au cours du temps. Ces variables sont définies comme des *constantes* LP de type `id`. Les *variables de preuves*, ou variables

*statiques*, apparaissent dans la preuve UNITY et sont codées comme des *variables*[15] LP de type `id_of_var`. Ce sont des méta-variables utilisées pour exprimer des propriétés que doit satisfaire le programme. Dans la littérature, les premières sont dites *flexibles*, par opposition aux secondes dites *rigides*. Par exemple, la spécification $\{x = n\}x :=?\{x = n + 5\}$ exprime le fait que pour tout $n$, si $x = n$ alors l'exécution de l'affectation doit affecter une valeur à $x$ de telle sorte que $x = n + 5$ soit valide. $x$ est donc un identificateur et $n$ une variable de preuve.
Les identificateurs d'un programme et les variables de preuves prennent des valeurs d'un certain type, *entier,liste,...*. C'est ce qu'on appelle les données dans un langage de programmation, ou les types abstraits. Nous définissons les types de données : naturels (`Nat`), séquences (`Seq`), enregistrement (`rec`). Les spécifications correspondantes sont définies comme les spécifications LSL [Gut83], modifiées pour axiomatiser nos propres types abstraits. Ces spécifications, ainsi que les preuves associées, seront détaillées au chapitre 3.

Les expressions de prédicats sont construites à partir d'*expressions*, d'opérateurs relationnels et de connecteurs logiques.

Les expressions arithmétiques sont construites à l'aide d'expressions et d'opérateurs arithmétiques :

`__+__,__-__,__*__:Exp,Exp -> Exp`

Une variable de type `id_of_var` ou `id` est une expression de *base*, ainsi qu'un entier, une séquence, etc. L'absence d'inclusion de types en LP nécessite des fonctions de conversions.

## Conversion

L'opérateur `+: Exp,Exp -> Exp` crée une nouvelle expression à partir de deux expressions. Il est clair que cet opérateur diffère de l'opérateur `+: Nat,Nat -> Nat`, mais comme on l'imagine, ces deux opérateurs obéissent aux même lois. Dans le contexte de la sémantique des programmes, "5" est une expression entière, qui ne contient pas de variables. Par ailleurs, les identificateurs apparaissent dans les assertions à prouver et sont aussi des expressions. Étant donné que la logique de LP ne contient pas de sous types, nous avons simulé l'inclusion de `id`, `id_of_var` et `Nat` dans `Exp` à l'aide des fonctions `id_to_exp` et `nat_to_exp`.

- `nat_to_exp`: transforme un naturel en une expression.
- `id_to_exp`: transforme un identificateur en une expression.
- `id_to_nat`: transforme un identificateur en un naturel.
- `id_to_bexp`: transforme un identificateur de booléen en une expression booléenne.
- `seq_to_exp`: transforme une séquence en une expression.

**Remarque**: Nous avons en fait deux fonctions de conversion de même nom `id_to_nat`. L'une a la signature : `Id -> Nat` et l'autre `Id_of_var -> Nat`. Par exemple, $kr$ est un identificateur de naturel de type `id`, $x$ est une identificateur de variable de type `id_of_var` et $id\_to\_nat(kr) < id\_to\_nat(x)$.

Ces fonctions de conversion doivent satisfaire un certain nombre de propriétés: notons $+_E$ un opérateur arithmétique ($\{+, -, *\}$) dont la signature est `Exp,Exp -> Exp` et $+_N$, l'opérateur

15. Les quantificateurs portent sur les variables de preuve.

correspondant dont la signature est `Nat,Nat -> Nat`. De même, $<_E$ représente un opérateur relationnel ($\{<, \leq, \backslash=\}$) dont la signature est `Exp,Exp -> Bexp` et $<_N$, l'opérateur correspondant dont la signature est `Nat,Nat -> Bool`

**Simplification :**

$$\begin{array}{lll}
\forall n, m \in N : & (nat_to_exp(n) \ +_E \ nat_to_exp(m)) & = nat_to_exp(n \ +_N \ m)) \\
\forall n, m \in N : & ((nat_to_exp(n) \ <_E \ nat_to_exp(m)) = T) & = (n \ <_N \ m) \\
\forall n, m \in N : & ((nat_to_exp(n) \ <_E \ nat_to_exp(m)) = F) & = \neg(n \ <_N \ m)
\end{array}$$

**Exemple 2.2** *L'expression* `nat_to_exp(i) + nat_to_exp(s(j))`*, où* `i` *et* `j` *sont deux entiers naturels, est réécrite en* `nat_to_exp(s(i+j))`*. Ces règles nous permettent de simplifier les expressions pour se ramener à des expressions sur les entiers, pour lesquels nous disposons d'un ensemble de règles et de théorèmes.*

Ces règles de simplification existent pour chaque type d'opérandes :

- Les deux opérandes sont de même type `Nat`.(voir précédemment).
- Les deux opérandes sont de même type, identificateur (`Id`), ou bien identificateur de variables (`Id_of_var`) :

$$\begin{array}{lll}
id_to_exp(i_1) \ +_E \ id_to_exp(i_2) & = & nat_to_exp(id_to_nat(i_1) +_N id_to_nat(i_2)) \\
((id_to_exp(i_1) \ <_E \ id_to_exp(i_2)) = T) & = & (id_to_nat(i_1) \ <_N \ id_to_nat(i_2)) \\
((id_to_exp(i_1) \ <_E \ id_to_exp(i_2)) = F) & = & \neg(id_to_nat(i_1) \ <_N \ id_to_nat(i_2))
\end{array}$$

- Les deux opérandes sont de types différents :
  ▷ Entier et identificateur (`Id` ou `Id_of_var`) :

$$\begin{array}{lll}
nat_to_exp(n) \ +_E \ id_to_exp(i_1) & = & nat_to_exp(n +_N id_to_nat(i_1)) \\
((nat_to_exp(n) \ <_E \ id_to_exp(i_1)) = T) & = & (n \ <_N \ id_to_nat(i_1)) \\
((nat_to_exp(n) \ <_E \ id_to_exp(i_1)) = F) & = & \neg(n \ <_N \ id_to_nat(i_1))
\end{array}$$

  ▷ Identificateur (`Id`) et identificateur de variable (`Id_of_var`) :

$$\begin{array}{lll}
id_to_exp(id_1) \ +_E \ id_to_exp(iv_1) = nat_to_exp(id_to_nat(id_1) +_N id_to_nat(iv_1)) & & \\
((id_to_exp(id_1) \ <_E \ id_to_exp(iv_2)) = T) & = & (id_to_nat(id_1) \ <_N \ id_to_nat(iv_2)) \\
((id_to_exp(id_1) \ <_E \ id_to_exp(iv_2)) = F) & = & \neg(id_to_nat(id_1) \ <_N \ id_to_nat(iv_2))
\end{array}$$

**Remarque 2.3** *Si l'opérateur considéré n'est pas commutatif (- par exemple) et les opérandes de types différents, les règles de simplification sont dupliquées.*

Intuitivement, si nous avons un fait de la forme $(nat\_to\_exp(1) = id\_to\_exp(y))$, cela nous indique que l'identificateur $y$ a pour valeur l'entier 1 $(id\_to\_nat(y) = 1)$. Les règles de déduction suivantes permettent à LP de déduire cette information dès qu'un tel fait est vrai dans le système courant :

```
ass
when (nat_to_exp(n)=id_to_exp(id1)) yield (n=id_to_nat(id1));
when (nat_to_exp(n)=id_to_exp(iv1)) yield (n=id_to_nat(iv1));
..
```

**Abréviation :**
$nat\_to\_exp(id\_to\_nat(id1)) = id\_to\_exp(id1)$
$nat\_to\_exp(id\_to\_nat(iv1)) = id\_to\_exp(iv1)$

De façon générale, à chaque sorte $S$ sont associés deux fonctions de conversion et une règle d'abréviation :

- $id\_to\_S$, transforme un identificateur (`Id` ou `Id_of_var`) en un élément de la sorte $S$,
- $S\_to\_exp$, transforme tout élément de la sorte $S$ en une expression (de base),
- $S\_to\_exp(id\_to\_S(id)) = id\_to\_exp(id)$.

### 2.4.2 L'affectation

La partie *assign* d'un programme UNITY est un ensemble non vide d'instructions, et toute instruction est une affectation. Pour formaliser les affectations, nous utilisons quatre opérateurs, correspondants aux quatre types d'affectations UNITY :

- `assg: paire -> Act`: affectation simple avec une paire de type `(id,exp)` ou `(id,Bexp)`. Suivant que `id` soit un identificateur d'entier, ou de booléen, cette fonction lui affecte l'expression `exp` ou bien l'expression booléenne `Bexp`.
- `cond_assg: pair,Bexp -> Act`: affectation conditionnelle où `Bexp` représente la condition.
- `mult_assg: Set -> Act`: affectation multiple, où `Set` est l'ensemble de paires d'affectation `(id,exp)`.
- `cond_mult_assg: Set,Bexp -> Act`: affectation conditionnelle multiple, où `Set` est l'ensemble de paires et `Bexp` la condition.

Une affectation multiple est une affectation parallèle, qui consiste à effectuer les affectations atomiques de façon séquentielle dans un ordre quelconque.

**Exemple**: L'affectation $x, y := e(x), e(y)$ est équivalente à la séquence $(x := e(x); y := e(y))$ ou bien $y := e(y); x := e(x)$ indifféremment. Par contre, l'affectation $(x, y) := (e(x), f(x, y))$ est équivalent à la séquence $y := f(x, y); x := e(x)$ et non à l'inverse. Ce traitement sera détaillé dans le paragraphe consacré aux substitutions.

## 2.5 Le calcul des wp

Les définitions des opérateurs temporels d'UNITY sont basées sur les triplets de Hoare $\{P\}s\{Q\}$, où $s$ représente une instruction d'affectation d'UNITY, $P$ une pré-condition qui doit être vraie avant l'exécution de $s$ et $Q$ une post-condition qui résulte de l'exécution de $s$. Les triplets de Hoare et les règles d'inférence associées ont été définis pour formaliser le raisonnement sur la correction partielle des programmes. Dijkstra a introduit le calcul de *la plus faible pré-condition*, ou calcul des *wp*, comme un outil formel pour la vérification et la construction des programmes à partir des spécifications formelles. Dû à la simplicité du langage de programmation d'UNITY, qui ne comporte qu'une seule construction, le calcul des *wp* associé est considérablement simplifié. Nous avons donc choisi cette approche déductive, qui permet de traiter des domaines infinis (de données), parce qu'elle peut être facilement associée avec un démonstrateur de théorème.

Dans ce calcul, $wlp(s, Q)$ est la plus *faible* pré-condition qui doit être valide avant l'exécution de $s$ de façon à ce que $Q$ soit valide après l'exécution de $s$. Par conséquent, une condition $P$ est une pré-condition suffisante pour $Q$ par rapport à $s$ si et seulement si $P$ est plus *forte* que $wlp(s, Q)$.

$$\{P\}s\{Q\} \quad ssi \quad P \Rightarrow wlp(s, Q)$$

Pour les affectations UNITY, dont l'exécution termine et qui sont déterministes dans le sens où les expressions considérées dans l'affectation sont totales, $wlp(s, Q)$ est équivalente à $wp(s, Q)$. Nous n'allons pas reprendre dans ce document la théorie de la sémantique des langages de programmation définie en terme de pré- et post-conditions. Pour une étude complète, le lecteur peut se reporter à [Dij76, Dij89]. Nous allons dans ce qui suit énoncer les propriétés qui nous sont utiles pour notre formalisation des affectations UNITY.

La plus faible pré-condition pour une affectation $a$ déterministe vérifie les propriétés suivantes :

- (Loi du Miracle exclu) La plus faible pré-condition sous laquelle $a$ établit la post-condition $false$ est équivalente à $false$ (ou bien il n'existe pas d'état à partir duquel l'exécution de $a$ résulte en un état vérifiant le prédicat $faux$) :

$$[wp(a, false) \equiv false] \tag{2.7}$$

- (Distributivité de la conjonction) La plus faible pré-condition sous laquelle $a$ établit la post-condition $P \wedge Q$ est équivalente à la conjonction de la plus faible pré-condition sous laquelle $a$ établit la post-condition $P$ et de la plus faible pré-condition sous laquelle $a$ établit la post-condition $Q$ :

$$[wp(a, P \wedge Q)) \equiv wp(a, P) \wedge wp(b, Q)] \tag{2.8}$$

  Le prédicat valide dans les états à partir desquels l'exécution de $a$ résulte en un état vérifiant $P \wedge Q$ est équivalent à la conjonction du prédicat valide dans les états à partir desquels l'exécution de $a$ résulte en un état vérifiant $P$ et du prédicat valide dans les états à partir desquels l'exécution de $a$ résulte en un état vérifiant $Q$. (Si $a$ n'est pas une affectation alors la terminaison est exigée pour l'un des deux prédicats, c'est à dire $wp$ est remplacé par $wlp$).

- Le prédicat valide dans les états à partir desquels l'exécution de $a$ résulte en un état vérifiant $true$ est équivalent à $true$.

$$[wp(a, true) \equiv true] \tag{2.9}$$

Si on fixe l'instruction $s$, $wp$ est une fonction d'ordre supérieur qui transforme un prédicat en un prédicat : $wp_s : Q \rightarrow wp_s(Q)$. Voici quelques propriétés que doit vérifier $wp$ :

- Règle de Leibniz : $(\forall P, Q, [P \equiv Q] \Rightarrow [wp(P) \equiv wp(Q)])$
- Monotonie : $[wp \ \ monotone \equiv (\forall P, Q, [P \Rightarrow Q] \Rightarrow [wp(P) \Rightarrow wp(Q)])]$
- Conjonction : $[(wp \ conjonctif \ sur \ W) \equiv (wp(\forall P : P \in W : P) \equiv (\forall P : P \in W : wp(P)))]$
- Disjonction : $[(wp \ disjonctif \ sur \ W) \equiv (wp(\exists P : P \in W : P) \equiv (\exists P : P \in W : wp(P)))]$

Nous avons défini l'opérateur *wp* comme étant une fonction ayant deux arguments, une action et une expression de prédicat et qui renvoi une expression de prédicat : `wp:Act,Bexp ->Bexp`. Nous avons simplifier les axiomes de conjonction (vu que notre langage objet ne comprend pas de quantification) à une conjonction finie pour deux prédicats $p$ et $q$. La simplification est identique pour la propriété de disjonction :

```
set name wp_prop
assert
wp(a1,F) = F;
wp(a1,T) = T;
wp(a1,p) ^ wp(a1,q) = wp(a1,p ^ q);                    %conjonction
wp(a1,p) \or wp(a1,q) = wp(a1,p \or q);                %disjonstion
((p \=> q) = T) => ((wp(a1,p) \=> wp(a1,q)) = T)       %monotonie
..
```

La preuve (triviale) de la règle de Leibnitz est la suivante :

```
prove ((p \<=> q) = T) => ((wp(a1,p) \<=> wp(a1,q)) = T)
  resume by => -m
    <>  => subgoal
    critical-pair *hyp with bool_str*
    [] => subgoal
  [] conjecture
qed
```

Nous avons par ailleurs défini les propriétés usuelles des triplets de Hoare, en utilisant *wp*, telle que la règle de disjonction ou la règle de conséquence :

$$\frac{P \Rightarrow R, \{R\}a1\{Q\}, Q \Rightarrow C}{\{P\}a1\{C\}} \tag{2.10}$$

$$\frac{\{P\}a1\{Q\}, \{C\}a1\{D\}}{\{P \wedge C\}a1\{Q \wedge D\}, \{P \vee C\}a1\{Q \vee D\}} \tag{2.11}$$

formalisées en LP :

```
Set name wp_disj_conj
assert
(((p \=> wp(a1,q))=T) /\ ((c \=> wp(a1,d))=T))
 => ((((p \or c) \=> wp(a1, (q \or d)))=T)
    /\ (((p ^ c) \=> wp(a1, (q ^ d)))=T))
..

Set name wp_impl
assert
  (((p \=> r) = T)
   /\ ((r \=> wp(a1,q))=T)
   /\ ((q \=> c) = T))
   => ((p \=> wp(a1,c))=T)
..
```

Enfin, nous avons défini la sémantique de l'affectation à l'aide de l'opérateur *wp* : soit $X := \Sigma \; if \; b$, une affectation où $X$ est une liste d'identificateurs, $\Sigma$ une liste d'expressions pouvant dépendre de $X$, et $b$ une expression booléenne. La sémantique de cette affectation est :

$$\begin{array}{lcl} wp(x := E, Q) & \equiv & Q^E_x \\ wp(x := E \; if \; b, Q) & \equiv & (b \Rightarrow wp(x := E, Q)) \wedge (\neg b \Rightarrow Q) \\ wp(X := \Sigma \; if \; b, Q) & \equiv & (b \Rightarrow Q^\Sigma_X) \wedge (\neg b \Rightarrow Q) \end{array}$$

où $Q_x^E$ est obtenue en substituant dans $Q$ toutes les occurrences de $x$ par $E$. De même, $Q_X^\Sigma$ est obtenue à partir de $Q$ en substituant de façon simultanée les variables $\{x_1 \ldots x_n\}$ dans X par les expressions $\{e_1 \ldots e_n\}$ de $\Sigma$.

En utilisant les fonctions définies au paragraphe 2.4.2, `assg` (affectation simple), `cond_assg` (affectation conditionnelle), `mult_assg` (affectation multiple) et `cond_mult_assg` (affectation multiple conditionnelle), nous formalisons la sémantique de l'affectation par les axiomes suivants :

```
Set name wp

assert
wp(assg(id1.ex),c) = sub_Bexp({id1.ex},c);

wp(assg(id1.b),c)  = sub_Bexp({id1.b},c);

wp(cond_assg(id1.ex,b),p)
      = (b \=> wp(assg(id1.ex),p)) ^ (nnot(b) \=> p);

wp(cond_assg(id1.c,b),p)
      = (b \=> wp(assg(id1.c),p))  ^ (nnot(b) \=> p);

wp(mult_assg(pl),p) = sub_Bexp(pl,p);

wp(cond_mult_assg(pl,b),p)
      =  (b \=> wp(mult_assg(pl),p)) ^ (nnot(b) \=> p);
..
```

La fonction `sub_Bexp(list,p)`, où `list` est la liste des paires `(id,exp)`, substitue chaque occurrence de l'identificateur `id` par `exp` dans l'expression booléenne `p`.

## 2.6 Les substitutions textuelles

Dans les systèmes de transitions modélisant les programmes UNITY, les affectations représentent les transitions du système d'un état à l'autre. Pour raisonner sur les transitions dans un modèle syntaxique où la notion d'état est absente, nous avons défini la sémantique de l'affectation à l'aide de l'opérateur *wp*, en termes de substitutions. Plus généralement, la substitution textuelle permet de raisonner sur l'affectation, dans n'importe quel langage de programmation utilisant l'affectation. En effet, la "définition" de l'affectation utilise la substitution textuelle, indépendamment de l'opérateur *wp* :

$$\{R[x := E]\}\ x := E\ \{R\}$$

Par conséquent, la substitution est l'opérateur principal de la formalisation de UNITY dans un modèle syntaxique et peut être considérée comme un opérateur qui transforme un prédicat (ou une assertion) en un prédicat.

La formule représentée par le triplet de Hoare comporte des variables liées et des variables de programmes mais pas de variables libres. Dans notre système, les variables liées sont de types `id_of_var` et les variables de programmes sont de type `Id`. Il est important de distinguer entre les variables LP, variables logiques qui ont une valeur quelconque d'un certain type, et les variables d'un programme UNITY (les identificateurs de programmes ou bien les variables de preuve). Cette différence est cruciale pour pouvoir formaliser les substitutions nécessaires aux affectations. Reprenons les différentes étapes qui mènent à l'application de la substitution :

```
  p \=> wp(a,q)
{a ≡ x:=E}
  p \=> sub_Bexp({x.E},q)
```

Nous définissons la substitution textuelle sur la structure des expressions et sur celle des expressions booléennes. Les expressions booléennes (prédicats) sont construites à partir d'expressions et d'opérateurs logiques ou relationnels. Par conséquent, la substitution est définie de façon structurelle à l'aide de deux opérateurs de substitution :

```
sub_exp :Set,Exp  -> Exp
sub_Bexp:Set,Bexp -> Bexp
```

`Set` représente l'ensemble des couples de la forme `(id.ex)`, correspondant à la substitution de `Id` par `ex`.

Quelques axiomes de substitutions sont :

```
assert
sub_exp({},ex) = ex;
sub_exp(pl,ex1+ex2) = sub_exp(pl,ex1) + sub_exp(pl,ex2);
...
sub_Bexp({},p) = p;
sub_Bexp(pl,T) = T;
sub_Bexp(pl,F) = F;
sub_Bexp(pl,p ^ q) = sub_Bexp(pl,p) ^ sub_Bexp(pl,q);
...
sub_Bexp(pl,ex1 < ex2) = sub_exp(pl,ex1) < sub_exp(pl,ex2);
```

Nous procédons de façon similaire pour les expressions et les structures booléennes construites avec les opérateurs arithmétiques ou logiques.

Nous utilisons le fait que LP permet de déclarer un opérateur comme étant associatif et/ou commutatif. *Set* est alors un ensemble de couples sur lequel l'union est un opérateur associatif et commutatif. Nous obtenons alors une liste non ordonnée de couples d'identificateurs et expressions :

```
Set name Set_of_pair_of_exp

declare variables x, y, z: Set,pp:Paire
declare operators
  {}:                -> Set
  insert:    Paire, Set -> Set
  {__}:      Paire -> Set
  __\u__   :  Set, Set  -> Set
  __\in__:       Paire, Set -> Bool
  ..
assert
   sort Set generated by {}, {__},\u;
   sort Set partitioned by \in;
   ac \u
..

assert
  {} \u x = x;
  {pp} = insert(pp,{});
  insert(pp,x)= {pp} \u x;
  ~(pp \in {});
  pp \in {p1}<=>pp=p1;
  pp \in insert(p1,x)<=>pp = p1 \/  pp \in x;
  ..
```

Cette représentation a la propriété qu'une variable peut apparaître plus d'une fois dans la partie gauche d'une affectation. La gestion d'une telle possibilité demeure sous la responsabilité du programmeur, qui doit s'assurer que pour une chaque variable, une valeur unique lui est attribuée [Cha88].

### 2.6.1 Les substitutions de base

Étudions comment se déroule la substitution pour des formules "de base", ce qui nous permettra de justifier l'existence d'une sorte `Id` pour représenter les identificateurs du programme et `Id_of_var` pour représenter les variables de preuve. Soit la propriété suivante à prouver, où `x` et `z` sont des variables du programme et $y$ une variable de preuve auxiliaire :

$$\{\mathtt{x} + y \geq 1 + \mathtt{z}\}\ \mathtt{x} := \mathtt{x} + 1\ \{\mathtt{x} + y \geq 1 + \mathtt{z}\}$$

Il faut donc substituer dans la post-condition `x` par `x + 1` tout en préservant inchangée la valeur de $y$ et de `z`. Dans notre système, `x` et `z` sont de type `Id` et $y$ est de type `Id_of_var`.

#### Pourquoi une sorte Id?

Un programme UNITY peut utiliser des variables représentant des entiers, des booléens, des listes, etc. Le premier argument de l'opérateur de substitution est un ensemble de paires, formées d'un identificateur et de l'expression à lui substituer. Une paire est construite avec l'opérateur "`.`" Par conséquent, on peut se demander de quel type doit être le premier argument d'une paire :

```
__.__: ?, Exp -> Paire
```

Nous avons deux possibilités, soit définir l'opérateur "`.`" pour chaque type de donnée, vu qu'il est possible d'utiliser le même nom pour différents opérateurs :

```
__.__: Nat,Exp  -> Paire
__.__: Seq,Exp  -> Paire
__.__: Bool,Exp -> Paire
```

soit utiliser une même sorte pour toutes les variables apparaissant dans le texte du programme. Si nous choisissons la première solution, nous aurons des substitutions de la forme :

```
sub_exp({n.ex},nat_to_exp(m))
 = if (n=m) then ex else nat_to_exp(m)
sub_exp({s1.ex},seq_to_exp(s2))
 = if (s1=s2) then ex else seq_to_exp(s1)
```

Cette solution n'est pas correcte dû à la comparaison[16]. Dans l'exemple introductif, nous devons substituer à `x`, l'expression `x + 1` dans l'expression `1 + z`. Cela revient à effectuer la substitution dans les deux sous expressions et en particulier dans l'expression `nat_to_exp(s(0))`. Nous allons donc alors tester si `x = nat_to_exp(s(0))`, or nous ne pouvons rien affirmer puisque la variable du programme `x` peut valoir `1` au cours de l'exécution du programme. Le problème est identique pour les autres types de variables. La comparaison doit porter sur le nom de la variable et non sur sa valeur. D'où la nécessité de définir une sorte `Id` représentant les variables du programmes, ou *identificateurs*.

16. Si on considère qu'un type de donnée n'est autre qu'un ensemble de valeurs, et d' opérations associées, cette comparaison porte sur "des valeurs".

### Pourquoi une sorte Id_of_var?

Reprenons notre exemple introductif où nous devons substituer à `x`, l'expression `x+1` dans l'expression $x + y$. Cela revient à substituer cette expression dans les deux sous expressions et en particulier dans l'expression $y$. Cette variable n'est pas une variable du programme, ni une constante entière. C'est une variable auxiliaire qui représente un entier, utilisée pour spécifier la propriété à montrer. Supposons que nous déclarons $y$ comme une variable de type `Nat`. La substitution de `x` par $x + 1$ dans l'expression $y$ doit nous retourner $y$. Par conséquent, une des règles de substitution doit être :

```
sub_exp({id.ex},nat_to_exp(m)) =  nat_to_exp(m)
```

Or, outre les opérateurs arithmétiques $+, -, *$, nous utilisons d'autres fonctions pour les expressions qui retournent un entier, en particulier, la fonction `len` qui calcule la taille d'une séquence (liste) :

```
len:Seq -> Nat
```

Supposons que l'on veuille substituer à une liste $l$, la liste $l \vdash e$ dans l'expression $len(l)$. En utilisant la règle précédente, nous aurons

```
sub_exp({l.(l ⊢ e)},nat_to_exp(len(l))) = nat_to_exp(len(l))
```

Ce qui est incorrect puisque le fait d'ajouter un élément à la liste n'a pas modifié sa taille.

Ce problème est identique pour toutes les fonctions dont la signature est `Nat -> Nat` car elles s'appliquent aussi aux identificateurs représentant un entier que l'on transforme à l'aide de l'opérateur `id_to_nat`. Par conséquent, nous ne pouvons pas représenter les variables auxiliaires comme étant des *variables* LP de type `Nat` (ou bien `Seq`, `Bool`).

Peut on alors les déclarer comme des variables de type `Id`? Supposons que $y$ soit déclarée comme telle, et que nous allons substituer à `x`, l'expression `x+1` dans l'expression $x+y$. Cela revient à substituer dans les deux sous expressions et en particulier dans l'expression $y$, représentée dans notre langage objet par `nat_to_exp(id_to_nat(y))`. Cette expression sera normalisée par la règle d'abréviation (cf. paragraphe 2.4.1) en `id_to_exp(y)`. Pour effectuer la substitution, nous allons alors utiliser une règle de la forme :

```
sub_exp({id1.ex},id_to_exp(id2))
  = if (id1=id2)
    then ex
    else id_to_exp(id2);
```

Nous sommes alors confrontés au même problème que précédemment, à savoir que nous ne pouvons affirmer `¬(x = y)`. En effet, `x` et `y` sont deux variables LP de type `Id`, l'équation `(x = y)` (ou `¬(x = y)`) est *incohérente* (cf. p.14). Il faudrait alors déclarer `x` et $y$ comme étant deux constantes de type `Id` et introduire l'axiome `¬(x = y)` dans la preuve. Par conséquent, un ensemble d'axiomes de cette forme devra être introduit pour tout identificateur du programme considéré et pour toute variable de preuve dont on pourrait avoir besoin.

```
prove
   (id_to_exp(x) \= (id_to_exp(y)-nat_to_exp(1)))
    \=>  wp(assg(x.nat_to_exp(s(id_to_nat(x)))),
            (id_to_exp(x) \= id_to_exp(y))) = T
  ..
  assert ~ (x = y)
  [] conjecture
qed
```

Cette solution n'est donc pas satisfaisante, dû à l'introduction de ce type d'axiome dans les preuves, d'autant plus que les variables de preuves ne sont pas forcément connues dès la spécification du programme. De plus, il nous semble inadéquat de comparer deux objets de nature différente, seules leurs valeurs peuvent être comparées. Nous avons alors décider de préciser la différence sémantique entre un identificateur apparaissant dans le texte du programme et une variable auxiliaire dont nous avons besoin pour spécifier les obligations de preuves.

Nous définissons donc une sorte `id_of_var` qui représente l'ensemble des variables de preuve et la substitution est définie comme étant l'*identité* sur des expressions réduites à des variables de type `Id_of_var`, de telle sorte que la substitution n'opère pas sur les variables de preuve.

Considérons à présent la substitution de `x` par `x+1` dans la post-condition $(1+\mathtt{z})$, représenté dans le système par l'expression `nat_to_exp(1) + id_to_exp(z)`. Cela revient à effectuer la substitution dans les deux membres de l'expressions et en particulier dans l'expression `id_to_exp(z)`. La règle correspondante est donc de la forme :

```
sub_exp({id1.ex},id_to_exp(id2))
     = (if (id1=id2) then ex else id_to_exp(id2));
```

Le test sur les identificateurs nécessite donc une comparaison pour pouvoir effectuer la substitution. Dans l'exemple, nous avons besoin de comparer `x` et `z` qui sont deux identificateurs du programme, pour pouvoir effectuer la substitution. Nous ne pouvons pas affirmer que $\mathtt{x} \neq \mathtt{z}$ (ou bien $\neg(\mathtt{x} = \mathtt{z})$), même si ce sont deux constantes de type `Id`, dû au fait qu'au cours du processus, la *valeur* de l'identificateur `x` peut être égale à celle de l'identificateur `z`. En fait "`x` " et "`z`" ne sont que des chaînes de caractères représentant le *nom* de la variable. Par conséquent, pour effectuer la substitution, il suffit de tester l'égalité des noms.

Nous définissons la sorte `Id` comme étant générée par les constructeurs `firstId` et `nextid`, de telle façon que tout identificateur soit écrit en fonction de ces deux constructeurs.

```
Declare operators
 firstId:     -> Id
 nextid : Id -> Id
..
assert sort Id generated freely by firstId,nextid
```

Cette règle d'induction structurelle introduit deux formules en plus du principe d'induction (cf. chapitre 1, page 16) :

```
~(firstId = nextid(id));
nextid(id1) = nextid(id2) <=> id1 = id2
```

Par analogie à un interpréteur qui transformerait de façon automatique le texte d'un programme UNITY en un script LP, nous affectons au premier identificateur "lu", la constante $firstId$, au second, $nextid(firstId)$, etc. Ceci nous assure que les noms des identificateurs sont distincts pour effectuer les comparaisons nécessaires à la substitution.

**Récapitulatif**: sur ce, que peuvent être les expressions de base dans lesquelles s'effectue la substitution?

- Les substitutions dans les expressions de la forme $nat\_to\_exp(n)$ sont explicitées cas par cas. En effet, nous ne pouvons pas écrire $sub\_exp(\{id.ex\}, nat\_to\_exp(n)) = nat\_to\_exp(n)$, du

fait que $n$ peut être de la forme $s(id\_to\_nat(id))$, ou bien plus généralement de la forme $f(id)$ où $f$ à la signature `Id -> Nat`. Auquel cas, il nous faut substituer $id$ par sa valeur.

- $nat\_to\_exp(n)$ avec $n$ constante entière: $0, s(0), s(s(0)), \ldots$, la substitution agit comme l'identité.
- $nat\_to\_exp(id\_to\_nat(id))$ : les fonctions d'abréviation normalisent cette expression à $id\_to\_exp(id)$.
- $nat\_to\_exp(id\_to\_nat(iv))$ : idem.
- $nat\_to\_exp(f(v_1, v_2, \ldots, v_n))$, où $f$ à la signature :$S_1, S_2, \ldots, S_n$ -> $Nat$. La règle de substitution est alors de la forme :

$$
\begin{aligned}
&sub\_exp(id.ex, nat\_to\_exp(f(Id_1, Id_2, \ldots, Id_n))) \\
&\quad = nat\_to\_exp(f(exp\_to\_S_1(sub\_exp(id.ex, S_1\_to\_exp(Id_1))), \\
&\qquad\qquad exp\_to\_S_2(sub\_exp(id.ex, S_2\_to\_exp(Id_2))), \\
&\qquad\qquad \ldots \\
&\qquad\qquad exp\_to\_S_n(sub\_exp(id.ex, S_n\_to\_exp(Id_n))) \\
&\qquad\qquad ) \\
&\qquad ) \\
&\quad )
\end{aligned}
$$

**Exemples:**

`pos: Exp,Seq -> Nat`, calcule la position d'un élément dans une séquence indexée.

```
sub_exp({id1.ex},nat_to_exp(pos(ex2,seq1)))
  = nat_to_exp(
       pos(sub_exp({id1.ex},ex2),
           exp_to_seq(sub_exp({id1.ex},seq_to_exp(seq1)))))
```

où la fonction triviale `exp_to_exp` n'apparaît pas, du fait que le premier argument de `pos` est une expression.

`len:Seq -> Nat` retourne la taille de la séquence.

```
sub_exp({id1.ex},nat_to_exp(len(seq1)))
  = nat_to_exp(
       len(exp_to_seq(sub_exp({id1.ex},seq_to_exp(seq1)))))
```

`.:record,Nat -> Nat` est une fonction qui retourne la valeur contenue dans le champ de type *entier naturel* d'un enregistrement.

```
sub_exp({id1.ex},nat_to_exp(rec1.ind))
  = nat_to_exp(
       exp_to_rec(sub_exp({id1.ex},rec_to_exp(rec1))).ind)
```

`s:Nat -> Nat:`

```
sub_exp({id1.ex},nat_to_exp(0)) = nat_to_exp(0);
sub_exp({id1.ex},nat_to_exp(s(n))
  = nat_to_exp(
       s(exp_to_nat(sub_exp({id1.ex},nat_to_exp(n)))))
```

• Les substitutions dans les expressions de la forme $id\_to\_exp(id)$ s'effectue en comparant les "noms" des identificateurs :

```
sub_exp({id1.ex},id_to_exp(id2))
    = (if (id1=id2) then ex else id_to_exp(id2));
```

• Les substitutions de base doivent prendre en compte les différentes structures de données, mais s'effectuent sur des objets de type **expression**. Toute structure de donnée susceptible d'être manipulée par un programme doit être convertie en un objet de type expression. Il est donc nécessaire de définir les substitutions pour les expressions construites à l'aide des opérateurs de conversion dont la signature est de la forme `S_to_exp: S -> Exp`. Nous avons vu, dans ce qui précède, le cas où $S$ représente les entiers naturels (`Nat`) ou bien les identificateurs (`Id`), mais à toute sorte définie correspond une fonction de conversion.

**Exemples:**

`tail: Seq -> Seq`: retourne la séquence au quelle on a ôté le premier élément:

```
sub_exp({id1.ex},seq_to_exp(tail(seq1)))
  = seq_to_exp(tail(exp_to_seq(sub_exp({id1.ex},seq_to_exp(seq1)))));
```

`⊢: Seq,Exp -> Seq`, ajoute un élément à la séquence:

```
sub_exp({id1.ex},seq_to_exp(seq1 +- exp1))
 = seq_to_exp(exp_to_seq(sub_exp({id1.ex},seq_to_exp(seq1)))
              +- sub_exp({id1.ex},exp1));
```

`!: Nat,Exp -> record`, construit un enregistrement à l'aide d'un entier et d'une expression. $n$`!`$exp1$ représente un élément de type `rec`, un enregistrement composé d'entier et d'expression.

```
sub_exp({id1.ex},rec_to_exp(n ! exp1))
  =  rec_to_exp(n ! sub_exp({id1.ex},exp1));
```

• Enfin, le dernier cas consiste à substituer dans les expressions de base construites sans opérateurs de conversion.

**Exemples:**

`head: Seq -> Exp`, retourne la tête de la séquence et `last: Seq -> Exp`, le dernier élément :

```
sub_exp({id1.ex},head(seq1))
  = head(exp_to_seq(sub_exp({id1.ex},seq_to_exp(seq1))));
sub_exp({id1.ex},last(seq1))
  = last(exp_to_seq(sub_exp({id1.ex},seq_to_exp(seq1))));
```

`[__]: Seq,Nat -> Exp`, retourne le énième élément d'une séquence :

```
sub_exp({id1.ex},seq1[n])
  = exp_to_seq(sub_exp({id1.ex},seq_to_exp(seq1)))[n];
```

• Les substitutions dans les *expressions booléennes* de base sont définies de façon similaire. Les expressions booléennes de base outre `T` et `F`, peuvent être des objets construits à partir des identificateurs du programme représentant des variables (du programme) booléennes :

**Exemple:**

```
x: Nat
y: Nat
z: bool
*corps du prog*
```

z est donc un identificateur du programme, que l'on déclare comme une constante de type `Id`, et qui peut être utilisé dans une formule du type $z \Rightarrow P$.
Par exemple, l'assertion $\{\mathtt{z} \wedge (\mathtt{x} > 0)\}\, \mathtt{x}, \mathtt{z} := \mathtt{x} + 1, \mathit{false}\, \{\mathtt{x} > 0 \wedge \neg \mathtt{z}\}$, peut être une propriété à montrer au cours de la vérification

Pour cela, nous définissons une fonction de conversion dont la signature est `id_to_Bexp: Id -> Bexp`, pour prendre en compte les identificateurs de variables booléennes. La règle de substitution correspondante est alors

```
assert
sub_Bexp({id1.b}, id_to_Bexp(id2))
  = (if (id1 = id2) then  b else id_to_Bexp(id2));
```

De même que pour les identificateurs de variables de type *entier*, nous pouvons avoir des variables de preuves qui représentent des booléens. Elles sont alors déclarées de type `Id_of_var`, sur lesquelles la substitution agit comme l'identité :

```
sub_Bexp({id1.b}, id_to_Bexp(iv2)) = id_to_Bexp(iv2);
```

Par ailleurs, nous disposons d'opérateurs qui retournent une expression booléenne, dont la signature est du type $S_1, \ldots, S_n$ -> *Bexp*. Par exemple, l'opérateur `inseq:Exp,Seq -> Bexp` teste l'appartenance de l'élément `exp` dans une séquence. La règle de substitution est alors :

```
sub_Bexp({id1.ex},inseq(exp1,seq1))
  = inseq(exp1,exp_to_seq(sub_exp({id1.ex},seq_to_exp(seq1))));
```

Enfin, substituer à un identificateur une expression $e$ dans une expression booléenne réduite à un identificateur de booléen retourne cet identificateur :

```
sub_Bexp({id1.ex},id_to_Bexp(id2)) = id_to_Bexp(id2);
sub_Bexp({id1.ex},id_to_Bexp(iv2)) = id_to_Bexp(iv2);
..
```

### 2.6.2 Les affectations multiples

Les affectations multiples nécessitent que les substitutions s'effectuent dans un ordre quelconque. En d'autres termes, le résultat doit être le même quelque soit l'ordre choisi pour affecter aux variables les expressions correspondantes. Comme les affectations multiples utilisent un ensemble de couple $(id, exp)$, et la substitution est définie de façon structurelle, l'ordre des affectations est quelconque. En effet, le terme dans lequel la substitution doit s'effectuer est décomposé jusqu'à obtenir une expression de base et ceci avant d'effectuer la substitution. De plus, les règles de substitution sont définies inductivement sur l'ensemble des couples de l'affectation :

```
sub_exp({id1.ex},id_to_exp(id2))
     = (if (id1=id2) then ex else id_to_exp(id2));

sub_exp({id1.ex}\u pl,id_to_exp(id2))
     = (if (id1=id2) then ex else sub_exp(pl,id_to_exp(id2)));
```

La substitution n'est effective que si l'expression est réduite à l'un des cas de base cités précédemment.

Nous pouvons ainsi prouver la propriété

$$\{(x = 1) \wedge (y = 2)\}\, x, y := y, x\, \{x = y + 1\}$$

où la liste des substitutions est représentée par une ensemble de couples $(id, exp)$. Cette assertion est traduite en

$$((x = 1) \wedge (y = 2)) \Rightarrow wp([x, y := y, x], x = y + 1)$$

Étudions comment se déroule l'affectation :

$$\begin{aligned}
&((x = 1) \wedge (y = 2)) \Rightarrow wp(mult_assg(\{(x, y)\} \cup \{y, x\}), x = y + 1)\\
&((x = 1) \wedge (y = 2)) \Rightarrow sub_bexp(\{x, y\} \cup \{y, x\}, x = y + 1)\\
&((x = 1) \wedge (y = 2)) \Rightarrow (sub_exp(\{x, y\} \cup \{y, x\}, x) = sub_exp(\{x, y\} \cup \{y, x\}, y + 1))\\
&((x = 1) \wedge (y = 2)) \Rightarrow (sub_exp(\{x, y\} \cup \{y, x\}, x)\\
&\qquad\qquad = sub_exp(\{x, y\} \cup \{y, x\}, y) + sub_exp(\{x, y\} \cup \{y, x\}, 1))\\
&((x = 1) \wedge (y = 2)) \Rightarrow (sub_exp(\{x, y\} \cup \{y, x\}, x) = sub_exp(\{x, y\} \cup \{y, x\}, y) + 1)\\
&((x = 1) \wedge (y = 2)) \Rightarrow (y = sub_exp(\{x, y\} \cup \{y, x\}, y) + 1)\\
&((x = 1) \wedge (y = 2)) \Rightarrow (y = sub_exp(\{y, x\}, y) + 1)\\
&((x = 1) \wedge (y = 2)) \Rightarrow (y = x + 1)\\
&true
\end{aligned}$$

Le résultat serait incorrect si la substitution était appliquée directement au terme $x = y + 1$ quelque soit l'ordre choisi. En effet, supposons que la substitution s'effectue sur la formule non décomposée :

$$\begin{aligned}
&((x = 1) \wedge (y = 2)) \Rightarrow sub_exp(\{x, y\} \cup \{y, x\}, x = y + 1)\\
&((x = 1) \wedge (y = 2)) \Rightarrow sub_exp(\{y, x\}, y = y + 1)\\
&((x = 1) \wedge (y = 2)) \Rightarrow (x = x + 1)
\end{aligned}$$

Ce qui est faux. De même si nous choisissons un ordre inverse :

$$\begin{aligned}
&((x = 1) \wedge (y = 2)) \Rightarrow sub_exp(\{y, x\} \cup \{x, y\}, x = y + 1)\\
&((x = 1) \wedge (y = 2)) \Rightarrow sub_exp(\{x, y\}, x = x + 1)\\
&((x = 1) \wedge (y = 2)) \Rightarrow (y = y + 1)
\end{aligned}$$

La preuve LP de l'assertion est la suivante :

```
declare operators x,y: -> Id        %x,y deux variables d'un programme P
assert
x = firstId;
y = nextid(firstId)
..

prove
(((id_to_exp(x)\= nat_to_exp(1))^ (id_to_exp(y) \= nat_to_exp(2))) = T)
 => ((wp(mult_assg({x.id_to_exp(y)} \u ({(y.id_to_exp(x))}\u {})),
              (id_to_exp(x)\= (id_to_exp(y)+ nat_to_exp(1))))) = T)
  resume by => -m
    <> => subgoal
    [] => subgoal
  [] conjecture
qed
```

La stratégie de réduction des termes peut être modifiée. Dans notre système, la stratégie utilisée est celle par défaut, `outside-in`, qui consiste à réduire un terme en essayant de réécrire tout le terme avant d'essayer de réécrire les sous termes. Mais nous avons aussi à notre disposition

la stratégie **inside-out**, où LP essaye de réécrire les sous-termes avant de réécrire le terme. Par exemple, avec la stratégie par défaut, nous avons les étapes réécriture :

$sub\_exp(\{x, s(x)\}, x + 0)$
$sub\_exp(\{x, s(x)\}, x) + sub\_exp(\{x, s(x)\}, 0)$
$s(x) + sub\_exp(\{x, s(x)\}, 0)$
$s(x) + 0$
$s(x)$

Si nous utilisons la stratégie **inside-out**, les étapes sont :

$sub\_exp(\{x, s(x)\}, x + 0)$
$sub\_exp(\{x, s(x)\}, x)$
$s(x)$

Néanmoins, nous préférons la stratégie par défaut qui nous permet de subdiviser les expressions jusqu'a obtenir des expressions de base, sur lesquelles les erreurs sont plus faciles à détecter. En effet, avec cette stratégie, l'introduction d'un axiome incorrect dans une preuve peut fausser la substitution, ce qui est difficile à détecter. Par exemple, pour exprimer un fait que nous pensons valide au cours d'une vérification, nous introduisons un axiome de la forme $y + 1 = 1$. Supposons maintenant que pour prouver une assertion donnée, nous devons effectuer une substitution dans une expression de la forme $(y + 1) + z$ :

$sub\_exp(\{y, x\}, (y + 1) + z)$
$sub\_exp(\{y, x\}, 1 + z)$
$1 + z$

alors qu'elle devrait être :

$sub\_exp(\{y, x\}, (y + 1) + z)$
$(x + 1) + z)$

En conclusion, les axiomes, les théorèmes ou plus généralement les faits, relatifs à l'application en cours de vérification, ne doivent pas être utilisés comme des règles de simplification lors de la substitution. En effet, la substitution textuelle (comme son nom l'indique) ne doit utiliser que la forme syntaxique d'une expression et non sa sémantique. Cette différence est cruciale lorsqu'il s'agit de retrouver l'origine d'une erreur pour une preuve dont la mécanisation a échouée. Ce qui est l'un des buts fondamental de ce travail.

## 2.7 Les opérateurs temporels

Les opérateurs temporels sont des fonctions d'ordre supérieur dont les arguments sont des prédicats. Rappelons la définition de la relation *unless*.

Pour un programme $P$ :

$$p \ unless \ q \equiv \langle \forall s : s \in P :: \{p \wedge \neg q\} s \{p \vee q\} \rangle$$

Soit en utilisant le *wp*:

$$p \ unless \ q \equiv \langle \forall s : s \in P :: (p \wedge \neg q) \Rightarrow wp(s, (p \vee q)) \rangle$$

Étant donné que le démonstrateur du LARCH est un langage du premier ordre, la formalisation des prédicats par des *expressions de prédicats* nous permet de définir les opérateurs temporels par des fonctions du premier ordre, dont les arguments sont deux expressions de prédicats de type `Bexp` et un programme de type `Actlist` :

```
declare operators
unless,ensures      :Bexp,Bexp,Actlist -> bool
leads_to,exist_Act  :Bexp,Bexp,Actlist -> bool
wp                  :Act,Bexp          -> Bexp
```

La spécification LP des opérateurs temporels est opérationnelle, car elle définit une fonction par sa méthode de calcul.

**Sûreté** :

```
Set name unless
assert
 unless(p,q,anil) = true
 unless(p,q,l(a)) = (((p ^ nnot(q)) \=> wp(a,p \or q))=T)
 unless(p,q,cons(a,act_l)) = unless(p,q,a) /\ unless(p,q,act_l)
```

La définition de l'opérateur *unless* est inductive par rapport à la liste d'actions `cons(a,act_l)`, qui représente la liste de toutes les actions du programme[17]. Pour une instruction atomique *a*, *unless(p,q,a)* exprime le fait que si *p* est valide et *q* est non valide dans un certain état du programme, alors après l'exécution de *a*, *q* est valide ou *p* est encore valide. Par conséquent, par induction sur le nombre d'instructions du programme à exécuter , on montre que *p* reste valide au moins jusqu'a ce que *q* le soit.
Nous verrons que le schéma d'induction défini pour la sorte `Actlist` nous permet de prouver les règles du système de preuve d'UNITY par induction sur la variable représentant le programme.

L'invariant d'un programme est un prédicat valide à l'état initial et qui doit être *préservé* par toutes les transitions du programme. C'est donc un prédicat `stable`, valide à l'état initial. En d'autres termes, le prédicat *I* est un invariant du programme si le prédicat qui représente l'état initial implique *I* et *I* est vrai dans tous les états à moins que le prédicat *false* ne le soit :

```
Set name Invariant
assert
 Inv(p,pgm,init_cond) = ((init_cond \=> p)=T) /\ unless(p,F,pgm)
```

**Progrès** Les propriétés de progrès forment un sous ensemble des propriétés de *vivacité*. Alors qu'une propriété de sûreté exprime ce que le programme *ne doit pas faire*, les propriétés de progrès exprime ce que le programme *doit* accomplir. La propriété de base pour exprimer le progrès en UNITY est *ensures*. Comme son nom l'indique, c'est une propriété qui "garantit" que le programme vérifiera une certaine propriété. *p ensures q* est valide pour un programme *P* si et seulement si dès que le prédicat *p* est valide, il reste vrai jusqu'à ce que *q* le soit et il doit exister une transition qui garantit la validité de *q*. En particulier, un programme ne contenant aucune action ne peut rien garantir. Par conséquent, une propriété de vivacité ne peut être vérifiée que si la liste des actions représentant le programme est non vide. Nous définissons la relation *Ensures* comme suit :

```
Set name ensures
```

17. Cette définition peut être écrite uniquement avec le premier et le troisième axiome dans lequel on spécifie la propriété pour une action du programme. Pour des raisons de lisibilité, nous utilisons le deuxième axiome pour expliciter *unless* pour une action du programme.

```
assert
 ensures(p,q,anil) = false
 ensures(p,q,pgm) = unless(p,q,pgm) /\ exists_act(p,q,pgm)
..
Set name exist_act
assert
 exists_act(p,q,anil) = false
 exists_act(p,q,a) = (((p ^ nnot(q)) \=> wp(a,q)) = T)
 exists_act(p,q,cons(a,act_l))
  = exists_act(p,q,a) \/ exists_act(p,q,act_l)
```

La fonction *exists_act* cherche l'action qui établit la validité du prédicat $q$. Elle teste l'existence d'une action $a$ dans *pgm* vérifiant l'assertion $\{p \wedge \neg q\}a\{q\}$. Du point de vue opérationnel, un programme *pgm* vérifie la propriété *ensures(p,q,pgm)* si et seulement si dès que le programme atteint un état vérifiant $p$, tout état successeur vérifiera p, jusqu'à ce qu'un état vérifie $q$. Du fait que la définition de `ensures` utilise celle de `unless`, cette relation est aussi définie inductivement sur la liste des actions du programme.

La relation *ensures* permet de définir une relation plus puissante, à savoir *leads_to*. Un programme a la propriété $p$ *leads_to* $q$ si et seulement si dès que le prédicat $p$ est valide alors le prédicat $q$ sera fatalement valide "plus tard". Formellement, la relation *leads_to* est la fermeture transitive et disjonctive de la relation *ensures*. Plus précisément, un programme $P$ a la propriété $p$ *leads_to* $q$ si et seulement si on peut dériver cette propriété en appliquant un nombre fini de fois les règles d'inférence suivantes :

$$LEADS\_TO : \quad \frac{p \ \ ensures \ \ q}{p \ \ leads\_to \ \ q}$$

$$TRANS\_LEAD : \quad \frac{p \ \ leads\_to \ \ r, r \ \ leads\_to \ \ q}{p \ \ leads\_to \ \ q}$$

$$DISJ\_LEAD : \ For \ \ any \ \ set \ \ W, \quad \frac{(\forall m \in W : p(m) \ \ leads\_to \ \ q)}{(\exists m \in W : p(m)) \ \ leads\_to \ \ q}$$

Par conséquent, à l'inverse de *unless* et *ensures*, la définition formelle de la relation *leads_to* n'est pas opérationnelle. Pour les relations *unless* et *ensures*, la propriété doit être vérifiée par rapport à toutes les transitions du programme. Par contre, la validité du prédicat $p$ ne doit pas être préservée par toutes les transitions du programme.

Considérons le modèle basé sur les séquences d'exécution du programme : Soit $S$ une séquence dont $S_i$ est le i-ème élément, composé d'une paire formée d'un état $S_i.e$ et d'une étiquette $S_i.t$. $S_i.e$ est l'état du programme atteint par l'exécution de l'instruction $S_{i-1}.t$ dans l'état $S_{i-1}.e$. $S_i.t$ est l'instruction choisie pour être exécutée dans l'état $S_i.e$. L'état initial de $S$ est noté $S_0.e$. Dû au caractère déterministe des affectations d'UNITY, pour tout $S$ et tout $i \geq 0$, $S_{i+1}.e$ est déterminé de façon unique avec $S_i.e$ et $S_i.t$. La contrainte d'équité d'UNITY exige que pour tout $S$ et tout instruction $s$, $S_i.t = s$ pour un nombre infini d'indice $i$.

Dans ce modèle, $\{p\}a\{q\}$ s'écrit :

$$(p[S_i] \wedge (S_i.t = a)) \Rightarrow q[S_{i+1}]$$

où $p[S_i]$ représente le fait que le prédicat $p$ est valide dans l'état $S_i$.*e.*

*p unless q* s'écrit :

$$\forall S, \forall i : (p \wedge \neg q)[S_i] \Rightarrow (p \vee q)[S_{i+1}]$$

*p ensures q* s'écrit:

$$\forall S, \forall i : p[S_i] \Rightarrow (\exists j : i \leq j :: q[S_j] \wedge (\forall k : i \leq k \leq j :: p[S_k]))$$

*p leads_to q* s'écrit :

$$\forall S, \forall i : p[S_i] \Rightarrow (\exists j : i \leq j :: q[S_j])$$

Ce modèle montre que pour prouver *p unless q*, il suffit de montrer que pour tout état où le prédicat $p \wedge \neg q$ est valide, quelle que soit l'instruction choisie, l'état résultant doit vérifier $p \vee q$. Le "pour tout" nous a permis de définir la relation *unless* comme une propriété à vérifier sur toutes les actions de la liste représentant le programme. De plus, cette définition ne fait pas mention des états du programme, car le prédicat $p$ doit rester valide jusqu'a ce que $q$ le soit.
Pour montrer *p ensures q*, le processus est le même avec une propriété supplémentaire à monter : il existe une action dans la liste qui permet d'établir $q$. Pour cela, il suffit de parcourir la liste des actions, en essayant de montrer la propriété pour chaque action de la liste.
Par contre, pour montrer *p leads_to q*, nous ne pouvons pas raisonner sur les actions du programme. En effet, la sémantique de cet opérateur dit que la validité de $p$ dans un état quelconque garantit la validité de $q$ dans un état futur, sans pour autant exiger que $p$ demeure valide tant que $q$ ne l'est pas. En d'autres termes, pour prouver qu'un programme $P$ possède la propriété *p leads_to q*, il faut raisonner sur les états atteints par le programme et les transitions effectuées d'un état à l'autre. Ceci nécessite donc une définition en terme de *trace* d'exécution [Gol90].

Une autre définition du *leads_ to* serait en termes de relation : *p leads_to q* est la relation qui mène par les transitions du programme, d'un ensemble d'états satisfaisant le prédicat $p$ à l'ensemble des états satisfaisant $q$. Cette relation est alors définie comme étant la fermeture transitive et disjonctive de la relation *ensures*. Pour formaliser ces concepts, nous aurions utilisé un principe similaire à celui utilisé pour définir les objets "expressions de prédicat". Nous aurions défini des objets de type "expressions de relation", à l'aide d'une sorte `Relation` et des variables `r1,r2` de type `Relation`. Sur cette sorte, nous aurions défini les opérateurs d'union, d'inclusion, d'intersection, ainsi qu'un ordre partiel, etc. La relation *ensures* serait alors définie comme une constante de type `relation` et la relation *leads_ to* serait `ensures+`.

Nous avons choisi une définition plus simple, car notre objectif est la vérification mécanisée de la correction des programmes UNITY. En effet, la formalisation de ces concepts dans notre langage objet fait que les relations temporelles de UNITY auraient pour codomaine `Bexp` et non plus `Bool`. Par conséquent, le raisonnement et les preuves seraient plus compliqués et plus lourds à gérer dans notre modèle axiomatique.

Nous définissons l'opérateur *leads_ to* en utilisant trois axiomes sous forme d'implication :

```
Set name leads_to
assert
 when ensures(p,q,pgm) yield leads_to(p,q,pgm)
 (leads_to(p,r,pgm) /\ leads_to(r,q,pgm)) => leads_to(p,q,pgm)
 (leads_to(p,r,pgm) /\ leads_to(q,r,pgm)) => leads_to(p \or q,r,pgm)
```

Pour pouvoir inférer $leads\_to(p, q, pgm)$ dès que $ensures(p, q, pgm)$ est valide dans le système courant, nous utilisons une règle de *déduction*, sémantiquement équivalente à une implication mais dont elle diffère du point de vue opérationnel (cf. chapitre 1, p.17).

**Point fixe d'un programme** Le point fixe représente un état stable atteint par le programme. Cet état est représenté par un prédicat, tel que l'exécution de n'importe quelle instruction du programme ne modifie pas la validité de ce prédicat. Par conséquent, un point fixe n'est pas un opérateur, comme *unless*, *ensures*, etc. C'est une expression de prédicats, qu'on calcule en utilisant une fonction récursive sur les actions du programme.

```
Set name Pointfixe
fp(l(assg(id1.ex))) = rec({id1.ex});
fp(l(cond_assg(id1.ex,b))) = (nnot(b) \or rec({id1.ex}));
fp(l(mult_assg(pl))) = rec(pl);
fp(l(cond_mult_assg(pl,b))) = (nnot(b) \or rec(pl));
fp(cons(a1,al)) = fp(l(a1)) ^ fp(al);
fp(nil) = T;
..
assert
rec({}) = T ;
rec({id1.ex}) = (id_to_exp(id1) \= ex);
rec({p1}\u pl) = rec({p1}) ^ rec(pl);
```

**Un principe d'induction pour *leads_to*** Chandy et Misra proposent un principe d'induction pour l'opérateur *leads_to*, basé sur l'observation que dans la plupart des programmes concurrents, montrer une propriété de progrès revient a exhiber une certaine quantité qui "décroît". Ce principe est défini par la règle d'inférence suivante [Cha88] :

$$\textbf{Induction :} \quad \frac{\forall m : m \in W :: p \wedge M = m \; leads\_to \; (p \wedge M <_w m) \vee q}{p \; leads\_to \; q}$$

$W$ est un ensemble bien fondé pour la relation $<_w$ et $M$ est une fonction (la *métrique*) d'états du programme à $W$. L'hypothèse de cette règle d'inférence est que partant d'un état où $p$ est valide et où la valeur de $M = v$, l'exécution du programme atteint fatalement un état où $q$ est valide, ou bien un état où $p$ est valide avec une valeur $v\prime$ de la métrique $M$ inférieure à $v$. Étant donné que la valeur de la métrique ne peut décroître infiniment, on atteint fatalement un état où $q$ est valide.

La fonction $M$ et l'ordre dépendent du programme à prouver. Par conséquent, $M$ et l'ordre correspondant doivent être considérés comme des paramètres à instancier. Ce qui n'est pas possible en LP, comme dans la plupart des démonstrateurs automatiques. Par exemple, $M$ peut être une fonction binaire $M \equiv f(x, y) = (x, y)$, avec un ordre lexicographique sur des couples d'entiers, ou une fonction unaire $f(x) = x + 2$ avec l'ordre $<$ sur les entiers.

Nous avons défini un ordre lexicographique sur des listes d'expressions arithmétiques. Dans la prémisse de la règle d'inférence, le résultat de la comparaison apparaît comme un argument de la fonction *leads_to*. Il doit donc être du type de ses arguments, c'est à dire `Bexp` (et non `bool`). Nous formalisons cet ordre à l'aide d'une fonction dont la signature est `<<:exp_list,exp_list -> Bexp`.

```
Set name lexico
assert
enil << exp_list1 = T;
```

```
exp_list1 ≪ enil = (exp_list1 == enil);
cons(ex1,exp_list1) ≪ cons(ex2,exp_list2)
 = (if ((exp_list1 == enil)=T) /\ ((exp_list2 == enil)=T)
    then ex1 < ex2
    else ((ex1 < ex2) \or ((ex1 \= ex2) ^ (exp_list1 ≪ exp_list2))))
..

assert
(exp_list1 == enil) = (if (exp_list1=enil) then T else F);
(enil == exp_list1) = (if (exp_list1=enil) then T else F);
(cons(ex1,exp_list1) == cons(ex2,exp_list2))
  = ((ex1 \= ex2) ^ (exp_list1 == exp_list2))
..
```

où `exp_list1` et `exp_list2` sont des listes d'expressions, `ex1` et `ex2` des expressions. La fonction `==:exp_list,exp_list->Bexp` teste l'égalité de deux listes d'expressions.

Le principe d'induction est alors formalisé avec LP comme suit :

```
Set name IND_leads_to
assert
leads_to((p ^ (exp_list1 == exp_list2)),
         (p ^ (exp_list1 ≪ exp_list2)) \or q,
          pgm)
 => leads_to(p,q,pgm)
```

A l'aide de ce principe, nous avons prouvé le corollaire suivant :

$$\frac{p \wedge M = m \ \ leads\_to \ \ M < m}{true \ \ \ leads\_to \ \neg p}$$

formalisé en LP :

```
Set name induc_princ2
assert
 leads_to(p ^ (l1 == l2), (l1 ≪ l2),pgm)
  => leads_to(T,nnot(p),pgm)
```

Nous verrons dans les chapitres suivants les applications du principe d'induction.

## 2.8 Correction

### 2.8.1 La formalisation d'Unity

Nous avons formalisé UNITY avec un modèle syntaxique basé sur les pré- et post-conditions, où les programmes sont représentés comme des systèmes de transition et les propriétés sont vérifiées pour tout état de l'espace défini par le programme. Les définitions formelles des opérateurs temporels d'UNITY sont alors définis comme des relations entre les transitions d'états du programme en utilisant les pré- et post-conditions. En utilisant ces définitions, tout état vérifiant une pré-condition d'une propriété doit mener à un état vérifiant la post-condition de la propriété. Par conséquent, les prédicats sont implicitement quantifiés par rapport à tous les états possibles définis par le domaine des données du programme.

Dans ce modèle, les opérateurs temporels *unless* et *ensures* sont définis inductivement sur la structure du programme, ce qui permet de prouver la propriété pour tout les états possibles du programme. La relation *leads_to* est définie par trois règles d'inférence. Il est évident que

pour fournir un fondement théorique à UNITY, la définition de *leads_to* est incomplète, puisque nous ne l'avons pas définie comme étant la fermeture transitive de la relation *ensures*. Notre formalisation de UNITY est donc incomplète (nous ne pouvons pas prouver avec les règles de notre système tout ce qui est prouvable par la logique d'UNITY, par exemple, comme nous le verrons dans le chapitre suivant, les théorèmes que vérifient la relation *leads_to*). Cependant, nous verrons dans les chapitres consacrés aux applications que cela n'intervient pas dans la vérification des propriétés des programmes que nous étudierons. Autrement dit, l'incomplétude est une limitation théorique qui n'a pas d'incidences pratiques.

Pour définir les types de donnée, nous avons utilisé le style des spécifications LSL. Nous avons étendu ces spécifications pour prendre en compte les types de donnée dont nous avions besoin et produire des définitions formelles des concepts de bases des problèmes traités dans nos études de cas. Un effort supplémentaire a été nécessaire pour simplifier la traduction dans une forme acceptable par le démonstrateur.

Néanmoins, il ne suffit pas d'écrire des spécifications claires et concises pour effectuer des preuves correctes. En effet, si une spécification comprend un ensemble d'axiomes introduits plus par commodité que par soucis de correction, la validité de la vérification n'offre pas de garanties suffisantes. Ainsi en B, les règles d'induction sont introduites sans être vérifiées par le système, ce qui peut être dangereux si une règle incorrecte est introduite. L'utilisation d'une méthode incrémentale qui consiste à introduire une base restreinte d'axiomes acceptés par le sens commun, puis à enrichir cette base par des théorèmes qui sont prouvés à mesure qu'on les introduit, permet non pas de garantir totalement la correction de la spécification, mais tout au moins de se prémunir considérablement contre l'introduction d'erreurs.

Pour définir le calcul des prédicats sur lequel repose le modèle que nous avons choisi, nous avons appliqué cette méthodologie. Nous nous sommes aussi basé sur des définitions usuelles qui ont fait leur preuves comme celles proposées par Dijkstra dans le cadre de la sémantique des langages de programmation. Partant d'un ensemble minimal d'axiomes, cette démarche consiste à définir les nouveaux opérateurs de façon incrémentale. Chaque opérateur est introduit par un axiome mettant en jeu les opérateurs précédemment définis et toute définition supposée équivalente est démontrée comme un théorème. Sous l'hypothèse que les méthodes de preuves sont consistantes et que les preuves elle mêmes n'introduisent pas d'inconsistance, toute formule prouvée est une conséquence logique de l'ensemble d'axiomes. Nous verrons dans les chapitres suivants le détail de ces preuves.

Le wp_calcul nous permet de raisonner sur la structure syntaxique du programme à prouver, plutôt que sur les traces d'exécution. Nous verrons dans la suite de ce document que les preuves ne font pas appel à des notions d'exécution de programme, par opposition aux preuves dans le système NQTHM de Goldschlag dans [Gol90]. Notre approche se rapproche plutôt de celle de Brown et Mery décrite dans [Bro93].

### 2.8.2 L'axiome de substitution

Le but de ce travail n'étant pas de fournir un modèle théorique pour UNITY, nous n'allons pas discuter de la consistance de la théorie proposée. Néanmoins, nous désirons tout de même discuter de l'axiome de substitution défini dans la logique d'UNITY.

Dans [San91], l'auteur montre que l'axiome de substitution [Cha88] utilisé avec les trois prédicats temporels produit un système inconsistant. Depuis, plusieurs travaux ont été menés pour résoudre le problème d'inconsistance généré par l'introduction de cet axiome [San91], [Kna94], [Pra94], [Mis90], dont certains proposent une re-définition des opérateurs afin de restreindre les

états aux seuls états accessibles.

L'axiome de substitution est une généralisation de la règle de Leibnitz, "si $x = y$ est un invariant du programme alors on peut remplacer $x$ par $y$ dans toute propriété du programme". En particulier, si $I$ est un invariant de programme, alors on peut remplacer dans une propriété $I$ par *true* et inversement.

**Exemple:** Soit le programme *Expl* à une seule instruction,

**Program** {Expl}
**Declare** $x$ : *integer*
**initially** $x = 1$
**assign**
$\{xact\}\ x := x + 1 \quad if\ \neg(x = 1)$
**end**{Expl}

Ce programme vérifie les propriétés suivantes :

$\{x = 2\}\quad xact\quad \{\neg(x = 1)\}$
$init \Rightarrow (x = 1) \wedge \{x = 1\}\quad xact\quad \{x = 1\}$

Maintenant, si nous appliquons l'axiome de substitution, $x = 1$ étant un invariant, nous pouvons le remplacer par *true* dans la première propriété. Nous obtenons $\{x = 2\}\quad xact\quad \{false\}$, soit $stable(x = 2, expl)$. Or si nous appliquons la définition de *stable*, cette propriété sera réécrite en :

$\{x = 2\}\quad x := x + 1\ \ if\ \neg(x = 1)\quad \{x = 2\}$
$(x = 2) \Rightarrow wp((x := x + 1, if\ \neg(x = 1)), x = 2)$
$(x = 2) \Rightarrow ((\neg(x = 1) \Rightarrow (s(x) = 2)) \wedge ((x = 1) \Rightarrow (x = 2)))$
$(x = 2) \Rightarrow ((x = 1) \vee (s(x) = 2))$

Ce qui est faux. Le problème provient du fait que $x = 1$ est un invariant n'implique pas que $x = 1$ soit valide dans tous les états possibles du programme (les états possibles pour le programme sont ceux définis par le domaine de ses variables) mais uniquement dans les états accessibles par le programme. Or le seul état accessible par le programme *Expl* est précisément l'état caractérisé par le prédicat $x = 1$.

Notre formalisation ne comprend pas l'axiome de substitution qui exprime la propriété suivante (où *Op* est un opérateur UNITY) :

$$\frac{p = q, Op(p, r, prg)}{Op(q, r, prg)}$$

Dans notre formalisation, deux prédicats sont égaux s'ils le sont pour tout état (égalité syntaxique). En effet, la preuve de la règle précédente est triviale dans notre système :

```
((p = q) /\  unless(p,r,pgm)) => unless(q,r,pgm)
```

Un prédicat $I$ est un invariant du programme si et seulement si `inv(I,prg,init) = true` et non pas `I = T`. Par conséquent, le remplacement (réécriture) de $I$ par *true* dans une formule de la forme $I \wedge J$ n'est pas légal dans notre système. En effet, la formule `inv(I,prg,init)` est réécrite en `true` et non le terme `I`.
Comme conséquence, le théorème suivant, qui découle de l'axiome de substitution :

$$\frac{unless(p, q, prg), invariant(\neg q, prg, init)}{stable(p, prg)}$$

ne peut pas être prouvé dans notre système car la preuve nécessite l'axiome suivant (axiome de substitution) :

$$Subt\_ax\_Consq : \quad \frac{invariant(\neg q, prg, init)}{\neg q = true}$$

En effet, la preuve est la suivante, où nous avons introduit l'axiome *Subt_ax_Consq* avec la clause `assert` :

```
set name Subt_ax_Consq
prove (unless(p,q,pgm) /\  inv(nnot(q),pgm,init)) => stable(p,pgm)
  resume  by induction on pgm
    <> basis subgoal
    [] basis subgoal
    <> induction subgoal
    res by =>m
      <> => subgoal
      assert inv(nnot(qc),pgmc,initc) => (nnot(qc)=T)
      prove qc = F
        cri-p  Subt_ax_Consq with bool_str*
        [] conjecture
      [] => subgoal
    [] induction subgoal
  [] conjecture
```

L'axiome utilisé n'est pas valide pour tous les états du programme considéré mais uniquement pour les états accessibles du programme à partir de *init*. `I = T` exprime le fait que le prédicat `I` est valide dans tous les états possibles. Avec la notation utilisée au paragraaphe 2.3, cela veut dire que $[I]$ est différent de $inv(I, pgm, init)$. En effet, nous avons l'axiome trivial suivant :

$$\frac{[I]}{inv(I, pgm, init)}$$

L'inverse n'est pas valide pour tous les états mais uniquement pour les états accessibles à partir de l'état initial *init*. En conclusion, pour introduire l'axiome de substitution, il nous faut définir une logique où les opérateurs d'UNITY sont définis uniquement par rapport aux états accessibles du programme.

La logique d'UNITY sans l'axiome de substitution est incomplète (avec des conditions initiales arbitraire) [Kna94]. Reprenons l'exemple décrit dans [Kna94] : de façon informelle, on prend pour modèle des arbres infinis étiquetés (graphes orientés acycliques) dont les noeuds représentent les états et les arcs sont les transitions effectuées par les actions du programme. Il y a un arc $s$ entre $u$ et $v$ si et seulement si exécuter $s$ dans l'état $u$ résulte dans l'état $v$. De plus, pour tout état initial $r$, il existe un arbre de racine $r$. $e \models p$ si et seulement si $p$ est valide dans l'état $e$. Pour un programme $P$ avec une condition initial $ic$, $(ic, P) \models \Phi$ si et seulement si pour toute racine $r$ validant $ic$, $\Phi$ est vrai dans tous les noeud de tous les chemins dont la racine est $r$. Le théorème de complétude pour *unless* dans ce modèle nous donne

$$(true, Prog) \vdash P \ unless \ Q \ ssi \ (true, Prog) \models P \Rightarrow A(P \ W \ Q)$$

où $A$ est la modalité "pour tous les chemins" ($\Box$) et "W" est "à moins que" ($U$).

Prenons maintenant le programme dont l'unique action $s$ est $x := 2 \ \ if \ \neg(x = 0)$. Alors, partant de l'état initial $x = 0$, nous pouvons construire un arbre tel que dès que $x = 0 \vee x = 1$ est vrai, il le reste à moins que *false* ne devienne vrai :

$$x = 0 \longrightarrow x = 0 \longrightarrow x = 0 \ldots$$

Comme cet arbre est le seul que l'on peut construire à partir de la racine $x = 0$, on a la formule temporelle $x = 0 \vee x = 1 \Rightarrow A(x = 0 \vee x = 1\ W\ false)$. Par contre, la formule UNITY, $x = 0 \vee x = 1$ *unless* $false$, qui lui correspond, ne peut être prouvée pour ce programme. Par conséquent nous avons une formule valide dans le modèle mais qui ne peut être déduite par la logique d'UNITY.

La logique d'UNITY sans conditions initiales et sans l'axiome de substitution est correcte et relativement complète au sens de Cook [Coo78]).

**La logique d'UNITY avec des conditions initiales et sans l'axiome de substitution est incomplète mais correcte. Or c'est précisément ce qui nous intéresse puisque notre travail s'inscrit particulièrement dans la vérification des preuves et non dans la recherche des preuves.**

## 2.9 Conclusion

Nous avons décrit dans ce chapitre la formalisation et la mécanisation d'UNITY. Nous avons montré que l'axiomatisation à l'aide d'un démonstrateur de théorème du premier ordre est assez naturelle pour permettre un raisonnement syntaxique et des preuves formelles. En fait, cette simplicité est obtenue par une sophistication du codage. L'axiomatisation des types abstraits utilisés, tels que les naturels ou bien les séquences sont une extension des spécifications LSL [Gut93]. Ceci est très important pour la réutilisation du système, par la communauté LARCH ainsi que pour les spécifieurs en général. En effet, pour une meilleure utilisation ultérieure du système, les spécifications algébriques sont un atout majeur car elle permettent une formulation concise et claire.

L'un des reproches formulé contre les méthodes formelles est que le langage et la logique utilisés par les démonstrateurs de théorèmes sont difficiles à comprendre et à écrire. Nous avons vu que les spécifications du langage LARCH ne requièrent pas un lourd bagage logique, le style des spécifications algébriques repose sur des concepts syntaxiques simples. De plus, le caractère principal d'une spécification LARCH est d'être indépendante de l'implémentation. Ceci rappelle un deuxième aspect d'UNITY à savoir la séparation entre l'algorithme et l'architecture cible. Il faut noter que la complexité de nos spécifications est due principalement à l'inclusion explicite des sortes les unes dans les autres, à cause de l'absence de polymorphisme en LP. Par conséquent, nous avons concentré nos efforts sur la lisibilité, par une correspondance entre les noms des fonctions et leur sémantique.

L'un des reproches émis contre LP est la simplicité de l'interface pour définir des théories, c'est à dire l'absence de méta-langage. Or la philosophie à la base de la conception de LP est tout d'abord la simplicité. Le méta-langage risque de le compliquer. Par ailleurs, notre but est justement de montrer que malgré cette apparente simplicité, les méthodes proposées sont puissantes et que la partie nécessaire du méta-langage n'engendre pas de complexité. De plus, cela rejoint notre choix d'UNITY comme environnement de spécification qui, malgré une logique assez simple, est suffisamment puissant pour traiter de nombreux exemples. De plus LP ne nécessite pas l'apprentissage d'un autre formalisme (comme LISP pour NQTHM, où ML pour HOL). Par contre, l'absence de méta-langage peut être considérée comme un frein pour définir des tactiques et des stratégie de preuve. Nous verrons par la suite que ce problème peut être contourné par des techniques assez simples.

Par ailleurs, les méthodes formelles, indépendamment de la vérification, sont à la base d'un besoin de communication entre diverses communautés. En effet, pour la réutilisation et la maintenance des systèmes informatiques de taille importante, il est nécessaire que le formalisme utilisé soit compréhensible par une large (et surtout hétérogène) communauté de spécifieurs et d'utilisateurs. Par contre, un formalisme spécialisé et spécifique à une application particulière peut nuire au système, en ralentissant le transfert de technologie et la vivacité. Ces considérations ont été étudiées par les concepteur du LARCH pour proposer un formalisme simple et des outils conviviaux.

Le deuxième reproche contre les méthodes formelles porte sur la vérification, à savoir que prouver des théorèmes est considéré comme une tâche difficile. Cette tâche est complexe mais non compliquée, dès lors que la preuve est considérée comme un objet formel, qui, par analogie à un programme, doit être conçu, codé et corrigé. Par conséquent, les méthodes de vérification doivent être à la fois nombreuses, puissantes et relativement simples. Le développement du démonstrateur LP est basé sur le fait que la première tentative de vérification est souvent infructueuse. Partant de ce principe, il faut alors pouvoir corriger une preuve d'une propriété sans pour autant reprendre toute la vérification du système considéré. Ceci est crucial quand il s'agit de prouver la correction de programmes concurrents, qui sont, de par leur nature, complexes.

Le chapitre suivant est consacré aux preuves effectuées sur les types de données tels que les naturels et les séquences. Nous verrons par la suite les améliorations qui portent sur l'axiomatisation des opérateurs temporels d'UNITY et nous terminerons cette partie consacrée à la formalisation, par une méthodologie de preuve basée sur notre expérience dans l'utilisation de LP. La deuxième partie de ce document est consacrée aux applications de la mécanisation d'UNITY. Plus précisément, nous présentons la preuve d'un protocole de communication à travers des canaux défectueux [Che95b], la preuve d'un programme de contrôle d'un ascenseur [Che95a], ainsi que celle du problème des lecteurs-rédacteurs.

# Chapitre 3

# Spécification et Vérification de la base initiale

## Introduction

Une des exigences que nous nous sommes fixées, lors de l'élaboration de ce travail, est la réutilisation des méthodes formelles définies pour la spécification et pour la vérification. La réutilisation concerne aussi bien les spécifications que la méthodologie de vérification utilisée. A la différence du système NQTHM [Boy79, Boy88] ou HOL [Gor93], le démonstrateur du LARCH ne contient aucune théorie prédéfinie. Par conséquent, une première étape de conception consiste à construire une base de "connaissances", à savoir une théorie LP ou bien un ensemble de théories définissant les objets de base sur lesquels repose la vérification. Néanmoins, pour garantir un minimum de cohérence à cette vérification, il est nécessaire d'assurer un minimum de correction de ces spécifications, qui est la condition nécessaire de la correction des résultats que l'on va obtenir.

La notion de "correction" d'une spécification est une notion qui n'est pas complètement formalisable. En effet, la question de savoir si un système spécifié possède le comportement désiré ne peut être formalisable que dans la mesure où l'on dispose d'une description précise (formelle) des propriétés attendues du système considéré. Mais là est précisément le rôle de la spécification, et si l'on exclut quelques exemples triviaux directement issus des mathématiques, il est bien évident que l'on ne dispose pas en général d'une autre description formelle du système avec laquelle comparer sa spécification, mais simplement d'un cahier des charges issu d'une analyse des besoins.

S'il semble aussi problématique de réussir à formaliser la notion de correction d'une spécification, il est néanmoins souhaitable de savoir au moins donner des critères de correction d'une spécification. Deux critères correspondent à des interrogations naturelles sur une spécification, la **consistance** (*cohérence*) et la **complétude**. Intuitivement, la complétude d'une spécification correspond à la question de savoir si "tout est spécifié" ou si les axiomes de la spécification "suffisent" à décrire le système. La consistance correspond elle, à la question de savoir si les axiomes de la spécification ne sont pas contradictoires. Pour définir la base de connaissance et plus précisément les types abstraits de données dont nous avons besoin, nous avons utilisé les spécifications LSL [Gut83, Gut93] des types abstraits que nous avons étendues ou modifiées pour prendre en compte nos propres types de données.

Le but de ce chapitre est de décrire d'une part, l'utilisation des spécifications LSL et d'autre part les différentes méthodes de preuves disponibles en LP. Il peut être vu comme une illustra-

tion de l'utilisation des techniques proposées pour effectuer les preuves ainsi que des difficultés et complexités que l'on peut rencontrer. Par ailleurs, nous n'allons pas utiliser les facilités de LaTeXpour l'écriture des scripts LP car nous préférons détailler les preuves telles qu'elles sont présentées à l'utilisateur.

## 3.1 Les obligations de preuves

Pour vérifier la consistance de nos spécifications, nous utilisons la sémantique des spécification LSL. Chaque *trait* `Lsl` comprend, outre un ensemble d'axiomes, un ensemble de propriétés à prouver (obligations de preuves). Les obligations de preuves associées à une spécification LSL sont de trois types : *La consistance*, qui consiste à vérifier qu'une spécification n'est pas contradictoire, la *theory containment*, qui consiste à vérifier que la spécification induit les conséquences prévues et enfin *la complétude relative*, à savoir qu'un ensemble d'axiome est bien défini.

La consistance ne peut être spécifiée, car c'est une assertion sur ce que ne contient pas la théorie. Néanmoins, elle est implicitement requise pour toute spécification : celle-ci ne doit pas contenir l'équation `true = false`. Les deux autres types d'obligations de preuves sont spécifiées avec les mots clés **implies** et **assumes**.

La clause `Implies` n'ajoute pas de nouvelles propriétés à la théorie définie par le trait. Par contre, elle exprime des propriétés sur la spécification. Cela permet d'inclure des propriétés redondantes, c'est à dire les propriétés que l'on pense avoir définies dans la spécification. Cette information redondante peut être de deux types, soit des axiomes, pour exprimer les propriétés que l'on pense être dans la théorie, soit une propriété introduite avec le mot clé **converts**, pour exprimer des conditions suffisantes (syntaxiques, portant sur la forme des axiomes) pour assurer *la complétude suffisante* [Gut78]. La clause **assumes** introduit une condition sous laquelle le trait peut être inclus dans un autre. Par exemple, si la condition **assumes** d'un trait $T1$ est $C$, alors tout trait $T2$, qui inclurait $T1$, doit vérifier la condition $C$.

Nous allons décrire l'utilisation de ces techniques de spécification et de vérification pour la construction de la base de règles définissant les entiers naturels, les séquences, et les opérateurs temporels.

## 3.2 Les entiers et leurs preuves

### 3.2.1 La base initiale

La formalisation des entiers est basée sur la spécification LSL correspondante. Nous commençons par définir les opérateurs usuels sur les naturels :

```
set name Nat

declare sort Nat
declare variables x,y,z,i,j,n,m: Nat
declare operators
  0,1,2                    :          -> Nat
  s                        : Nat      -> Nat
  __+__,__*__,__-__        : Nat, Nat -> Nat
  __<=__,__<__,__>=__,__>__: Nat,Nat -> Bool
  ..
% informations pour l'orientation

register height * >  + >  2 >  1 >  s >  0 % for noeq-dsmpos ordering
```

```
register polynomial 0     2,          2 % for polynomial 2 ordering
register polynomial 1     5,          5
register polynomial 2     8,          8
register polynomial +     x + y + 1,  x*y
register polynomial s     x + 2,      x + 2
register polynomial *     x*y,        x*y

% Axiomes

assert
  ac +;
  ac *;
  sort Nat generated freely by 0, s;          % Nat.3
  well founded < ;                            % Nat.4
  i + 0 = i;
  i + s(j) = s(i + j);
  i * 0 = 0;
  i * s(j) = (i * j) + i;
  i * (j + n) = (i * j) + (i * n);
  1 = s(0);
  2 = s(1);
..
```

Cette spécification définit deux schémas d'induction. Le premier schéma est celui usuel pour les entiers, à savoir qu'ils sont générés par 0 et $s$. Le mot clé `freely` (cf. paragraphe 1.3.3) permet à LP d'introduire les axiomes :

```
Nat.3.1: 0 = s(n) -> false
Nat.3.2: s(n1) = s(n2) <=> (n1 = n2)
```

Le deuxième schéma d'induction, `Nat.4`, définit < comme étant une relation bien fondée (p.17) sur les naturels.

La figure 3.1 résume la construction de la base des entiers naturels. En particulier, chaque rectangle représente un *trait* définissant une spécification et chaque flèche représente les propriétés que doit satisfaire la spécification. Ainsi, le trait `TotalOrder` inclut le trait `Istotal` et le trait `DerivedOrder`.

### 3.2.2 Propriétés sur les ordres

La spécification `Naturels(Nat)` *inclut* celle des opérateurs arithmétiques, `ArithOps(Nat)`, qui elle, *suppose* (assumes) un ordre total sur `Nat`, `TotalOrder(Nat)`. D'après la sémantique de la clause `assumes`, nous devons inclure le trait `TotalOrder(Nat)`, et montrer que les assertions de `TotalOrder(Nat)` sont induites par celles de `Naturels(Nat)` et celles de `ArithOps(Nat)`. Ceci peut être fait syntaxiquement car `Naturels(Nat)` inclut `TotalOrder(Nat)`.

Nous allons définir les ordres usuels sur les entiers, à savoir <= comme étant un ordre total sur la sorte `Nat`, c'est à dire inclure la spécification `TotalOrder(Nat)`.

```
assert
 (i <= i);                                  %reflexif
 (i <= j /\ j <= n ) => (i <=  n);          %transitif
 (i <= j /\ j <= i)  <=> (i=j);             %antisymetrique
 (i <= j \/ j <= i);                        %total
..
```

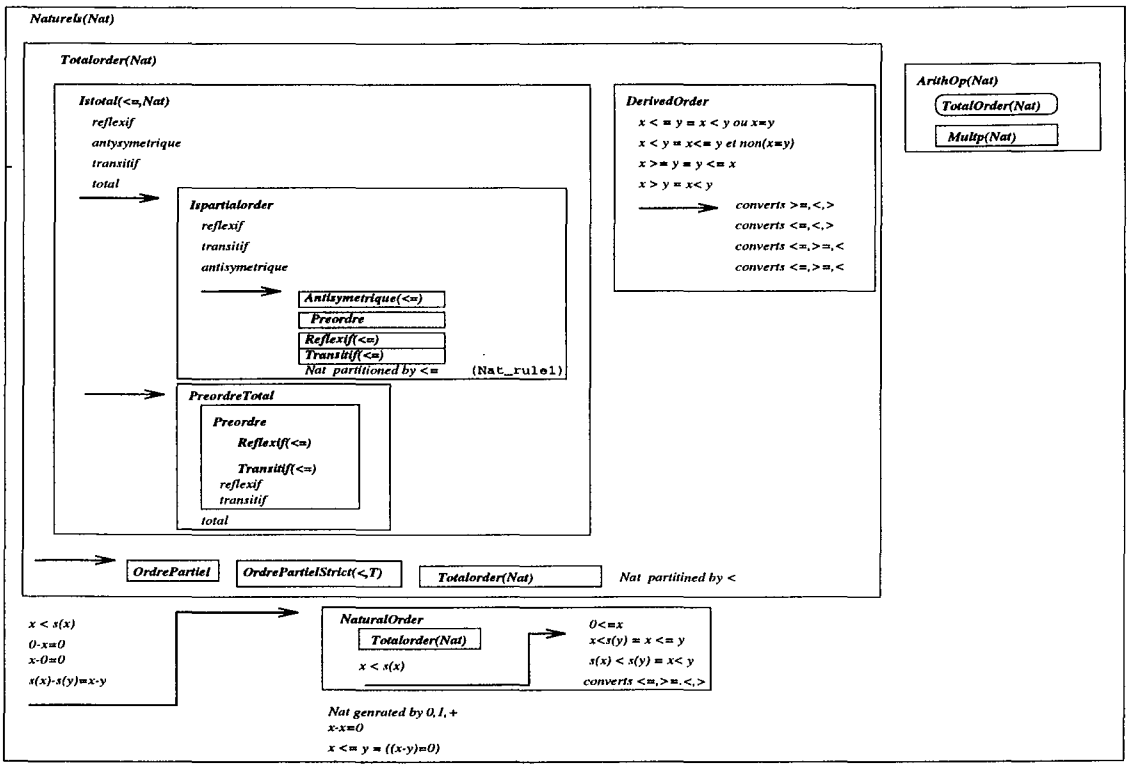



FIG. 3.1 – *La spécification des entiers naturels*

Nous devons montrer que ces axiomes définissent <= comme un ordre partiel (clause `implies` schématisé par la flèche dans la figure 3.1). Puis, nous devons montrer qu'un ordre partiel est aussi anti-symétrique, que c'est un pré-ordre, réflexif et transitif. Toutes ces obligations de preuves sont *redondantes*, trivialement vraies en utilisant la définition.

Une propriété sur l'ordre partiel à prouver est une règle `partitioned by`, c'est à dire une règle de déduction (cf. paragraphe 1.3.3) :

```
set name Nat_rule1
when \A n2 (n <=  n2 <=> n1 <=  n2),
     \A n2 (n2 <=  n <=> n2 <=  n1)
yield n = n1
```

dont la preuve comprend deux preuves par **induction** (*double induction*):

**LpPreuve 3.1**

```
prove Sort Nat partitioned by <=
    resume by induction on n using Nat.3
      resume by induction on n1 using Nat.3    %Cas de base
        resume by contradiction
          set name lemme
          crit-p *hyp with Nat*
          crit-p lemme* with Nat*
      resume by induction on n1 using Nat.3    %Pas d'induction
        resume by contradiction
          set name lemme
```

```
          crit-p *hyp with Nat*
          crit-p lemme* with Nat*
        resume by =>-m
          crit-p *hyp with Nat*
          crit-p Nat* with Nat*
qed
```

La preuve débute par une induction sur l'une des variables libres. LP génère le cas de base, introduit une constante d'induction `nc`, une hypothèse d'induction et le but correspondant à l'étape d'induction :

```
Creating subgoals for proof by structural induction on 'n'
Basis subgoal:
  Subgoal 1: \A n2 (0 <= n2 <=> n1 <= n2)
              /\ \A n2 (n2 <= 0 <=> n2 <= n1)
              => 0 = n1
Induction constant: nc
Induction hypothesis:
  natInductHyp.1:
     \A n2 (nc <= n2 <=> n1 <= n2)
       /\ \A n2 (n2 <= nc <=> n2 <= n1)
      => nc = n1
Induction subgoal:
  Subgoal 2: \A n2 (s(nc) <= n2 <=> n1 <= n2)
              /\ \A n2 (n2 <= s(nc) <=> n2 <= n1)
```

▶ Cas de base : nous commençons par montrer le cas de base, par induction sur la deuxième variable, en l'occurrence `n1` :

```
Creating subgoals for proof by structural induction on 'n1'
Basis subgoal:
  Subgoal 1: \A n2 (0 <= n2 <=> 0 <= n2)
             /\ \A n2 (n2 <= 0 <=> n2 <= 0)
              => 0 = 0
Induction constant: n1c
Induction hypothesis:
  natInductHyp.1:
     \A n2 (0 <= n2 <=> n1c <= n2)
      /\ \A n2 (n2 <= 0 <=> n2 <= n1c) => 0 = n1c
Induction subgoal:
  Subgoal 2: \A n2 (0 <= n2 <=> s(n1c) <= n2)
              /\ \A n2 (n2 <= 0 <=> n2 <= s(n1c))
             => 0 = s(n1c)
```

▶▶ Cas de base : *trivial.*
▶▶ Pas d'induction : le sous-but correspondant à l'étape d'induction est normalisé à :

```
~(\A n2 (0 <= n2 <=> s(n1c) <= n2)
    /\ \A n2 (n2 <= 0 <=> n2 <= s(n1c)))
```

Pour prouver ce but, nous procédons par **contradiction**. Nous supposons la négation du but pour essayer de générer une inconsistance, ce qui revient à montrer le but `true=false`. Ayant les hypothèses:

```
NatContraHyp.1.1: s(n1c) <= n2 -> 0 <= n2
NatContraHyp.1.2: n2 <= s(n1c) -> n2 <= 0
```

```
NatInductHyp.1: \A n2 (0 <= n2 <=> n1c <= n2)
                /\ \A n2 (n2 <= 0 <=> n2 <= n1c)
                 => 0 = n1c
                 -> true
```

nous calculons les **paires critiques** entre les hypothèses et les axiomes définissant les naturels, c'est à dire toute règle dont le nom est préfixé par `Nat`, ce qui s'écrit en LP, `Nat*`. Le calcul de paires critiques entre la première hypothèse et l'axiome d'anti-symétrie (`Nat.14`) donne la paire critique non triviale, `lemme.1: 0 = j <=> s(n1c) = j`. Puis nous calculons les paires critiques entre les règles obtenues[18] et les axiomes de la base. La règle `lemme.1` avec `Nat.3.1: 0 = s(n) -> false` nous donne la paire critique `false`.

**Remarque 3.1** *Si nous avions défini les naturels comme étant construits avec 0 et* `suc` *sans l'option* `freely`*, la preuve aurait été légèrement différente. En effet, le sous-but du pas d'induction* `subgoal 2` *n'aurait pas été réécrit en une formule de la forme* `~(f = g)`*, dû à l'absence de l'axiome* `0 = s(nc)`*. Nous aurions utilisé la commande* `resume by =>-m`*, afin d'obtenir le sous-but* `0 = s(n1c)` *et les hypothèses correspondantes:*

```
NatImpliesHyp.1.1: s(n1c) <= n2 -> 0 <= n2
NatImpliesHyp.1.2: n2 <= s(n1c) -> n2 <= 0
NatInductHyp.1: \A n2 (0 <= n2 <=> n1c <= n2)
                /\ \A n2 (n2 <= 0 <=> n2 <= n1c)
                 => 0 = n1c
                 -> true
```

*Puis, nous aurions invoqué un calcul de paires critiques, qui, avec* `Nat.14` *et* `NatImpliesHyp.1.1` *donne la paire critique:*

```
Nat.17: 0 = j <=> s(n1c) = j
```

*qui permet de réduire le but à* `true`.

□

▶ Pas d'induction : nous avons donc prouvé le cas de base de la preuve par induction sur la variable `n`. Pour l'étape d'induction, nous procédons de nouveau par induction sur `n1`, nous obtenons le schéma suivant :

```
Creating subgoals for proof by structural induction on 'n1'
Basis subgoal:
  Subgoal 1: \A n2 (s(nc) <= n2 <=> 0 <= n2)
             /\ \A n2 (n2 <=  s(nc) <=> n2 <= 0)
             => s(nc) = 0
Induction constant: n1c
Induction hypothesis:
  NatInductHyp.2:
     \A n2 (s(nc) <= n2 <=> n1c <= n2) /\ \A n2 (n2 <= s(nc) <=> n2 <= n1c)
       => s(nc) = n1c
Induction subgoal:
  Subgoal 2: \A n2 (s(nc) <=  n2 <=> s(n1c) <= n2)
             /\ \A n2 (n2 <= s(nc) <=> n2 <= s(n1c))
             => s(nc) = s(n1c)
```

18. D'où l'utilité de donner un nom aux paires critiques qui seront calculées afin d'accélérer le processus.

▶▶Cas de base : par contradiction.
▶▶Pas d'induction :

```
\A n2 (s(nc) <= n2 <=> s(n1c) <= n2)
  /\ \A n2 (n2 <= s(nc) <=> n2 <= s(n1c))
  => nc = n1c
```

La conjecture ne contient plus de variable libre, nous pouvons alors utiliser la méthode syntaxique (cf. paragraphe 1.4.3) **implique**[19], nous obtenons le but (`nc = n1c`) et les hypothèses :

```
NatImpliesHyp.1.1: s(nc) <= n2 -> s(n1c) <= n2
NatImpliesHyp.1.2: n2 <= s(nc) -> n2 <= s(n1c)
NatInductHyp.1: \A n2 (nc <= n2 <=> n1 <= n2)
                 /\ \A n2 (n2 <= nc <=> n2 <=  n1)
                 => nc = n1
                 -> true
NatInductHyp.2: \A n2 (s(n1c) <= n2 <=> n1c <= n2)
                 /\ \A n2 (n2 <= s(n1c) <=> n2 <= n1c)
                 => s(nc) = n1c
                 -> true
```

En calculant les paires critiques entre ces hypothèses et les axiomes (la première hypothèse et l'anti-symétrie), nous obtenons la paire critique :

```
lemme.1: s(n1c) = j <=> s(nc) = j
```

avec laquelle nous calculons les paires critiques avec la base (`Nat.3.2: s(n) = s(n1) -> n = n1`) pour obtenir les paires critiques :

```
lemme.3: n = nc <=> n = n1c
lemme.4: nc = n1 <=> n1 = n1c
```

**Remarque 3.2** *Nous aurions pu remplacer les paires de commandes de calcul de paires critiques par la commande* `complete`. *En effet, le résultat est identique, mais nous préférons la première parce que la* completion *n'est efficace que sur une base de règles de petite taille. Sinon, la complétion est un processus très lent et le nombre de paires critiques générées peut être élevé avant d'obtenir la paire qui permet de réécrire le but à* `true`.

□

Nous définissons maintenant les ordres dérivés de `<=` :

```
assert
 i <= j  <=> (i < j \/ i=j);
 i < j   <=> (i <= j /\ ~(i=j));
 i >= j  <=> j <= i;
 i > j   <=> j < i ;
..
```

Il faut montrer que cet ordre total (`<=`) implique un certain nombre de propriétés (figure 3.1):

• `>=` est un ordre total sur `Nat`. Nous montrons que `>=` est un *observateur* de la sorte `Nat`, c'est à dire une règle identique à `Nat_rule1` avec `>=`. La preuve correspondante est identique à la

19. En effet, comme nous l'avons expliqué au chapitre 1, les méthodes syntaxiques doit être utilisée avec précaution car elles remplacent les variables libres par des constantes sur lesquelles, bien sûr, on ne peut plus appliquer l'induction.

preuve3.1, car >= est défini en fonction de <=. Il suffit alors de "rejouer" la preuve.

• < et > sont des ordres totaux stricts : irréflexifs, transitifs, asymétriques et totaux. Ce sont des propriétés que nous pouvons déduire de notre spécification (c'est à dire des axiomes et théorèmes de la base). Par conséquent, les preuves correspondantes se ramènent à des calculs de paires critiques :

**LpPreuve 3.2**

```
prove ~(i < i)                                %irreflexif
  resume by contradiction
    crit-p *hyp with Nat*
qed

prove ((i < j) /\ (j < n)) => (i < n)         %transitif
    resume by =>-m
      crit-p *hyp with Nat*
      crit-p Nat* with Nat*
qed

prove (i < j) => ~(j < i) by =>-m             %asymetrique
    resume by contradiction
       crit-p *hyp with Nat*
qed

prove (i < j) \/ (j < i) \/ (i=j)             %total
  [] conjecture
qed
```

**Complétude relative** Un exemple de propriété LSL, `converts`, pour spécifier la complétude relative est la suivante :

```
converts >=,<,>
```

Cette propriété exprime le fait que les opérateurs cités sont bien définis, relativement à <=, c'est à dire que si on fixe l'interprétation de tous les autres opérateurs (en particulier <=), il y une interprétation unique de ces trois opérateurs. Pour montrer une telle propriété avec LP, on définit trois autres opérateurs, >>=, >> et << et on prouve les obligations de preuve correspondantes :

```
declare operators
__<<__,__>>=__,__>>__: Nat,Nat -> Bool

assert
Nat partitioned by  >=
assert
 i <= j  <=> (i << j \/ i=j);
 i << j  <=> (i <=  j /\ ~(i=j));
 i >>= j <=> j <= i;
 i >> j  <=> j << i ;
..
prove i < j = i ≪ j
  [] conjecture
prove i < j = i ≪ j
  [] conjecture
prove i > j = i ≫ j
  [] conjecture
prove i >= j = i ≫= j
  [] conjecture
```

• Une dernière propriété à prouver sur < est la règle équivalente à `Nat_rule1` pour cet opérateur, dont la preuve est identique à la preuve 3.1. Plus précisément, ces preuves (pour >= et <) ne requièrent aucune **interaction**. Il nous suffit de "rejouer" la preuve 3.1, car les noms utilisés (`Nat*`, `*hyp`) sont génériques et le nom `lemme` est local à la preuve.

**Remarque 3.3** *Ce traitement simule la parametrisation d'une preuve* LP. *En effet, la preuve 3.1 peut être paramétrée avec une variable* `Ord` *qui sera instanciée par* <=, >= *et* <, *pour déduire les trois preuves correspondantes. En* LP, *il nous suffit de rejouer les preuves en remplaçant uniquement l'ordre.*

Pour achever la spécification des naturels, `Naturels(Nat)`, nous définissons la soustraction et nous introduisons les axiomes suivants :

```
assert
 i < s(i);
 0-i=0;
 i-0=i;
s(i)-s(j) = i-j
..
```

A présent, conformément à la spécification LSL (figure 3.1), nous devons montrer les obligations de preuves associées à la spécification des naturels, exprimées par la clause `implies`. Ce sont des propriétés qui doivent être induites par l'ensemble de faits présents dans la base `Naturels(Nat)`, à savoir que l'on a bien définit un ordre sur les entiers naturels (la théorie `naturels(Nat)` induit `naturalOrder`), et que les naturels sont aussi générés par `0,1,+`, et enfin deux propriétés de la soustraction.

Nous commençons par montrer que la théorie `Naturels(Nat)` induit `NaturelOrder`. Celle-ci inclut la théorie `TotalOrder(Nat)` et introduit un axiome `i < s(i)`. Or, nous avons déjà montré `TotalOrder(Nat)` et l'axiome introduit est redondant.
A présent, il faut montrer les propriétés induites par la théorie `NaturelOrder`:

```
0 <= i
(i <  s(j)) <=>  (i <=  j)
(s(i) <  s(j)) <=>  (i <  j)
```

Étudions la preuve de la troisième propriété, par double induction sur `i` et sur `j`, en utilisant la règle d'induction `Nat.3`. L'induction sur la variable `i` produit le schéma suivant :

```
Creating subgoals for proof by structural induction on 'i'
Basis subgoal:
  Subgoal 1: s(0) <  s(j) <=>  0 <  j
Induction constant: ic
Induction hypothesis:
  NatInductHyp.1: s(ic) <  s(j) <=>  ic <  j
Induction subgoal:
  Subgoal 2: s(s(ic)) <  s(j) <=>  s(ic) <  j
```

▶ Cas de base : le sous-but est normalisé à

```
s(0) <  j \/ s(0) = j <=>  0 <  j
```

Nous procédons par induction sur `j` :

▶▶Cas de base : le sous-but est normalisé à

```
~(s(0) <  0)
```

prouvé par contradiction. L'hypothèse (`s(0) < 0`) est alors ajoutée à la base de faits pour essayer de déduire la formule `false`. Celle ci est déduite par calcul de paires critiques entre les hypothèses (il y en a qu'une à ce niveau) et la base de règles. Plus précisément, la règle utilisée est :

```
Nat.24: i <  j => ~(j <  i) ->  true
```

avec `i = s(0)` et `j = 0`. Le terme `(0 < s(0))` est réécrit en `~(0 < 0 \/ 0 = 0)` puis en `false`.
□

▶▶ Pas d'induction :

```
s(0) < s(jc) \/ s(0) = s(jc) <=> 0 < s(jc)
```

est automatiquement résolu par LP en utilisant l'hypothèse d'induction

```
s(0) < jc \/ s(0) = jc <=> 0 < jc
```

□

▶ Pas d'induction (sur `i`) :

```
s(s(ic)) <  j \/ s(s(ic)) = j <=>  s(ic) <  j
```

A nouveau, nous procédons par induction sur `j`, ce qui nous donne le schéma suivant :

```
Creating subgoals for proof by structural induction on 'j'
Basis subgoal:
  Subgoal 1: s(s(ic)) <  0 \/ s(s(ic)) = 0 <=>  s(ic) <  0
Induction constant: jc
Induction hypothesis:
  NatInductHyp.2: s(s(ic)) <  jc \/ s(s(ic)) = jc <=>  s(ic) <  jc
Induction subgoal:
  Subgoal 2: s(s(ic)) <  s(jc) \/ s(s(ic)) = s(jc) <=>  s(ic) <  s(jc)
```

▶▶ Cas de base : il est prouvé par la méthode `<=>` , qui se ramène à montrer deux implications.

```
Creating subgoals for proof of <=>
=> hypothesis:
  NatImpliesHyp.1: s(ic) <  0
=> subgoal:
  s(s(ic)) <  0
<=  hypothesis:
  NatImpliesHyp.2: s(s(ic)) <  0
<=  subgoal:
  s(ic) <  0
```

Chaque cas est résolu par calcul de paires critiques entre l'antécédent de l'implication (hypothèse de l'implication) et la base, en particulier avec la règle :

```
Nat.24: i <  j => ~(j <  i) -> true
```

qui génère une inconsistance. En effet, l'antécédent de chaque implication se réduit à `false` et par conséquent, le but correspondant à `true`.
□

▶▶ Pas d'induction : il est prouvé de façon automatique où LP utilise l'hypothèse d'induction,

```
NatInductHyp.1.
```
□

**Remarque 3.4** *Cette preuve nécessite une interaction minime avec le démonstrateur. En effet, la stratégie de preuve que nous utilisons est la suivante : la présence de deux variable libres requiert une double induction, puis pour chaque cas de base :*

- *Si c'est une négation alors on poursuit par contradiction puis par calcul de paire critique pour déduire une inconsistance.*
- *Sinon, si la conjecture contient un opérateur logique alors utiliser la méthode syntaxique correspondante.*
- *Sinon, si la conjecture n'est pas sous forme de négation et ne contient pas d'opérateur logique, ni de variable libre alors demander un calcul de paires critiques entre les hypothèses et la base courante.*

*En effet, le script de la preuve décrite précédemment est le suivant :*

**LpPreuve 3.3**
```
prove (s(i) < s(j)) <=> (i < j) by induction on i using Nat.3
   resume by induction on j using Nat.3
      resume by contradiction
        crit-p *hyp with Nat*
    resume by induction on j using Nat.3
      resume by <=>
        crit-p *hyp with Nat*
        crit-p *hyp with Nat*
qed
```

Outre les trois propriétés vues plus haut, la théorie `NaturelOrder` doit impliquer les propriétés de complétude relative des opérateurs <=, >=, <, >. Ces propriétés sont triviales et le principe de preuve utilisé est similaire à celui décrit précédemment (cf. p.78).

### 3.2.3 Preuve d'un schéma d'induction

Outre la théorie `NaturelOrder`, nous devons montrer que la théorie `Naturels(Nat)` induit un deuxième schéma d'induction sur les entiers naturels, à savoir qu'ils sont construits avec `0`, `1` et +. la preuve correspondante est par induction en utilisant le premier schéma d'induction :

**LpPreuve 3.4**
```
prove Sort Nat generated by 0,1,+
    resume by induction using Nat.3
      set name lemma
      crit-p *GenHyp with *GenHyp
      crit-p *InductHyp with lemma
qed
```

En réponse à la commande `prove ..`, LP introduit un opérateur `isgenerated: Nat -> Bool`, ajoute trois hypothèses (une par constructeur déclaré dans la règle à prouver) :

```
NatGenHyp.1: isGenerated(0)
NatGenHyp.2: isGenerated(1)
NatGenHyp.3: isGenerated(n) /\ isGenerated(n1) => isGenerated(n + n1)
```

puis essaye de prouver le but `isgenerated(n)`. A ce niveau, LP a besoin de l'aide de l'utilisateur pour poursuivre la preuve. Nous procédons alors par induction en utilisant la règle d'induction `Nat.3`:

```
Creating subgoals for proof by structural induction on 'n'
Basis subgoal:
  Subgoal 1: isGenerated(0)
Induction constant: nc
Induction hypothesis:
  Nat-InductHyp.1: isGenerated(nc)
Induction subgoal:
  Subgoal 2: isGenerated(s(nc))
```

▶ Cas de base: trivial, éliminé à l'aide de l'hypothèse `NatGenHyp.1`.
▶ Pas d'induction: nous commençons à calculer les paires critiques entre les hypothèses de "génération", nous obtenons la paire critique non triviale:

```
  lemma.1: isGenerated(n) => isGenerated(s(n))
```

puis nous calculons les paires critiques entre cette règle et l'hypothèse d'induction. Nous obtenons:

```
 lemma.3: isGenerated(s(nc))
```

Ce qui est exactement le but cherché.
□.

**Remarque 3.5** *Cette preuve est importante pour la réutilisation car les commandes sont génériques. Les noms des hypothèses sont générés par* LP, *qui utilise le nom du contexte courant comme préfixe à* `GenHyp`, *ce qui nous permet de définir une structure générale pour la preuve des règles d'induction. En effet, le nombre d'hypothèses générées est égal au nombre de constructeurs de la règle à prouver.*

Enfin, la spécification `Naturels(Nat)` doit satisfaire deux propriétés de la soustraction:

```
 i-i=0
 ((i-j)=0) <=> (i <= j)
```

La preuve de la première propriété se fait par induction sur `i` en utilisant le premier schéma d'induction `Nat.3`, et celle de la seconde utilise une double induction sur `i` et `j` avec `Nat.3`, une preuve par contradiction et un calcul de paires critiques (pour générer `false`).

### 3.2.4 Preuve d'un existentiel

Soit la propriété "pour tout entier naturel, il en existe un plus grand". Pour montrer cette propriété, nous utilisons la quantification existentielle, et la méthode de preuve associée, à savoir **la spécialisation** (cf. paragraphe 1.3.2).

**LpPreuve 3.5**

```
prove \A i \E j (i <= j)
  resume by specializing j to s(i)
    < >  specialization subgoal
    [] specialization subgoal
  [] conjecture
qed
```

La simplicité de la preuve découle du fait que pour montrer qu'il existe un "j" vérifiant la propriété, il suffit de "choisir" "j" comme étant le successeur de "i". C'est ce qui est traduit par la commande de "spécialisation". En réponse, LP remplace toutes les occurrences de la variable `j` liées par le quantificateur, par le terme `s(i)` et normalise le but à `true`.

### 3.2.5 Déduction

Le calcul de paires critiques effectué sur la base courante nous permet d'une part d'éliminer les redondances et d'autre part de déduire des propriétés supplémentaires, dont on pourrait avoir besoin dans la suite de la vérification. De plus, ce calcul permet de vérifier la cohérence de la base, bien qu'à cette étape de la construction, tout fait introduit a été prouvé comme étant un théorème.

Avec notre base courante, nous obtenons des propriétés intéressantes telles que :

```
i < j /\ ~(s(i) = j) <=> s(i) < j
```

### 3.2.6 Preuve par cas

Nous nous proposons de montrer une propriété de la multiplication sur les entiers :

```
(~(i=0) /\ ~(j=0)) => ((i*j) >=  i)
```

Cette propriété nécessite une double induction ainsi qu'un traitement **par cas** afin d'utiliser les hypothèses d'induction. En effet, nous commençons par une induction sur `i` dont le cas de base est trivial (antécédent de l'implication se réduit à `false`), puis LP introduit l'hypothèse d'induction :

```
NatInductHyp.1: ~(ic = 0) /\ ~(j = 0)
                => ic <  (ic * j) \/ ic = ic * j ->  true
```

et le but d'induction normalisé :

```
~(j = 0) => ic < ((j * ic) + j)
```

Nous poursuivons alors par induction sur `j` dont le cas de base est trivial. LP introduit l'hypothèse d'induction

```
NatInductHyp.2:~(jc = 0) => ic < ((jc * ic) + jc) -> true
```

afin de montrer le but :

```
ic < ((ic * jc) + ic + jc) \/ ic = (ic * jc) + ic + jc
```

Nous savons, de façon intuitive, que nous devons utiliser la deuxième hypothèse. En effet, si `i < K` alors `i < K + R`. Pour cela, nous montrons le lemme suivant:

```
\A i \A j \A m ((i > j) => ((i + m) > j))
```

dont la preuve utilise une induction sur `i`, puis un calcul de paires critiques pour le cas de base et une induction sur `j` pour l'étape d'induction.

Pour utiliser ce lemme avec le terme `ic <((jc * ic) + jc)`, il faut que celui ci soit un fait dans le système. Nous procédons par cas suivant que `~(jc=0)`, ou `(jc=0)`.

- `~(jc=0)` : en instanciant la variable `i` du lemme par `((jc * ic) + jc)`, `j` par `ic` et `m` par `jc`.

  Ceci est fait implicitement par calcul de paires critiques entre les hypothèses et la base de règles. LP arrête le calcul dès que la paire critique : `ic <((ic * jc) + ic + jc)` est générée.

- `(jc=0)` : le but est réduit à `ic <= ic`, donc à `true`.

□

### 3.2.7 Stratégie de choix

Soit la propriété suivante à montrer :

```
((j < i) => (s(i-j)=(s(i)-j)))
```

La preuve utilise une induction sur les deux variables, sur `j` puis sur `i`. Le cas de base, `(j = 0)`, est prouvé sans l'aide de l'utilisateur et le pas d'induction est prouvé par induction sur `i`, dont seul le cas de base nécessite une interaction avec l'utilisateur.

En effet, le sous-but à montrer est :

```
s(jc) < 0 => s(0 - s(jc)) = 0 - jc
```

avec comme unique hypothèse

```
NatInductHyp.1: jc <  i => s(i - jc) = s(i) - jc -> true
```

En procédant par implication, l'hypothèse `s(jc) < 0` est ajoutée à la base de faits. Nous calculons les paires critiques entre cette hypothèse et la règle

```
Nat.37: i < j /\ j < i -> false
```

Nous obtenons la paire critique :

```
s(jc) <  0 /\ 0  <  s(jc) -> false
```

réduite à `false`. Par conséquent, l'implication a été prouvée en réduisant son antécédent à `false`.

Nous aurions pu résoudre ce but sans utiliser la méthode d'implication mais par calcul de paires critiques entre les hypothèses et la base de faits puis de nouveau entre les paires obtenues et la base. Néanmoins, nous privilégions la première méthode, la **stratégie** étant d'utiliser la méthode la plus immédiate par rapport à la forme de la conjecture. En effet, comme LP n'est pas un démonstrateur automatique mais semi-automatique, cette stratégie augmente le degré d'automatisme en diminuant la recherche de la méthode à utiliser. Ainsi, si nous avons une implication, nous procédons d'abord par implication pour rajouter l'hypothèse (antécédent) à la base de faits puis nous calculons ce que l'on peut "obtenir" avec la base des règles. La contradiction est utilisée lorsque le but à prouver est sous forme d'une négation, ou bien sous forme d'une conjecture ne contenant aucun opérateur logique auquel est associée une méthode LP.

Ceci achève la construction de la base de faits représentant les entiers naturels. Nous n'avons pas décrit toutes les propriétés de la base, mais nous avons détaillé certaines preuves pour illustrer les principales méthodes de preuves disponibles en LP.

## 3.3 Les expressions et leurs preuves

Les preuves sur les expressions sont essentiellement des preuves de lemmes sur les expressions arithmétiques, mettant en jeu les opérateurs $+$, $-$, $*$.

Nous avons définis l'égalité syntaxique entre deux expressions à l'aide de l'opérateur $\backslash=$, que nous avons introduit comme un opérateur commutatif. Les opérateurs $+$, $*$, sont définis comme étant associatifs et commutatifs. Les principaux axiomes sur les expressions sont les suivants :

```
set name Expression
assert
```

```
ex \= ex  = T;
when (ex1 \= ex2) = T yield ex1 = ex2;
when (ex1 \= ex2) = F yield ~(ex1=ex2)
..

set name Arith_expression

%if ex appears in a term with +,-,* then it is an arithmetic expression
assert
(ex + nat_to_exp(0)) = ex;
(ex - ex) = nat_to_exp(0);
(ex - nat_to_exp(0)) = ex;
(ex * nat_to_exp(0)) = nat_to_exp(0);
(ex1+ex2)*ex3 = (ex1*ex3)+(ex2*ex3);
..
```

Nous avons défini les ordres <= et < dont la signature est : `Exp,Exp -> Bexp`, par analogie à leurs correspondants sur les entiers :

```
set name Ord_expression
set immunity on
assert (ex1 <= ex1) = T;
set immunity off

assert
(((ex1 <= ex2)=T) /\  ((ex2 <= ex3)=T)) => ((ex1 <= ex3) = T);
(((ex1 <= ex2)=T) /\  ((ex2 <= ex1)=T)) <=> (ex1=ex2);
((ex1 <= ex2)=T)  <=>  (((ex1 < ex2)=T) \/  (ex1 = ex2));
((ex1 < ex2)=T) => (((ex1 <= ex2)=T) /\ ~(ex1=ex2));
(ex1 >= ex2) = (ex2 <= ex1);
(ex1 > ex2) = (ex2 < ex1) ;
..
```

Par conséquent, la plupart des lemmes prouvés sur les expressions arithmétiques sont similaires à ceux prouvés sur les entiers. Par exemple, la preuve suivante :

```
prove ~((ex < ex)=T)
  resume by contradiction
    <> contradiction subgoal
    crit-p *hyp with Ord_expression*
    [] contradiction subgoal
  [] conjecture
qed
```

est "calquée" sur celle effectuée pour `i ∈ Nat` (preuve 3.2). En effet, les commandes utilisées sont identiques, où le nom `Nat*` représentait la base de règles définissant les entiers, est remplacé par le nom `Ord_expression*` représentant celle des expressions. De même, pour montrer que < est transitif, la preuve, déduite de la preuve 3.2, est la suivante :

```
prove (((ex1 < ex2)=T) /\ ((ex2 < ex3)=T)) => ((ex1 < ex3)=T)
  resume by =>-m
    <> => subgoal
    crit-p *hyp with Ord_exp*
    crit-p Ord_exp* with Ord_exp*
    [] => subgoal
  [] conjecture
qed
```

## 3.4 Les séquences et leurs preuves

Les séquences constituent le type de donnée le plus important de notre formalisation. En effet, ce type abstrait peut avoir différentes implémentations, à savoir les listes, les piles, les files, les chaînes de caractères, etc.

Nous commençons par définir les séquences comme étant des *files* à *double entrées*, `doublefiles(Exp,Seq)` :

```
set name Seq

declare sort Seq
%-------------------- a sequence is a double ended queue
declare operators
emptyseq    :       -> Seq
__-|__      :Exp,Seq-> Seq
__|-__      :Seq,Exp-> Seq
count       :Exp,Seq-> Nat
inseq       :Exp,Seq-> Bexp
head,last   :Seq    -> Exp
tail,initseq:Seq    -> Seq
len         :Seq    -> Nat
isempty     :Seq    -> Bool
..

declare variables
exp1,exp2:Exp
seq1,seq2:Seq
..
%-------------------- Induction Rule
set name seq_ind1
assert   sort Seq generated freely by emptyseq,|-;

set name Seq
assert
count(exp1,emptyseq)     = 0;
count(exp1,exp2 -| seq1) = (count(exp1,seq1)+ (if exp1=exp2 then 1 else 0));
inseq(exp1,seq1)         = bool_to_bexp(count(exp1,seq1) >  0);
exp1 -| emptyseq         = emptyseq |- exp1;
exp1 -| (seq1 |- exp2)   = (exp1 -| seq1) |- exp2 ;
head(exp1 -| seq1)       = exp1;
last(seq1 |- exp1)       = exp1;
tail(exp1 -| seq1)       = seq1;
initseq(seq1 |- exp1)    = seq1;
len(emptyseq)            = 0;
len(seq1 |- exp1)        = (len(seq1)+1);
isempty(seq1)           <=> (seq1  = emptyseq);
..
```

Nous allons montrer que cette spécification induit celle des piles, des files, et des conteneurs, en utilisant une démarche descendante.

### 3.4.1 Les Piles

La spécification `doublefiles(Exp,Seq)` induit la spécification du type abstrait *pile*. C'est à dire, si nous considérons `head` comme étant `top`, `tail` comme étant `pop`, `-|` pour `push` et `len` pour `size`, cette spécification doit vérifier les propriétés des piles :

```
set name seq_ind2
```

```
assert Sort Seq generated by emptyseq,-|

set name Seq
assert
head(e -| seq)= e
tail(e -| seq)= e
len(emptyseq) = 0
len(e -| seq) = len(seq) + 1
..
```

dont les preuves sont immédiates. Par contre, ces axiomes nous fournissent une deuxième règle d'induction `seq_ind2` pour la sorte `Seq`, utilisant le deuxième constructeur `-|`. Par conséquent, une preuve par induction structurelle sur une variable de la sorte `Seq` nécessite un choix de la règle d'induction à utiliser. Comme nous le verrons dans la suite de ce chapitre, la règle d'induction peut être choisie suivant la forme de la conjecture et en particulier suivant les opérateurs sur lesquels porte la propriété à prouver. Suivant que ces opérateurs sont définis en fonction du constructeur `|-` ou bien du constructeur `-|`, nous choisissons la règle d'induction correspondante.

### Les conteneurs

Le type abstrait *pile* induit le type abstrait *conteneur*, dans le sens le plus général, avec `emptyseq`, `-|` pour `insert`, `head`, et `tail`. Les propriétés à vérifier sont les suivantes (les autres propriétés sont redondantes) :

```
Seq partitioned  by isempty,head,tail
~ isempty(seq1) => (count(exp1,head(seq1) -| tail(seq1))=count(exp1,seq1))
~ isempty(seq1) => (count(head(seq1),seq1) > 0)
```

• Le premier axiome est traduit en une règle de déduction :

```
when isempty(s2) <=> isempty(s3),
     head(s2) = head(s3),
     tail(s2) = tail(s3)
yield s2 = s3
```

dont la preuve utilise une double induction structurelle sur les variables `s2` et `s3` :

### LpPreuve 3.6

```
prove Sort Seq partitioned by isempty,head,tail
    resume by induction on s2 using seq_ind2
      resume by induction on s3 using seq_ind2
qed
```

En réponse à la commande `prove..`, LP génère le cas de base, le pas d'induction, et l'hypothèse d'induction :

```
Basis subgoal:
  Subgoal 1: (emptyseq = emptyseq <=> s3 = emptyseq)
              /\ head(emptyseq) = head(s3)
              /\ tail(emptyseq) = tail(s3)
              => emptyseq = s3
Induction constant: s2c
Induction hypothesis:
  seqInductHyp.1:
     (s2c = emptyseq <=> s3 = emptyseq)
       /\ head(s2c) = head(s3)
```

```
      /\ tail(s2c) = tail(s3)
      => s2c = s3
Induction subgoal:
  Subgoal 2: (e -| s2c = emptyseq <=> s3 = emptyseq)
             /\ head(e -| s2c) = head(s3)
             /\ tail(e -| s2c) = tail(s3)
```

Le cas de base est résolu de façon automatique, la première hypothèse de l'implication donne `s3 -> emptyseq` d'où la conséquence. Pour l'étape d'induction, nous procédons par induction sur la deuxième variable `s3`. Ce qui provoque deux nouveaux sous-buts à prouver :

```
Basis subgoal:
  Subgoal 1: s2c = tail(emptyseq) /\ ~(emptyseq = emptyseq)
             => head(emptyseq) -| s2c = emptyseq
Induction constant: s3c
Induction hypothesis:
  seqInductHyp.2:
     s2c = tail(s3c) /\ ~(s3c = emptyseq) => head(s3c) -| s2c = s3c
Induction subgoal:
  Subgoal 2: s2c = tail(e -| s3c) /\ ~(e -| s3c = emptyseq)
             => head(e -| s3c) -| s2c = e -| s3c
```

Le cas de base est de nouveau trivial du fait que l'antécédent de l'implication se réduit à `false`. Le pas d'induction est résolu de façon automatique par LP, par normalisation et en utilisant les axiomes induits par l'option `freely` de la règle d'induction `seq_ind2` :

```
~(e -| emptyseq)
(e -| seq1 = e -| seq2) <=> (seq1=seq2)
```

□

• La deuxième et troisième propriété se montrent par induction sur la variable `seq1` en utilisant la règle d'induction `seq_ind2`.

Le choix de la règle d'induction `seq_ind2` provient de la définition des opérateurs, `head`, `tail` et `count`, définis en fonction de `-|`.

### 3.4.2 Les files

La spécification `doublefiles(Exp,Seq)` induit celle des files, en prenant l'opérateur `-|` pour l'ajout (`append`), `last` pour la tête de la file (`head`), et `initseq` pour la queue de la file (`tail`). Nous avons alors deux des propriétés à prouver :

```
last(exp1 -| seq1)
      = (if seq1 = emptyseq then exp1 else last(seq1))
initseq(exp1 -| seq1)
      = (if seq1 = emptyseq then emptyseq else exp1 -| initseq(seq1))
```

Ces deux propriétés sont prouvées par induction structurelle sur la variable `seq1` en utilisant la règle d'induction `seq_ind1`, définissant l'opérateur `|-` comme générateur, car les opérations `last` et `initseq` sont définies par rapport à ce constructeur.

### 3.4.3 Obligation de preuves

Nous avons vu que la spécification `doublefiles(Exp,Seq)` induit celle des piles et celle des files. Nous allons étudier maintenant les propriétés que la spécification `doublefiles(Exp,Seq)` doit satisfaire :

**(P1) :** `Sort Seq partitioned by len,head,tail`

Cette propriété s'écrit :

```
when len(s2) = len(s3), head(s2) = head(s3), tail(s2) = tail(s3)
  yield s2 = s3
```

La preuve de cette propriété met en évidence l'importance du cas de base, un cas souvent "bâclé" dans les preuves manuelles. En effet, le cas de base nécessite plus d'attention et plus d'interaction avec le démonstrateur que le pas d'induction.

Nous commençons la preuve par une induction structurelle sur l'une des deux variables, par exemple `s2`. La règle d'induction choisie est `seq_ind2`, puisque `head` et `tail` sont définis à l'aide de `-|` et `len` a été "redéfinie" à l'aide de cet opérateur (une des propriété des séquences vues comme des piles). Nous obtenons le cas de base, l'hypothèse d'induction et le pas d'induction suivants :

```
Creating subgoals for proof by structural induction on 's2'
Basis subgoal:
  Subgoal 1: head(emptyseq) = head(s3)
             /\ len(emptyseq) = len(s3)
             /\ tail(emptyseq) = tail(s3)
             => emptyseq = s3
Induction constant: s2c
Induction hypothesis:
  seqInductHyp.1:
     head(s2c) = head(s3) /\ len(s2c) = len(s3) /\ tail(s2c) = tail(s3)
       => s2c = s3
Induction subgoal:
  Subgoal 2: head(e -| s2c) = head(s3)
             /\ len(e -| s2c) = len(s3)
             /\ tail(e -| s2c) = tail(s3)
             => e -| s2c = s3
```

▶ Cas de base : nous procédons par induction sur la variable `s3`, avec la règle d'induction `seq_ind2`. Nous obtenons les sous-buts et hypothèses suivantes :

```
Creating subgoals for proof by structural induction on 's3'
Basis subgoal:
  Subgoal 1: head(emptyseq) = head(emptyseq)
             /\ 0 = len(emptyseq)
             /\ tail(emptyseq) = tail(emptyseq)
             => emptyseq = emptyseq
Induction constant: s3c
Induction hypothesis:
  seqInductHyp.1:
     head(emptyseq) = head(s3c) /\ 0 = len(s3c)
     /\ tail(emptyseq) = tail(s3c)
       => emptyseq = s3c
```

```
Induction subgoal:
  Subgoal 2: head(emptyseq) = head(e -| s3c)
             /\ 0 = len(e -| s3c)
             /\ tail(emptyseq) = tail(e -| s3c)
             => emptyseq = e -| s3c
```

▶▶ Cas de base : trivial.
▶▶ Pas d'induction : il est prouvé par normalisation. LP réécrit le terme (`0 = len(e -| s3c)`) de l'antécédent de l'implication en (`s(len(s3c)) = 0`) qui se réduit à `false`.

**Remarque 3.6** *Le cas de base de l'induction sur* `s2` *aurait pu être prouvé par implication puis par contradiction. En effet, la méthode d'implication aurait introduit trois hypothèses :*

```
len(s3) = 0
head(emptyseq) = head(s3)
tail(emptyseq) = tail(s3)
```

*et le but* `s3 = emptyseq`.

*N'ayant pas l'axiome* `len(s) = 0 => s = emptyseq`, *il aurait fallu procéder par contradiction, pour introduire les deux hypothèses*

```
len(s3) = 0
~(s3 = emptyseq)
```

*afin de déduire* `false` *(par calcul de paires critiques). Par ailleurs, du fait que nous vérifions cette spécification de façon incrémentale, les spécifications des piles et des files sont définies* relativement *à deux axiomes* `head(emptyseq)` *et* `tail(emptyseq)`, *c'est à dire que la valeur de ces deux termes est explicitement non définie.*

▶ Pas d'induction : Il nous faut prouver maintenant la pas d'induction sur la variable `s2`.
L'état de la preuve est :

```
seqInductHyp.1: head(s2c) = head(s3)
                /\ len(s2c) = len(s3)
                /\ tail(s2c) = tail(s3)
                => s2c = s3
                -> true
Current subgoal:
       s(len(s2c)) = len(s3) /\ s2c = tail(s3) => head(s3) -| s2c = s3
```

De nouveau, nous avons besoin de l'induction sur la deuxième variable `s3`, nous obtenons le schéma suivant :

```
Subgoal 1: s(len(s2c)) = len(emptyseq) /\ s2c = tail(emptyseq)
           => head(emptyseq) -| s2c = emptyseq
Induction constant: s3c
Induction hypothesis:
  seqInductHyp.2:
    s(len(s2c)) = len(s3c) /\ s2c = tail(s3c) => head(s3c) -| s2c = s3c
Induction subgoal:
  Subgoal 2: s(len(s2c)) = len(e -| s3c) /\ s2c = tail(e -| s3c)
             => head(e -| s3c) -| s2c = e -| s3c
```

▶▶ Cas de base: trivial, du fait que `s(len(s2c)) = len(emptyseq)` se réduit à `false` et par conséquent l'implication à `true`.
▶▶ Pas d'induction : Il est résolu en utilisant l'implication, dont les antécédents sont `len(s3c) = len(s2c), s2c = s3c` et la conséquence, `e -| s2c = e -| s3c`.
□

**(P2)** : `sort Seq partitioned by len, last,initseq`

Nous utilisons la même structure de preuve que celle de la propriété précédente. En effet, les deux propriétés sont duales, suivant que l'on considère une file à double entrée comme étant une pile ou une file. Par conséquent, seule la règle d'induction diffère puisque les files sont générées à l'aide de `-|`, d'où le choix de `seq_ind1`. La preuve est la suivante, où nous avons remplacé les opérateurs `head` et `tail` par `last` et `initseq` et la règle d'induction `seq_ind2` par `seq_ind1` :

### LpPreuve 3.7

```
prove sort Seq partitioned by len, last,initseq
    resume by induction on s2 using seq_ind1
      resume by induction on s3 using seq_ind1
      resume by induction on s3 using seq_ind1
        resume by =>-m
qed
```

**(P3)**: Une propriété que doit vérifier une file à double entrée est :

```
~ isempty(seq1) => (((head(seq1) -| tail(seq1)) = seq1)
                /\ ((initseq(seq1) |- last(seq1))= seq1))
```

La preuve nécessite une preuve par induction, où la règle utilisée est `seq_ind2` car initialement, `initseq` et `last` sont définies à l'aide de `|-` mais nous avons "prouvé" leurs définitions sur `-|`. Par contre, nous n'aurions pas pu utiliser `seq_ind1` car `head` et `tail` n'ont pas encore été redéfinies avec `|-`.

### LpPreuve 3.8

```
prove
  ~ isempty(seq1) => (((head(seq1) -| tail(seq1)) = seq1)
                   /\ ((initseq(seq1) |- last(seq1))= seq1))
  ..
  resume by induction on seq1 using seq_ind2
    resume by case seq1c = emptyseq
      crit-p *hyp with seq*
qed
```

Les deux premières commandes nous donnent le schéma suivant :

```
Creating subgoals for proof by structural induction on 'seq1'
Basis subgoal:
  Subgoal 1: ~(emptyseq = emptyseq)
              => head(emptyseq) -| tail(emptyseq) = emptyseq
                   /\ initseq(emptyseq) |- last(emptyseq) = emptyseq
Induction constant: seq1c
Induction hypothesis:
  seqInductHyp.1:
     ~(seq1c = emptyseq)
       => head(seq1c) -| tail(seq1c) = seq1c
```

```
          /\ initseq(seq1c) |- last(seq1c) = seq1c
Induction subgoal:
  Subgoal 2: ~(e -| seq1c = emptyseq)
             => head(e -| seq1c) -| tail(e -| seq1c) = e -| seq1c
                  /\ initseq(e -| seq1c) |- last(e -| seq1c) = e -| seq1c
```

▶ Cas de base : trivial, prouvé par normalisation.
▶ Pas d'induction : le sous-but est normalisé (en utilisant les théorèmes prouvés précédemment sur `initseq` et `last`) :

```
    (if seq1c = emptyseq then emptyseq else e -| initseq(seq1c))
 |- (if seq1c = emptyseq then e else last(seq1c))
 = e -| seq1c
```

Quelle méthode doit on alors utiliser pour résoudre ce but ? La conjecture est une formule close, donc l'induction n'est pas possible. De plus, cette formule ne contient aucun connecteur logique auquel est associée une méthode de preuve (*syntaxique*). La méthode pour résoudre les formules *conditionnelles* (cf. paragraphe1.3.3) n'est pas utilisable du fait que `if` n'est pas le symbole principal de la conjecture. De plus, l'hypothèse d'induction, sous forme d'une implication, n'est pas directement utilisable car elle ne peut être appliquée que si la séquence est non vide. Par conséquent, nous allons procéder par cas, suivant la "valeur" de la constante `seic` :
▶▶ Cas (`seq1c = emptyseq`) : le but normalisé à

```
emptyseq |- e = e -| emptyseq
```

est un axiome de la spécification.
□
▶▶ Cas `~(seq1c = emptyseq)` : le but est normalisé à

```
(e -| initseq(seq1c)) |- last(seq1c) = e -| seq1c
```

et l'hypothèse à :

```
seqInductHyp.1.1: initseq(seq1c) |- last(seq1c) = seq1c
seqInductHyp.1.2: head(seq1c) -| tail(seq1c) = seq1c
```

Il suffit alors de "déplacer" les parenthèses de la conjecture afin de pouvoir appliquer la seconde hypothèse. Pour cela, nous invoquons un calcul de paires critiques entre les hypothèses et les faits de la base (`Seq*`). Ce calcul produit la paire critique

```
exp1 -| seq1c = (exp1 -| initseq(seq1c)) |- last(seq1c)
```

obtenue avec l'hypothèse et l'axiome, `Seq.5: exp1 -| (seq1 |- exp2) -> (exp1 -| seq1) |- exp2`.
□

**Remarque 3.7** *Nous aurions pu instancier directement les variables de* `Seq.5` *par les valeurs correspondantes pour terminer la preuve (*`exp1` *par* `e`, `seq1` *par* `seic`, `exp2` *par* `last(seq1c)`*). Mais nous préférons le calcul des paires critiques qui, en général, nous permet plus d'automatisme et nous évite de chercher quel axiome instancier afin de déduire le résultat. Plus précisément, le calcul des paires critiques est à invoquer lorsque la preuve est suspendue et que nous savons que la (ou les) propriété(s) utile(s) pour conclure appartient à la base de faits.*

**(P4)** : la dernière propriété à prouver pour la spécification `Seq` (`doublefile(Exp,Seq)`) concerne l'opérateur `count` qui est défini à l'aide `-|`. Comme nous disposons de deux constructeurs pour la sorte `Seq`, nous complétons notre spécification en définissant `count` sur le deuxième constructeur :

**LpPreuve 3.9**

```
prove
 count(exp1,seq1 |- exp2)
   = (count(exp1,seq1) + (if exp1=exp2 then 1 else 0))
 ..
 resume by induction using seq_ind2
    crit-p seq* with seq*
    crit-p seq* with seq*
qed
```

Cette preuve par induction nécessite deux calculs de paires critiques, un pour le cas de base et un pour le pas d'induction. Le premier calcul produit la paire critique

```
Seq.25: count(exp1, emptyseq |- exp2) = (if exp1 = exp2 then s(0) else 0)
```

obtenue avec les deux règles :

```
Seq.2: count(exp1, exp2 -| seq1)
        -> count(exp1, seq1) + (if exp1 = exp2 then s(0) else 0)
Seq.4: exp1 -| emptyseq -> emptyseq |- exp1
```

Le deuxième produit la paire critique

```
count(exp1, (e -| seq1) |- exp2)
           = count(exp1, seq1 |- exp2) + (if exp1 = e then s(0) else 0)
```

obtenue avec

```
Seq.2: count(exp1, exp2 -| seq1)
        -> count(exp1, seq1) + (if exp1 = exp2 then s(0) else 0)
Seq.5: exp1 -| (seq1 |- exp2) -> (exp1 -| seq1) |- exp2
```

Ceci achève la spécification et la vérification des obligations de preuves des séquences comme étant des `files` à double entrée.

### 3.4.4 Enrichissement

Nous avons besoin à présent d'autres opérateurs sur les séquences, tels que l'extraction d'un *préfixe*, la *concaténation* de deux séquences, ainsi que de l'opération d'accès à un élément donné de la séquence ([_]). Plus précisément, nous allons spécifier le type *chaîne*, dont la spécification inclut celle des *listes* qui elle inclut celle des files à double entrée `doublefiles(Exp,Seq)`.

`List(Exp,Seq)` : A la spécification obtenue des séquences, comme étant des files à double entrée, nous rajoutons les opérateurs usuels pour les listes :

```
declare operators
sing  : Exp     -> Seq
concat: Seq,Seq -> Seq
..
assert
sing(exp1) = emptyseq |- exp1;
concat(seq1,emptyseq) = seq1;
concat(seq1,seq2 |- exp1) = concat(seq1,seq2) |- exp1;
..
```

Nous allons montrer que nous avons introduit une nouvelle règle d'induction, à savoir que la sorte `Seq` est générée par `emptyseq`, `sing` et `concat`.

### LpPreuve 3.10

```
set name seq_ind3
prove sort Seq generated by emptyseq,sing,concat
    resume by induction using seq_ind1
      set name lemma
      crit-p *GenHyp with *GenHyp
      crit-p *InductHyp with lemma
qed
```

Cette preuve est générique, où nous utilisons une structure identique à celle de la preuve 3.4 pour les entiers naturels. Elle consiste à montrer que tout élément de la sorte `Seq` peut être construit en utilisant uniquement les opérateurs `emptyseq,sing,concat`:

En réponse à la commande `prove ..`, LP introduit un l'opérateur `isgenerated: Seq -> Bool`, et les trois hypothèses :

```
isgenerated(emptyseq)
isgenerated(sing(e1))
(isgenerated(s1) /\ isgenerated(s2))
 => isgenerated(concat(s1,s2))
```

puis essaye de prouver le but `isgenerated(s1)`. Nous poursuivons par induction en utilisant la règle d'induction `seq_ind1` puisque `concat` est défini à l'aide de `|-` :

```
Creating subgoals for proof by structural induction on 's2'
Basis subgoal:
  Subgoal 1: isGenerated(emptyseq)
Induction constant: s2c
Induction hypothesis:
  seq_ind3InductHyp.1: isGenerated(s2c)
Induction subgoal:
  Subgoal 2: isGenerated(s2c |- e)
```

▶ Cas de base : trivial, c'est une des hypothèses.
▶ Pas d'induction : par calcul de paires critiques entre les hypothèses. En effet, en remplaçant la variable `s2` de la troisième hypothèse par `emptyseq |- e`, nous obtenons

```
isGenerated(s1) /\ isGenerated(emptyseq |- e)
 => isGenerated(concat(s1, emptyseq |- e))
-> true
```

normalisée à

```
isGenerated(s1)  => isGenerated(s1 |- e) -> true
```

Il suffit alors de remplacer `s1` par la constante d'induction `s2c` pour obtenir le résultat. Ceci est traduit par le calcul de paires critiques entre la règle obtenue (dont le nom est préfixé par `lemma`) et les hypothèses d'induction `*InductHyp`.
□.

Nous obtenons ainsi trois règles d'induction pour la sorte `Seq`:

```
seq_ind1.1: sort Seq generated freely by emptyseq, |-
seq_ind2.1: sort Seq generated freely by emptyseq, -|
seq_ind3.1: sort Seq generated by emptyseq, sing, concat
```

`String(exp,Seq)`: Nous introduisons maintenant les opérateurs sur les séquences considérées comme des chaînes d'éléments de la sorte `Exp`.

```
declare op
__[__]:Seq,Nat-> Exp
prefix:Seq,Nat-> Seq
pos:Exp,Seq-> Nat
..
assert
tail(emptyseq)=emptyseq;
initseq(emptyseq)=emptyseq;
seq1[1] = head(seq1);
seq1[s(n)] = tail(seq1)[n];
prefix(emptyseq,n) = emptyseq;
prefix(seq1,0) = emptyseq;
prefix(exp1 -| seq1,s(n)) =  (exp1 -| prefix(seq1,n));
..
```

A la différence de la spécification `Lsl` correspondante, nous considérons que l'indice de parcours débute à 0 et non à 1.

Nous allons prouver à présent les obligations de preuve induites par la spécification `String(exp,Seq)`. L'une des propriétés à prouver est :

```
(len(s1) = len(s2)) /\ (\A n (s1[n] = s2[n]))
 => (s1 = s2)
```

dont la preuve en LP est :

### LpPreuve 3.11

```
prove Sort Seq partitioned by len,__[__]
    resume by induction on s2 using seq_ind2
      resume by induction on s3 using seq_ind2
      resume by induction on s3 using seq_ind2
        resume by =>-m
          instantiate n by 1 in *hyp
          crit-p *hyp with *hyp
          crit-p *hyp with seq*
qed
```

Nous avons utilisé le même schéma que celui des preuves 3.6 et 3.7. En effet, par analogie, nous savons que nous devons utiliser une double induction, l'une pour le cas de base et l'autre pour le pas d'induction. La règle d'induction est choisie suivant les opérateurs : l'opérateur `[__]` est défini en fonction de `head` et `tail`, définis en fonction de `-|`. Nous choisissons donc la règle `seq_ind2`.

Néanmoins, le dernier sous-but nécessite une instanciation explicite. En effet, nous obtenons les hypothèses suivantes :

```
seqImpliesHyp.1.1: len(s3c) -> len(s2c)
seqImpliesHyp.1.2: (ec -| s2c)[n] -> (e1c -| s3c)[n]
seqInductHyp.1: len(s2c) = len(s3) /\ \A n (s2c[n] = s3[n])
                  => s2c = s3 -> true
seqInductHyp.2: s(len(s2c)) = len(s2c) /\ \A n ((e -| s2c)[n] = s3c[n])
                  => e -| s2c = s3c
                  -> true
```

et le but `ec = e1c /\s2c = s3c`

Pour pouvoir déduire `ec = e1c`, il nous faut instancier `n` par `1` dans la deuxième hypothèse. Puis un calcul de paire critiques entre les hypothèses et les règles du système nous donne les paires critiques:

```
Seq.37: \A n (s2c[n] = s3c[n]) => s2c = s3c  %implieshyp with inductHyp
Seq.39: s3c[n] = s2c[n]
```

où `Seq.39` est obtenue avec la deuxième hypothèse et la règle

```
Seq.29: seq1[s(n)] -> tail(seq1)[n]
```

### 3.4.5 Ordre partiel sur les séquences

Nous introduisons l'opérateur `isprefix`, qui teste si une sous-chaîne est un préfixe d'une chaîne donnée:

```
declare operators
isprefix:Seq,Seq-> Bool
..
assert
isprefix(seq1,seq2) <=>  (seq1 = prefix(seq2,len(seq1)));
..
```

Sous la condition que (>) soit un ordre partiel strict sur les entiers[20], nous vérifions que `isprefix` induit un ordre partiel sur les séquences:

**LpPreuve 3.12**

```
%reflexif
prove isprefix(seq1,seq1) by induction on seq1 using seq_ind2
qed

%transitif
prove isprefix(s1,s2) /\ isprefix(s2,s3) => isprefix(s1,s3)
  resume by induction on s1 using seq_ind2
    resume by induction on s2 using seq_ind2
      resume by induction on s3 using seq_ind2
qed

%antisymetrique
prove isprefix(s1,s2) /\ isprefix(s2,s1) <=> (s1 = s2)
  resume by induction on s1 using seq_ind2
    resume by induction using seq_ind2
qed

%total
prove sort Seq partitioned by isprefix
    resume by induction on s2 using seq_ind2
      resume by induction on s3 using seq_ind2
        resume by contrad
          crit-p *hyp with seq*
      resume by induction on s3 using seq_ind2
        resume by contrad
          crit-p *hyp with seq*
        resume by => -m
```

20. C'est une clause `assumes` : si une spécification est définie en "important" la spécification `Seq`, elle doit vérifier cette propriété

```
          instantiate s4 by ec -| emptyseq in *hyp
          crit-p *hyp with seq*
          crit-p seq* with seq*
qed
```

### Induction imbriquée

L'opérateur `isprefix` doit vérifier la propriété: `isprefix(prefix(seq1,n),seq1)` dont la preuve est :

**LpPreuve 3.13**

```
prove isprefix(prefix(seq1,n),seq1)
  resume by induction on n using Nat.3
    resume by induction using seq_ind2
qed
```

Cette preuve nécessite deux types d'induction. La première est une induction sur la variable `n` de la sorte `Nat`, car `prefix` est défini récursivement sur la longueur de la séquence, puis pour le pas d'induction, nous procédons par induction sur la variable de la sorte `Seq`. La particularité de cette preuve est que l'ordre dans lequel sont invoquées les deux inductions est quelconque. Nous aurions pu commencer par une induction sur la séquence puis une induction sur les entiers. Le résultat est identique.

Enfin, nous prouvons un ensemble de lemmes, afin de compléter la spécification de certains opérateurs, tels que `head` et `tail`. Ces opérateurs étant définis à l'aide de `-|`, les termes `head(seq1 |- e)` et `tail(seq1 |- e)` sont en forme normale, et donc susceptibles de bloquer la preuve.

**LpPreuve 3.14**

```
prove head(emptyseq |- exp1)= exp1                %seq_lemma.1
  crit-p seq* with seq*
qed

prove tail(emptyseq |- exp1) = emptyseq           %seq_lemma.2
  crit-p seq* with seq*
qed

prove
  ~ isempty(seq1)                                  %seq_lemma.3
    => (tail(seq1 |- exp1) = tail(seq1) |- exp1)
..
  resume by induction using seq_ind2
    crit-p seq* with seq*
qed

prove ~ isempty(seq1) => (head(seq1 |- exp1) = head(seq1))  %seq_lemma.4
  resume by induction using seq_ind2
    crit-p seq* with seq*
qed
```

Tous ces lemmes sont prouvés par calcul de paires critiques. C'est à dire que les propriétés à prouver sont des conséquences de notre spécification.

La preuve suivante démontre une fois une fois de plus l'importance du cas de base, généralement sous-estimé dans les preuves complexes. Il faut noter que nous avons utilisé cette propriété comme un lemme (non prouvé) dans une preuve d'une propriété de programme. Puis nous avons

repris ce lemme, afin de le prouver. Lors de cette vérification, nous avons découvert le caractère non trivial du lemme et par conséquent, l'erreur dans la preuve qui l'utilisait.

### LpPreuve 3.15

```
prove
 (s(n)<=  len(seq1))
   => (prefix(seq1,s(n)) = (prefix(seq1,n) |- seq1[s(n)]))
..
  resume by induction on n using Nat.3
    resume by =>-m                          % cas de base
      prove ~(isempty(seq1c))
        resume by contrad
      instantiate seq1 by seq1c in seq_lemma.6
    resume by induction using seq_ind2          % pas d'induction
      resume by =>-m
        crit-p *hyp with *hyp
qed
```

Le cas de base, l'hypothèse et le pas d'induction sont :

```
Creating subgoals for proof by structural induction on 'n'
Basis subgoal:
  Subgoal 1: 0 <  len(seq1)
              => prefix(seq1, s(0)) = prefix(seq1, 0) |- tail(seq1)[0]
Induction constant: nc
Induction hypothesis:
  seq_lemmaInductHyp.1:
     nc <  len(seq1)
      => prefix(seq1, s(nc)) = prefix(seq1, nc) |- tail(seq1)[nc]
Induction subgoal:
  Subgoal 2: s(nc) <  len(seq1)
              => prefix(seq1, s(s(nc)))
                   = prefix(seq1, s(nc)) |- tail(seq1)[s(nc)]
```

▶ Cas de base : il faut montrer que si la longueur de la séquence est strictement positive, alors le préfixe de longueur 1 est réduit à l'élément en tête de la séquence. Pour cela, nous disposons du lemme suivant prouvé en amont :

```
seq_lemma.6: ~(seq1 = emptyseq)
               => prefix(seq1, s(0)) = emptyseq |- head(seq1)
              -> true
```

Ce lemme n'est utilisable que pour des séquences non vides. Il faut donc montrer que `seq1` est non vide. Cela peut paraître trivial puisque la longueur de la séquence est strictement positive. Néanmoins, nous devons procéder par l'absurde, supposer que la séquence est vide et montrer l'inconsistance. C'est une preuve par contradiction.

▶ Pas d'induction : nous procédons par induction sur la variable de type `Seq`, nous obtenons le schéma suivant :

```
Creating subgoals for proof by structural induction on 'seq1'
Basis subgoal:
  Subgoal 1: s(nc) <  len(emptyseq)
              => prefix(emptyseq, s(s(nc)))
                   = prefix(emptyseq, s(nc)) |- tail(tail(emptyseq))[nc]
```

```
Induction constant: seq1c
Induction hypothesis:
  seq_lemmaInductHyp.2:
     s(nc) <  len(seq1c)
       => prefix(seq1c, s(s(nc)))
            = prefix(seq1c, s(nc)) |- tail(tail(seq1c))[nc]
Induction subgoal:
  Subgoal 2: s(nc) <  len(e -| seq1c)
               => prefix(e -| seq1c, s(s(nc)))
                    = prefix(e -| seq1c, s(nc))
                       |- tail(tail(e -| seq1c))[nc]
```

▶▶ Cas de base : trivial avec la définition de `prefix`.
▶▶ Pas d'induction : nous utilisons la méthode *implique*. L'hypothèse (`s(nc) < len(e -| seq1c)`) est rajoutée au contexte et le but est normalisé à :

```
e -| prefix(seq1c, s(nc)) = (e -| prefix(seq1c, nc)) |- tail(seq1c)[nc]
```

que nous résolvons par calcul de paires critiques entre les hypothèses, celle de l'implication et celle de l'induction sur `n` :

```
seq_lemmaImpliesHyp.1: nc <  len(seq1c) -> true
seq_lemmaInductHyp.1: nc <  len(seq1)
                        => prefix(seq1, s(nc))
                             = prefix(seq1, nc) |- tail(seq1)[nc]
                        -> true
seq_lemmaInductHyp.2: s(nc) <  len(seq1c)
                        => prefix(seq1c, s(s(nc)))
                             = prefix(seq1c, s(nc))
                                |- tail(tail(seq1c))[nc]
                        -> true
```

□

A la différence de la preuve 3.13, l'ordre de l'induction est important. Si nous commençons par une induction sur la variable de type `Seq`, nous obtenons le pas d'induction suivant :

```
n <  len(seq1c) \/  n = len(seq1c)
  => e -| prefix(seq1c, n) = prefix(e -| seq1c, n) |- seq1c[n]
```

puis si nous poursuivons par induction sur `n`, le cas de base est alors :

```
0 <  len(seq1c) \/  0 = len(seq1c)
  => e -| prefix(seq1c, 0) = prefix(e -| seq1c, 0) |- seq1c[0]
```

qui se réécrit en: `e = seq1c[0]`, ce qu'on ne peut pas résoudre du fait que `seq1[0]` n'est pas défini.

### 3.4.6 Preuve par induction de profondeur p

Soit la propriété, `seq_lemma.7` :

```
((1 <= n) /\ (n <= s(len(seq1))))
  => (if (s(len(seq1))=n) then (((seq1 |- exp1)[n])= exp1)
                          else (((seq1 |- exp1)[n])= seq1[n]))
```

Cette propriété ne peut être prouvée par induction sur n, car le cas de base $n = 0$ est trivial (l'antécédent de l'implication est faux pour $n = 0$) mais non significatif. Or la sémantique de "cas de base" est généralement une instance de la propriété pour la plus petite valeur, en l'occurrence 1 pour la propriété à prouver.
De plus, le pas d'induction serait :

```
nc = len(seq1) \/ nc <  len(seq1)
  => (if nc = len(seq1)
        then tail(seq1 |- exp1)[nc] = exp1
        else tail(seq1 |- exp1)[nc] = tail(seq1)[nc])
```

et si nous poursuivons par induction sur la variable `seq1`, le sous-but de base, correspondant à `seq1 = emptyseq`, serait de la forme :

```
nc = 0 => (nc = 0 => emptyseq[nc] = exp1)
```

qui se ramène à montrer que `emptyseq[0] = exp1`. Or `emptyseq[m]` est une valeur délibérément indéfinie.

L'induction sur n de profondeur 2 (chapitre 1, p.21) nous permet de prouver le cas de base intéressant (c'est à dire lorsque n vaut 1) sachant qu'une séquence est considérée comme un tableau dont les élément sont T[1]... T[len(T)].

L'induction de profondeur 2 génère deux cas de base et deux hypothèses d'induction :

```
Creating subgoals for proof by depth 2 structural induction on 'n'
Basis subgoals:
  Subgoal 1:  0 <= s(len(seq1))
              => (if s(len(seq1)) = 0
                    then (seq1 |- exp1)[0] = exp1
                    else (seq1 |- exp1)[0] = seq1[0])
  Subgoal 2: 1 <= s(0) /\ s(0) <=  s(len(seq1))
              => (if s(len(seq1)) = s(0)
                    then (seq1 |- exp1)[s(0)] = exp1
                    else (seq1 |- exp1)[s(0)] = seq1[s(0)])
Induction constant: nc
Induction hypothèses:
  seq_lemmaInductHyp.1:
     1 <= nc /\ (nc <= s(len(seq1)))
       => (if s(len(seq1)) = nc
             then (seq1 |- exp1)[nc] = exp1
             else (seq1 |- exp1)[nc] = seq1[nc])
  seq_lemmaInductHyp.2:
     1 <=  s(nc) /\ (s(nc) <= s(len(seq1)))
       => (if s(len(seq1)) = s(nc)
             then (seq1 |- exp1)[s(nc)] = exp1
             else (seq1 |- exp1)[s(nc)] = seq1[s(nc)])
Induction subgoal:
  Subgoal 3: 1 <=  s(s(nc))
              /\ (s(s(nc)) <= s(len(seq1)))
              => (if s(len(seq1)) = s(s(nc))
                    then (seq1 |- exp1)[s(s(nc))] = exp1
                    else (seq1 |- exp1)[s(s(nc))] = seq1[s(s(nc))])
```

Le premier cas de base étant trivial et non significatif, nous nous intéressons au second, `subgoal 2`, normalisé à :

```
if len(seq1) = 0
   then head(seq1 |- exp1) = exp1
   else head(seq1 |- exp1) = head(seq1)
```

La conjecture étant sous la forme d'une conditionnelle, nous procédons avec la méthode `if-method` (cf. chapitre 1, paragraphe 1.4.3). Nous obtenons le schéma suivant :

```
Creating subgoals for proof of 'if'
New constant: seq1c
Case 'true' hypothesis:
  seq_lemmaIfHyp.1: len(seq1c) = 0 <=> true
Case 'true' subgoal:
  head(seq1c |- exp1) = exp1 <=> true
Case 'false' hypothesis:
  seq_lemmaIfHyp.2: len(seq1c) = 0 <=> false
Case 'false' subgoal:
  head(seq1c |- exp1) = head(seq1c) <=> true
```

▶▶Le premier sous-but, correspondant au cas où la longueur de la séquence est nulle, est résolu par LP en utilisant la règle de déduction :

```
seq_lemma.1: when len(seq1) = 0 yield seq1 = emptyseq
```

et le lemme `seq_lemma.4` (cf. p.97).
▶▶ Le deuxième sous-but peut être résolu avec un des lemmes précédemment prouvés, `seq_lemma.4`, mais il nous faut l'information que la séquence est non vide, puisque sa taille est non nulle. La preuve n'est pas immédiate car la règle de déduction `seq_lemma.4` n'est utilisable que lorsque le fait `len(seq) = 0` est vrai dans la base de faits. La solution consiste à utiliser un calcul de paires critiques avec la base `Nat` afin de montrer que `0 < len(seq1c)`, puis à montrer par contradiction le lemme `0 < len(seq1c) => ~(seq1c = emptyseq)`.
Une fois obtenue l'information `~(seq1c = emptyseq)`, nous pouvons alors appliquer le lemme `seq_lemma.4`. Ce qui achève la preuve du cas de base de l'induction.
□

▶ Le but correspondant au pas d'induction est normalisé à :

```
nc <  len(seq1)
  => (if s(nc) = len(seq1)
         then tail(tail(seq1 |- exp1))[nc] = exp1
         else tail(tail(seq1 |- exp1))[nc] = tail(tail(seq1))[nc])
```

La preuve est alors par induction sur `seq1` et utilise les lemmes `seq_lemma.2` et `seq_lemma.3` (cf. page97) avec `e -|seq1c` pour la variable `seq1`.

### 3.4.7 Quantificateur existentiel

Une dernière propriété intéressante à prouver sur les séquences est que pour toute séquence `s` et pour toute élément `e`, si `e` appartient à `s` alors il existe un indice `i` tel que `s[i] = e`. Cette propriété comprend un quantificateur existentiel et sa preuve n'est pas immédiate.

**LpPreuve 3.16**

```
prove
  inseq(exp1,seq1)=T
    => (\E i ((1<= i) /\ (i <= len(seq1)) /\ (seq1[i] = exp1)))
  ..
  resume by induction using seq_ind1
    resume by =>-m
      resume by case exp1c = ec
        resume by specializing i to s(len(seq1c))
          instantiate
            seq1 by seq1c,n by s(len(seq1c)),exp1 by ec in seq_lemma*
          ..
        instantiate exp1 by exp1c in *hyp
        dec op Sk_i:-> Nat
        fix i as Sk_i in *InductHyp.1.1!
        resume by spec i to Sk_i
          instantiate n by Sk_i,seq1 by seq1c,exp1 by ec in seq_lemma.8
          prove ~(s(len(seq1c)) = Sk_i) by contrad
              crit-p *hyp with nat*
qed
```

La preuve débute par une induction sur `seq1` :

```
Creating subgoals for proof by structural induction on 'seq1'
Basis subgoal:
  Subgoal 1: bool_to_bexp(0 < count(exp1, emptyseq)) = T
              => \E i (0 < i
                       /\ emptyseq[i] = exp1
                       /\ (i <= len(emptyseq)))
Induction constant: seq1c
Induction hypothesis:
  seq_lemmaInductHyp.1:
     bool_to_bexp(0 < count(exp1, seq1c)) = T
       => \E i (0 < i /\ seq1c[i] = exp1 /\ (i <= len(seq1c)))
Induction subgoal:
  Subgoal 2: bool_to_bexp(0 < count(exp1, seq1c |- e)) = T
              => \E i (0 < i
                       /\ (seq1c |- e)[i] = exp1
                       /\ (i <= len(seq1c |- e)))
```

- ▶ Cas de base : trivial puisque la séquence vide ne contient aucun élément.
- ▶ Pas d'induction :

```
  bool_to_bexp(0 < (count(exp1,seq1c)+(if exp1=e then s(0) else 0)))= T
    => \E i (0 < i
             /\ (seq1c |- e)[i] = exp1
             /\ (i <= len(seq1c) \/ i = s(len(seq1c))))
```

Le but est sous forme d'une implication et aucune induction n'est applicable sur les variables libres (`exp` et `e`). Par conséquent, la seule méthode applicable est l'implication, dont l'hypothèse est :

```
seq_lemmaImpliesHyp.1.1: 0 < (count(exp1c, seq1c)
                                    + (if exp1c = ec then s(0) else 0))
                         -> true
```

et le but:

```
\E i (0 < i
      /\ (seq1c |- ec)[i] = exp1c
      /\ (i <= len(seq1c) \/ i = s(len(seq1c))))
```

Pour utiliser l'hypothèse, nous procédons par cas :

▶▶ Cas (`exp1c = ec`):
Il suffit alors de choisir `i` tel que `(seq1c |- ec)[i] = ec`, en l'occurrence `len(seq1c) + 1`. En LP, cela revient à *spécialiser* dans la conjecture, la variable quantifiée par le terme `s(len(seiqc))`. Nous obtenons le but :

```
0 < s(len(seq1c))
  /\ (seq1c |- ec)[s(len(seq1c))] = ec
  /\ (s(len(seq1c)) <= len(seq1c)
       \/ s(len(seq1c)) = s(len(seq1c)))
```

normalisé à

```
tail(seq1c |- ec)[len(seq1c)] = ec
```

Le lemme `seq_lemma.7`, prouvé précédemment, nous permet de déduire le résultat à condition d'instancier correctement les variables, `n` par `s(len(seiqc))`, `exp1` par `ec` et `seq1` par `seq1c`.
□

▶▶ Cas `~(exp1c = ec)`:
Cette hypothèse simplifie l'hypothèse `seq_lemmaImpliesHyp.1.1`, et nous obtenons l'ensemble d'hypothèses :

```
seq_lemmaCaseHyp.1.2   : exp1c = ec -> false
seq_lemmaImpliesHyp.1.1: 0 < count(exp1c, seq1c) -> true
seq_lemmaInductHyp.1   : bool_to_bexp(0 < count(exp1, seq1c)) = T
                          => \E i (0 < i
                                   /\ seq1c[i] = exp1
                                   /\ (i <= len(seq1c)))
                          -> true
```

En instanciant dans l'hypothèse d'induction `exp1` par `exp1c`, nous obtenons l'hypothèse :

```
seq_lemmaInductHyp.1.1:
      \E i (0 < i /\ (i <=  len(seq1c))
            /\ seq1c[i] = exp1c)
      -> true
```

alors que le but à prouver est:

```
\E i (0 < i /\ (i <=  s(len(seq1c)))
      /\ (seq1c |- ec)[i] = exp1c)
```

C'est un cas plus complexe que le premier pour lequel nous n'avons pas utilisé l'hypothèse d'induction. Dans le cas présent, nous avons besoin de l'hypothèse d'induction, contenant un quantificateur existentiel.

Le problème est que nous ne connaissons pas la valeur à attribuer à la variable `i`. L'idée consiste à faire "un changement de variable", afin de ramener le but à une forme similaire à celle de l'hypothèse. Nous choisissons une constante entière que nous appelons `Sk_i` (le $k$ pour faire le

changement de variable). Puis nous remplaçons (`fix`) dans l'hypothèse la variable quantifiée par ce terme, ceci revient à éliminer le quantificateur existentiel de l'hypothèse (cf. chapitre 1, paragraphe 1.3.2). Ce qui nous donne trois faits :

```
seq_lemma.22.1: 0 < Sk_i -> true
seq_lemma.22.2: seq1c[Sk_i] -> exp1c
seq_lemma.22.3: Sk_i <= len(seq1c) -> true
```

Comme nous savons que "il existe i tel que ", nous pouvons alors affirmer "soit `Sk_i` tel que". Il suffit de spécialiser (ou de choisir) dans la conjecture la variable quantifiée `i` à `Sk_i`. Le conjecture est alors *spécialisée* à

```
(seq1c |- ec)[Sk_i] = exp1c
```

Pour conclure, il nous faut utiliser le lemme `seq_lemma.7` :

```
((1 <= n) /\ (n <= s(len(seq1))))
  => (if (s(len(seq1))=n) then (((seq1 |- exp1)[n])= exp1)
                          else (((seq1 |- exp1)[n])= seq1[n]))
```

Pour cela, il faut montrer que `Sk_i` est strictement inférieur à `s(len(seq1c))`, ce qui est trivial d'après l'hypothèses `seq_lemma.22.3`. Il suffit alors de calculer les paires critiques entre cette hypothèse et la base des entiers. Le but est alors réécrit en `(seq1c +- ec)[Sk_i] = seq1c[Sk_i] = exp1c`, normalisé à `true` avec l'hypothèse `seq_lemma.22.2`.
□

Ceci achève la spécification et la vérification des séquences. La structure obtenue correspond à la figure 3.2, où la spécification des séquences inclut celle des chaînes, incluant celle des files à double entrée. Or une file à double entrée n'est autre qu'une pile, ou bien une file suivant le sens de l'ajout. De plus, si on ne tient pas compte du sens d'insertion, une pile ou une file représente un conteneur dans le sens le plus général du terme.

## 3.5 Les opérateurs temporels et leurs preuves

Le système de preuve d'UNITY est composé des règles d'inférence dérivées de la définition des principaux prédicats temporels *unless*, *ensures* et *leads_ to*. Dans les preuves de programmes, les règles dérivées jouent le même rôle que les lemmes dans les preuves mathématiques. Nous pouvons ainsi les introduire comme axiomes dans la preuve ou bien les utiliser comme des théorèmes. Dans ce dernier cas, elles doivent être prouvées avant leur utilisation.

La plupart des règles d'inférence sont formalisées comme des implications, et non comme des règles de déduction. Les preuves utilisent le calcul des prédicats et les propriétés de *wp*, dont nous rappelons les plus utilisées :

$$\textbf{wp\_impl} \ : \quad \frac{p \Rightarrow r, r \Rightarrow wp(s,q), q \Rightarrow r}{p \Rightarrow wp(s,r)}$$

$$\textbf{wp\_disj\_conj} \ : \ \frac{p \Rightarrow wp(s,q), c \Rightarrow wp(s,d)}{(p \vee c) \Rightarrow wp(s, q \vee d) \ , \ (p \wedge c) \Rightarrow wp(s, q \wedge d)}$$

Nous rappelons qu'un programme UNITY est formalisé comme une liste d'instruction d'affectations. La sorte `Actlist`, qui représente la sorte des programmes, est construite avec la liste vide `nil` et le constructeur `cons`.

```
Set name List_of_act
```

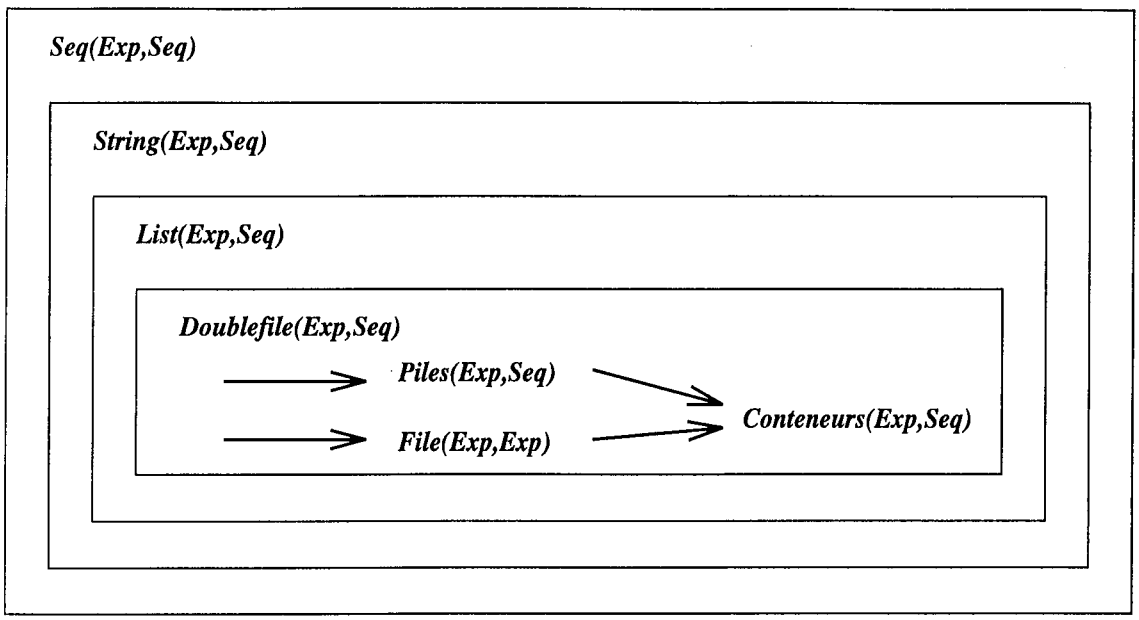



FIG. 3.2 – *La spécification des séquences*

```
declare operator
nil:              -> Actlist
cons:Act,Actlist -> Actlist
..
assert
  sort Actlist generated by nil, cons;
..
```

Nous disposons donc d'une règle d'induction sur la sorte `Actlist` qui nous permettra de prouver les théorèmes par induction.

```
sort Actlist generated by nil, cons:Act,Actlist -> Actlist
```

Nous avons prouvé la plupart des règles d'inférence mettant en jeu les prédicats *unless* et *ensures* par induction sur la liste des actions du programme. Ceci est dû au fait que ces prédicats sont définis inductivement sur les actions du programme:

```
declare operators
unless,ensures      :Bexp,Bexp,Actlist -> Bool
leads_to,exist_Act :Bexp,Bexp,Actlist -> Bool
wp                  :Act,Bexp          -> Bexp
```

### 3.5.1 Unless

Une propriété *unless* doit être vérifiée par toutes les transitions du programme. Rappelons la définition de la relation *unless*, ainsi que sa formalisation en LP.

Pour un programme $P$:

$$p \ unless \ q \stackrel{def}{=} \langle \forall s : s \in P :: \{p \wedge \neg q\} s \{p \vee q\} \rangle$$

Soit en utilisant le $wp$:

$$p \ unless \ q \equiv \langle \forall s : s \in P :: (p \wedge \neg q) \Rightarrow wp(s, p \vee q) \rangle$$

```
Set name unless
assert
 unless(p,q,anil) = true
 unless(p,q,a) = (((p ^ nnot(q)) \=> wp(a,p \or q))=T)
 unless(p,q,cons(a,act_1)) = unless(p,q,a) /\ unless(p,q,act_1)
```

Prouver une telle propriété revient à montrer qu'elle est préservée par toutes les actions du programme.

Comme nous le verrons dans les chapitres consacrés aux études de Cas, lorsque le programme est connu, la preuve d'une propriété *unless*, est semi-automatique. En effet, soit le programme `pgm` = $(a_2, \ldots, a_n)$, alors la formule `unless(p,q,pgm)` est réécrite de la façon suivante:

```
nnot(p) \or q \or wp(a2,p \or q) = T
 /\  nnot(p) \or q \or wp(a3,p \or q)  = T
 ...
 /\  nnot(p) \or q \or wp(an,p \or q)  = T
```

Pour chaque conjonction, le terme

```
nnot(p) \or q \or wp(ai,p \or q)
```

est simplifié en une tautologie si l'action *ai* ne modifie pas les variables de `(p \or q)`:

```
nnot(p) \or q \or wp(ai,p \or q)
→ nnot(p) \or q \or (p \or q)
→ T
```

Par conséquent, seule la vérification de la propriété pour les actions qui modifient le terme `(p \or q)` requiert éventuellement une interaction avec le démonstrateur.

Lorsque le programme n'est pas connu, pour montrer les propriétés vérifiées par la relation *unless* pour un programme quelconque représenté par une variable `pgm` de la sorte `Actlist`, nous utilisons l'induction. En effet, comme nous avons formalisé les programmes comme des listes d'affectations et l'opérateur `unless` inductivement sur cette liste, nous allons utiliser l'induction comme principale méthode de preuve.

L'une des règles d'inférence de l'opérateur *unless* est la règle d'*affaiblissement*:

$$\textbf{U\_consqWeak} : \frac{unless(p, q, pgm), q \Rightarrow r}{unless(p, r, pgm)}$$

**Preuve:**

| | |
|---|---|
| $\{p \wedge \neg q\} s \{p \vee q\}$ | (1) *hyp* |
| $q \Rightarrow r$ | (2) *hyp* |
| $\neg r \Rightarrow \neg q$ | (3) *contraposée avec* (2) |
| $p \wedge \neg r \Rightarrow p \wedge \neg q$ | (4) *propriétés de* $\Rightarrow$ *avec* (3) |
| $p \vee q \Rightarrow p \vee r$ | (5) *propriétés de* $\Rightarrow$ *avec* (2) |
| $(p \wedge \neg q) \Rightarrow wp(s, p \vee q)$ | (6) *définition de unless* |
| $\{p \wedge \neg r\} s \{p \vee r\}$ | *wp_impl avec* (4,5,6) |

Examinons maintenant la preuve mécanisée pour une instruction `a1` :

```
prove ((unless(p,q,l(a1))) /\ ((q \=> r)=T)) => (unless(p,r,l(a1))) by =>-m
    instantiate
      c by pc \or rc, q by pc \or qc,
      p by pc ^ nnot(rc), r by pc ^ nnot(qc),
      a1 by a1c  in wp_impl
      ..
qed
```

La seule interaction avec le démonstrateur consiste à instancier les variables de la règle `wp_impl` avec les constantes (les termes clos) correspondants. Les étapes de déduction de la preuve manuelle utilisant les hypothèses sont automatiques, car elles utilisent uniquement les propriétés des connecteurs logiques `^` , `\or` et `nnot`. Par conséquent, les étapes de calcul mettant en jeu les opérateurs logiques *prédicatifs* se ramènent à réécrire, en utilisant le système de réécriture définissant les expressions de prédicats (la base `Bool_str`, chapitre 2.3).

En fait, nous avons prouvé cette règle d'inférence directement pour un programme, par induction sur la liste d'action représentant le programme.

### LpPreuve 3.17

```
Set name u_consqWeak_pgm
prove
  ((unless(p,q,pgm))/\ ((q \=> r)=T)) => (unless(p,r,pgm))
   by induction pgm
..
    resume by =>-m
      instantiate
        c by pc \or rc,
        q by pc \or qc,
        p by pc ^ nnot(rc),
        r by pc ^ nnot(qc),
        a1 by ac in wp_impl
        ..
      set name lemme                    %pour utiliser l'hyp d'induction
      crit-p *hyp with *hyp
      crit-p lemme* with *hyp
qed
```

Le schéma de preuve de l'induction est le suivant :

```
Creating subgoals for proof by structural induction on 'pgm'
Basis subgoal:
  Subgoal 1: unless(p, q, nil) /\ q \=> r = T => unless(p, r, nil)
Induction constant: pgmc
Induction hypothesis:
  u_consqWeak_pgmInductHyp.1:
     unless(p, q, pgmc) /\ q \=> r = T => unless(p, r, pgmc)
Induction subgoal:
  Subgoal 2: unless(p, q, cons(a, pgmc)) /\ q \=> r = T
               => unless(p, r, cons(a, pgmc))
```

Le cas de base est automatiquement prouvé par LP, car `unless(p, q, nil)` se réduit à `true`. Le but du pas d'induction est sous forme d'une implication, nous procédons donc par `=>-method`, ce qui nous donne les hypothèses:

```
u_consqWeak_pgmImpliesHyp.1.1: nnot(pc) \or qc \or wp(ac, pc \or qc) -> T
```

```
u_consqWeak_pgmImpliesHyp.1.2: nnot(qc) \or rc -> T
u_consqWeak_pgmImpliesHyp.1.3: unless(pc, qc, pgmc) -> true
u_consqWeak_pgmInductHyp.1: unless(p, q, pgmc) /\ nnot(q) \or r = T
                                    => unless(p, r, pgmc)
                                    -> true
```

et le but est normalisé en utilisant la définition de \=> en :

```
(nnot(pc) \or rc \or wp(ac, pc \or rc) = T) /\ unless(pc, rc, pgmc)
```

Pour la première partie de la conjonction qui correspond au but `unless(pc, rc, l(ac))`, il suffit d'instancier la règle `wp_impl` comme pour la preuve "manuelle" précédente. Il nous reste à prouver le deuxième terme en utilisant l'induction. Pour cela, il suffit de remplacer dans l'hypothèse d'induction, `p` par `pc`, `q` par `qc` et `r` par `rc`. Ceci est fait par calcul de paires critiques entre l'hypothèse `u_consqWeak_pgmImpliesHyp.1.3` et l'hypothèse d'induction. Nous obtenons le fait

```
nnot(qc) \or r = T => unless(pc, r, pgmc)
```

avec lequel nous calculons les paires critiques en utilisant la règle `u_consqWeak_pgmImpliesHyp.1.2`, ce qui nous fournira le résultat.
□

La preuve mécanisée correspond à la preuve manuelle. La seule interaction avec le démonstrateur est l'instanciation de la règle `wp_impl`. Pour la suite, nous utilisons la méthode *classique* pour inférer les faits dont nous avons besoin, à savoir le calcul des paires critiques entre les hypothèses puis de nouveau entre les règles obtenues par ce calcul et les hypothèses.

**Remarque 3.8** *Pour la formalisation des règles d'inférence, nous pouvons utiliser les règles de déduction, dont la sémantique est celle d'une implication. La différence est essentiellement opérationnelle. De plus, la règle d'inférence, U_consqWeak, peut être formalisée comme une règle de déduction de deux façons différentes, suivant le degré d'automatisme que l'on désire obtenir :*

```
(1): when unless(p,q,pgm)/\ ((q \=> r)=T) yield unless(p,r,pgm)
(2): when unless(p,q,pgm),((q \=> r)=T) yield unless(p,r,pgm)
```

*La preuve de ces deux règles est identique à celle de l'implication, car* LP *transforme une conjecture sous forme de règle de déduction en une implication. Par contre, leurs utilisations diffèrent : pour inférer la conclusion* `unless(pc,rc,pgmc)` *en utilisant* `(1)`*, il faut inférer les faits* `unless(pc,qc,pgmc)` *et* `((qc \=>rc)=T)` *et instancier correctement les variables de la règle (1) par les constantes correspondantes. Par contre, avec* `(2)`*, dès que le fait* `unless(pc,qc,pgmc)` *est vrai, le système génère une deuxième règle de déduction :*

```
when ((qc \=> r)=T) yield (unless(pc,r,pgmc))
```

*puis dès que le fait* `((qc \=>rc)=T)` *est vrai, la conclusion de la règle est générée de façon automatique.*

*Néanmoins, nous utilisons peu les règles de déduction car elles ralentissent considérablement le système. En effet, dès qu'un nouveau fait est vrai,* LP *essaye de lui* appliquer *toutes les règles de déduction présentes dans le système (paragraphe 1.3.3). Bien que ce processus soit intéressant pour l'automatisation, il est coûteux car il génère des faits non pertinents pour la preuve courante et ralentit les preuves.*

A l'inverse des méthodes utilisées pour les naturels et les séquences, le calcul de paires critiques n'est pas une méthode très utilisée dans les preuves sur les opérateurs temporels, dû à la présence des connecteurs logiques et à leur manipulation. En effet, étudions la preuve de la propriété de *conjonction*[21] de *unless*

```
Set name u_conjonc_pgm
prove
   (unless(p,q,pgm) /\ unless(c,d,pgm))
      => unless((p ^ c),((p ^ d) \or (c ^ q) \or (q^ d)),pgm)
by induction on pgm
..
```

Nous procédons par induction sur la variable pgm. Le cas de base étant trivial, nous poursuivons par implication pour le pas d'induction. Nous obtenons le but[22] :

```
 dc \or nnot(cc) \or nnot(pc) \or qc
  \or wp(ac, (cc \or dc) ^ (pc \or qc))
 = T
 /\ unless(pc ^ cc, (cc \or dc) ^ (qc \or dc) ^ (pc \or qc), pgmc)
```

et les hypothèses

```
u_conjonc_pgmImpliesHyp.1.1: dc \or nnot(cc) \or wp(ac, cc \or dc) -> T
u_conjonc_pgmImpliesHyp.1.2: nnot(pc) \or qc \or wp(ac, pc \or qc) -> T
u_conjonc_pgmImpliesHyp.1.3: unless(cc, dc, pgmc) -> true
u_conjonc_pgmImpliesHyp.1.4: unless(pc, qc, pgmc) -> true
u_conjonc_pgmInductHyp.1: unless(p, q, pgmc) /\ unless(c, d, pgmc)
                                => unless(p ^ c,
                                          (c \or d) ^ (q \or d) ^ (p \or q),
                                          pgmc)
                                -> true
```

Pour prouver la première partie de la conjonction, il suffit d'utiliser la règle de disjonction du *wp* :

```
Set name wp_disj_conj
assert
(((p \=> wp(a1,q))=T) /\ ((c \=> wp(a1,d))=T))
 => ((((p \or c) \=> wp(a1,(q \or d)))=T)
       /\ (((p ^ c) \=> wp(a1, (q ^ d)))=T))
 ..
```

Si nous utilisons la méthode "classique", qui consiste à calculer les paires critiques entre les hypothèses et la règle wp_disj_conj, le processus est très long. En effet, le calcul de paires critiques entre l'hypothèse u_conjonc_pgmImpliesHyp.1.1et wp_disj_conj donne 33 paires critiques non triviales.
Par conséquent, nous préférons utiliser l'instanciation explicite, car d'après la preuve informelle, nous connaissons généralement la propriété à utiliser pour avoir le résultat.

Dans les preuves des théorèmes de l'opérateur *unless*, nous avons utilisé la possibilité de dés-activer certaines règles (les rendre **passives**, p. 19). En particulier, pour prouver la règle de *disjonction* de *unless* :

```
((unless(p,q,pgm))/\ (unless(c,d,pgm)))
```

21. Bien que les opérateurs temporels ne soient pas formalisés comme des relations binaires, nous utilisons la terminologie de Chandy et Misra [Cha88] et par conséquent, nous utilisons les mêmes noms pour les règles dérivées. De plus, ces règles, qui sont des théorèmes à prouver, ne dépendent pas du programme considéré.

22. L'implication, \=>, est définie à l'aide de **nnot** et \or

```
   => (unless((p \or c),(((nnot(p)^ d) \or (nnot(c)^ q)) \or (q^ d)),pgm))
  ..
```

nous avons "caché" les deux règles suivantes :

```
bool_str.18(IP): p \or (q ^ c) -> (p \or q) ^ (p \or c)
bool_str.19(IP): (p ^ q) \or (p ^ c) -> p ^ (q \or c)
```

C'est à dire que nous les avons immunisées et dés-activées dans le but d'éviter la réécriture en forme normale des termes de la forme (p \or (q ^ c)). Cela permet une utilisation plus directe des règles (pour l'instanciation), afin d'effectuer des preuves mécanisées dont la structure est celle des preuves manuelles.

### 3.5.2 Ensures

La relation *ensures* est la propriété de base pour exprimer le progrès en UNITY, dont la définition est :

$$p \ \ ensures \ \ q \stackrel{def}{=} (p \ \ unless \ \ q) \wedge \langle \exists s : s \in P :: \{p \wedge \neg q\} s \{q\} \rangle$$

Sa définition en LP est :

```
Set name ensures
assert
 ensures(p,q,anil) = false
 ensures(p,q,pgm) = unless(p,q,pgm) /\ exists_act(p,q,pgm)
..

Set name exist_act
assert
 exists_act(p,q,anil) = false
 exists_act(p,q,l(a)) = (((p ^ nnot(q)) \=> wp(a,q)) = T)
 exists_act(p,q,cons(a,act_l))
                  = exists_act(p,q,a) \/ exists_act(p,q,act_l)
```

De même que pour la relation *unless*, lorsque le programme est connu, la preuve d'une propriété *ensures* est semi-automatique. En effet, soit le programme `pgm` = $(a_2, \ldots, a_n)$, alors la formule `ensures(p,q,pgm)` est réécrite de la façon suivante :

```
nnot(p) \or wp(a2,p \or q) \or q = T
 /\  nnot(p) \or wp(a3,p \or q) \or q = T
 ...
 /\  nnot(p) \or wp(an,p \or q) \or q = T
 /\  nnot(p) \or wp(a2,q) \or q = T
       \/ nnot(p) \or wp(a3,q) \or q = T
       ...
       \/ nnot(p) \or wp(an,q) \or q = T)
```

Mis à part la dernière conjonction, qui correspond à la partie `exist_act`, les différentes conjonctions correspondent à la preuve de `unless(p,q,pgm)`. Pour vérifier une telle propriété, LP utilise les définitions de *wp*, celles des affectations et des substitutions, pour réécrire cette formule à `true` et par conséquent, s'il existe une action dans le programme vérifiant la partie `exist_act`, toute la disjonction sera réécrite à `true` de façon transparente.

Lorsque le programme n'est pas connu, pour montrer les propriétés vérifiées par la relation *ensures* pour un programme quelconque représenté par une variable `pgm` de la sorte `Actlist`,

nous utilisons l'induction sur `pgm`.

Tout programme UNITY contient au moins une action. En effet, les propriétés de progrès sont définies pour des programmes ayant au moins une instruction :

Si $P \equiv \emptyset$

$$\begin{aligned} p \ \textit{ensures} \ q &\equiv (p \ \textit{unless} \ q) \wedge \langle \exists s : \textit{false} :: \{p \wedge \neg q\} s \{q\} \rangle \\ &\equiv (p \ \textit{unless} \ q) \wedge \langle \textit{false} \rangle \\ &\equiv \textit{false} \end{aligned}$$

Par conséquent, certains théorèmes sur l'opérateur *ensures* seront prouvés avec un programme de la forme `cons(a1,pgm)`, pour s'assurer qu'il existe au moins une action dans le programme.

La première propriété que doit vérifier l'opérateur *ensures* est la réflexivité :

```
Set name reflex_ensures
prove ensures(p,p,cons(a1,pgm))
qed
```

Cette propriété est prouvée par normalisation car *unless* est réflexif et `(p ^ nnot(p)) -> T`. Par ailleurs, ce théorème est éliminé de la base car il est normalisé à `true`. Les principales étapes de la réécriture sont :

```
1.  ensures(p, p, cons(a1, pgm))
2.  unless(p, p, cons(a1, pgm)) /\ exist_Act(p, p, cons(a1, pgm))
3.  unless(p, p, cons(a1, pgm))
     /\ (nnot(p)  \or p \or wp(a1, p)= T \/ exist_Act(p, p, pgm))
...
6.  unless(p, p, cons(a1, pgm)) /\ (true \/ exist_Act(p, p, pgm))
...
12.  true /\ unless(p, p, pgm)
13.  unless(p, p, pgm)
14.  true
```

Étudions maintenons la preuve de la propriété d'*affaiblissement*:

$$\textbf{E\_consqWeak} \quad : \frac{\textit{ensures}(p, q, pgm), q \Rightarrow r}{\textit{ensures}(p, r, pgm)}$$

Intuitivement, ayant déjà prouvé le résultat correspondant pour *unless*, la preuve devrait être immédiate puisque *ensures* est défini en fonction de *unless*.

En effet, la propriété s'écrit :

```
ensures(p, q, pgm) /\ (q \=> r = T) => ensures(p, r, pgm)
```

puis se réécrit en

```
unless(p, q, pgm) /\ exist_Act(p, q, pgm)/\ nnot(q) \or r = T
  => unless(p, r, pgm) /\ exists_Act(p, r, pgm)
```

En utilisant la règle correspondante pour *unless*, et la méthode *implique*, le but est réduit à

```
exists_Act(pc, rc, pgm)
```

avec les hypothèses :

```
nnot(qc) \or rc -> T
exist_Act(pc, qc, pgmc) -> true
unless(pc, qc, pgmc) -> true
```

Nous ne pouvons résoudre le but car nous n'avons aucune règle à notre disposition pour le faire. L'idée est d'utiliser l'induction sur la variable pgm. Cette induction doit se faire avant l'utilisation de la méthode *implique*, sinon nous obtenons des constantes sur lesquelles nous ne pouvons pas appliquer l'induction.

Nous débutons par une induction sur la variable pgm :

```
Creating subgoals for proof by structural induction on 'pgm'
Basis subgoal:
  Subgoal 1: nnot(q) \or r = T /\ exist_Act(p, q, nil) /\ unless(p, q, nil)
             => unless(p, r, nil) /\ exist_Act(p, r, nil)
Induction constant: pgmc
Induction hypothesis:
  E_consqweak_pgmInductHyp.1:
     nnot(q) \or r = T /\ exist_Act(p, q, pgmc) /\ unless(p, q, pgmc)
       => unless(p, r, pgmc) /\ exist_Act(p, r, pgmc)
Induction subgoal:
  Subgoal 2: nnot(q) \or r = T
             /\ exist_Act(p, q, cons(a, pgmc))
             /\ unless(p, q, cons(a, pgmc))
             => unless(p,r,cons(a,pgmc)) /\ exist_Act(p,r,cons(a, pgmc))
```

▶ Cas de base : trivial car l'antécédent de l'implication est réduit à false.
▶ Pas d'induction : nous procédons par implication, nous obtenons les hypothèses :

```
E_consqweak_pgmImpliesHyp.1.1: nnot(pc) \or qc \or wp(ac, pc \or qc) -> T
E_consqweak_pgmImpliesHyp.1.2: nnot(qc) \or rc -> T
E_consqweak_pgmImpliesHyp.1.3: nnot(pc) \or qc \or wp(ac, qc) = T
                                  \/ exist_Act(pc, qc, pgmc)
                                -> true
E_consqweak_pgmImpliesHyp.1.4: unless(pc, qc, pgmc) -> true
E_consqweak_pgmInductHyp.1: nnot(q) \or r = T
                              /\ exist_Act(p, q, pgmc)
                              /\ unless(p, q, pgmc)
                            => unless(p, r, pgmc)
                                 /\ exist_Act(p, r, pgmc)
                               -> true
```

et le but

```
unless(pc,rc,cons(ac,pgmc)) /\ exist_Act(pc,rc,cons(ac, pgmc))
```

normalisé en utilisant la définition de unless :

```
nnot(pc) \or rc \or wp(ac, pc \or rc) = T
  /\ unless(pc, rc, pgmc)
  /\ (nnot(pc) \or rc \or wp(ac, rc) = T \/ exist_Act(pc, rc, pgmc))
```

Nous commençons par prouver le but unless(pc,rc,pgmc) en utilisant la règle correspondante de l'opérateur *unless*, u_consqWeak. Pour cela, un calcul de paire critique n'est pas très utile car la

variable pgm doit être instanciée par le terme cons(ac,pgmc) et non par pgmc, afin de déduire le fait unless(pc,qc,cons(ac,pgmc)), qui est subdivisé en deux règles (une pour la première action de la liste représentant le programme, et une pour le reste de la liste) :

```
u_consqWeak_pgm.1.1.1: nnot(pc) \or rc \or wp(ac, pc \or rc) = T
u_consqWeak_pgm.1.1.2: unless(pc, rc, pgmc)
```

Ces deux faits permettent de simplifier le but à

```
nnot(pc) \or rc \or wp(ac, rc) = T \/ exist_Act(pc, rc, pgmc)
```

Celui ci se ramène à montrer qu'il existe une action dans le programme "vérifiant" la propriété. Pour cela, nous allons procéder par cas, car l'hypothèse dont nous avons besoin est une implication :

```
E_consqweak_pgmImpliesHyp.1.3: nnot(pc) \or qc \or wp(ac, qc) = T
                                        \/ exist_Act(pc, qc, pgmc)
                                        -> true
```

Nous avons deux cas, soit l'action recherchée est ac, soit elle appartient à pgmc.
▶▶ Cas (nnot(pc) \or qc \or wp(ac, qc) = T) : nous procédons comme pour la preuve de *U_ conqWeak*, en utilisant la règle **wp_impl** : ayant $p \Rightarrow q$, $q \Rightarrow wp(a,r)$,$r \Rightarrow c$, on en déduit $p \Rightarrow wp(a,c)$.
□
▶▶ Cas exist_Act(pc,rc,pgmc) : nous utilisons l'hypothèse d'induction dont tous les antécédents sont vrais, afin de déduire exist_Act(pc,rc,pgmc). Ce qui est traduit en LP par un calcul de paires critiques entre les hypothèses de l'implication et l'hypothèse d'induction puis par le calcul des paires critiques entre les règles obtenues et les hypothèses de l'implication . □

La preuve correspondante est :

**LpPreuve 3.18**

```
Set name E_consqweak_pgm
prove
  (ensures(p,q,pgm) /\ ((q \=> r)=T)) => (ensures(p,r,pgm))
by induction pgm
..
    resume by =>-m
      instantiate
        p by pc, q by qc, pgm by cons(ac,pgmc),r by rc in u_consqweak_pgm
        ..
      resume by case (((nnot(qc)^ pc) \=>  wp(ac, qc)) = T)
        instantiate
          p by pc^ nnot(rc), r by pc^ nnot(qc), q by qc, c by rc,
          a1 by ac
        in wp_impl
        ..
        set name lemme
        crit-p *ImpliesHyp with *InductHyp
        crit-p *ImpliesHyp with lemme*
qed
```

**Stratégie** Les preuves des théorèmes de l'opérateur *ensures* ont une structure "générale". Nous commençons par une induction sur la variable `pgm`, où le cas de base est prouvé de façon automatique. Pour le pas d'induction, la stratégie consiste à prouver le but *unless*, puis à utiliser la preuve par cas pour montrer l'existence d'une action vérifiant la propriété. Les deux cas sont donnés par l'hypothèse d'induction qui contient une disjonction de la forme

```
((P ^ nnot(Q)) \=> wp(a1,Q)) \/ exist_act(P,Q,pgm)
```

Le premier cas consiste à vérifier la propriété lorsque la "première" action vérifie l'hypothèse d'induction. Nous utilisons alors l'instanciation des règles auxiliaires dont nous avons besoin, que nous avions utilisé dans la preuve "manuelle".
Le deuxième cas correspond à la situation où l'action cherchée se trouve dans le reste de la liste représentant le programme. Dans ce cas, nous utilisons l'hypothèse d'induction. "Utiliser l'hypothèse d'induction" se ramène à un premier calcul de paires critiques entre les hypothèses de l'implication (prémisse de la règle d'inférence à prouver) et l'hypothèse d'induction, puis un second calcul entre les règles obtenues par le calcul précédent et les hypothèses de l'implication. Ce double calcul est dû au fait que nous avons plusieurs hypothèses d'implication (prémisses). Ces deux commandes peuvent être remplacées par une seule commande où on instancie les variables de l'hypothèse d'induction par les constantes correspondantes (si cette instanciation donne un résultat immédiat (remarque 3.8)). Néanmoins, nous utilisons le première méthode qui se trouve être plus générale puisque nous n'utilisons pas les noms des variables. Le nombre de commandes de calcul de paires critiques dépend de la forme des hypothèses. Suivant le nombre de variables en commun entre les différentes hypothèses, nous pouvons déduire le nombre de commandes de calcul de paires critiques à faire.

### 3.5.3 Leads_to

La relation *ensures* permet de définir une relation de progrès plus puissante, à savoir *leads_to*. Cet relation est définie uniquement par les trois règles d'inférence, les deux premières sont formalisées comme des règles de déduction, la dernière comme une implication :

- Lorsque la propriété *ensures(p,q,pgm)* est valide dans le système courant, on infère *leads_to(p,q,pgm)*.

$$\textbf{LEADS\_TO} : \frac{ensures(p, q, pgm)}{leads\_to(p, q, pgm)}$$

- Lorsque les faits *leads_to(p,r,pgm)* et *leads_to(r,q,pgm)* sont valides, on infère *leads_to(p,q,pgm)*.

$$\textbf{TRANS\_LEAD} : \frac{leads\_to(p, r, pgm), leads\_to(r, q, pgm)}{leads\_to(p, q, pgm)}$$

- Enfin, nous avons axiomatisé une disjonction finie pour l'opérateur `leads_to`. Lorsque *leads_to(p1,q,pgm)* et *leads_to(p2,q,pgm)* sont valides, on infère *leads_to(p1∨ p2,q,pgm)*.

$$\textbf{DISJ\_LEAD} : \frac{leads\_to(p1, q, pgm), leads\_to(p2, q, pgm)}{leads\_to(p1 \vee p2, q, pgm)}$$

De façon générale, nous ne pouvons montrer les théorèmes du prédicat *leads_to*, car ces preuves nécessitent une induction sur la structure de la preuve, non formalisable dans notre système.

Comme exemple, nous allons décrire la preuve de la règle d'*impossibilité* (où $\mapsto$ représente la relation *leads_to*) :

$$IMPO\_LEAD : \frac{p \mapsto false}{\neg p}$$

La preuve utilise une induction sur le nombre de règles d'inférence appliquées pour inférer $p \mapsto false$.

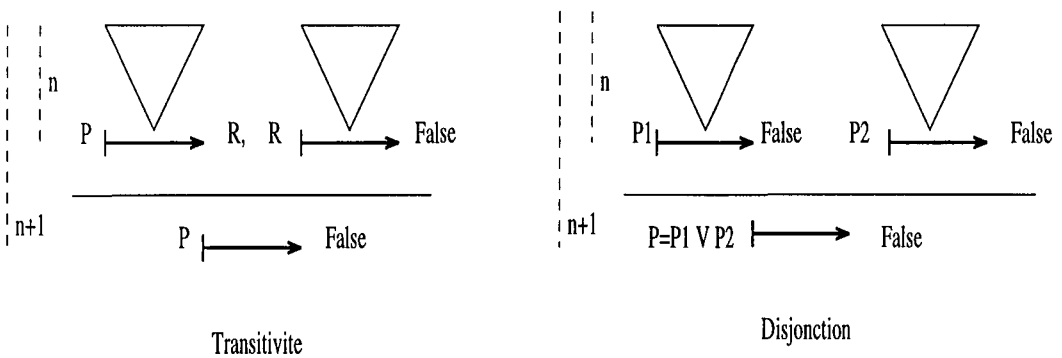



FIG. 3.3 – *Preuve de la règle d'*impossibilité

**Cas de base** : La longueur de la preuve est 1, on a inféré $p \mapsto false$ avec l'application d'une seule règle d'inférence, en l'occurrence la règle *LEADS_TO*. Le but est déduit en utilisant le théorème correspondant pour *ensures*.

**Hypothèse d'induction** : La propriété est vraie si le nombre de règles utilisées est inférieur ou égal à $n$.

**Pas d'induction** : Le nombres de règles appliquées pour inférer $p \mapsto false$ est $n+1$ (Figure 3.3). A l'étape $n$, nous avons donc appliqué la règle de transitivité ou bien celle de la disjonction :

**Transitivité** :

| | |
|---|---|
| $r \mapsto false$ | (1) *Hyp*1 |
| $\neg r$ | (2) *Hyp d'induction* sur (1) |
| $p \mapsto r$ | (3) *Hyp*2 |
| $p \mapsto false$ | (5) *avec* (3) et (2) |
| $\neg p$ | *Hyp d'induction sur* (5) |

**Disjonction** :

| | |
|---|---|
| $p_1 \mapsto false$ | (1) Hyp1 |
| $\neg p_1$ | (2) *Hyp d'induction sur* (1) |
| $p_2 \mapsto false$ | (3) *Hyp*2 |
| $\neg p_2$ | (4) *Hyp d'induction sur* (2) |
| $p \equiv p_1 \vee p_2$ | (5) *Hyp*3 |

| | |
|---|---|
| $\neg p \equiv \neg p_1 \wedge \neg p_2$ | (7) $CP1$ *avec* (5) |
| $\neg p$ | $CP1$ *avec* (2) et (4) |

Nous ne pouvons donc mécaniser cette preuve dans notre système puisque le raisonnement se fait sur le nombre d'application des règles d'inférence. Néanmoins, nous avons prouvé les règles qui ne nécessitent pas ce type de raisonnement, telle que la propriété suivante:

$$\textbf{Impl\_lead} : \frac{p \Rightarrow q}{p \mapsto q}$$

La preuve utilise la propriété correspondante pour *ensures* et la règle *LEADS_ TO*.

**LpPreuve 3.19**

```
Set name impl_lead
prove ((p \=> q)=T) => leads_to(p,q,cons(a1,pgm))
  resume by =>-m
    <> => subgoal
    instantiate p by pc,q by qc in ensures_coroll1
    instantiate p by pc,q by qc,pgm by cons(a1,pgm) in leads_to
    [] => subgoal
  [] conjecture
qed
```

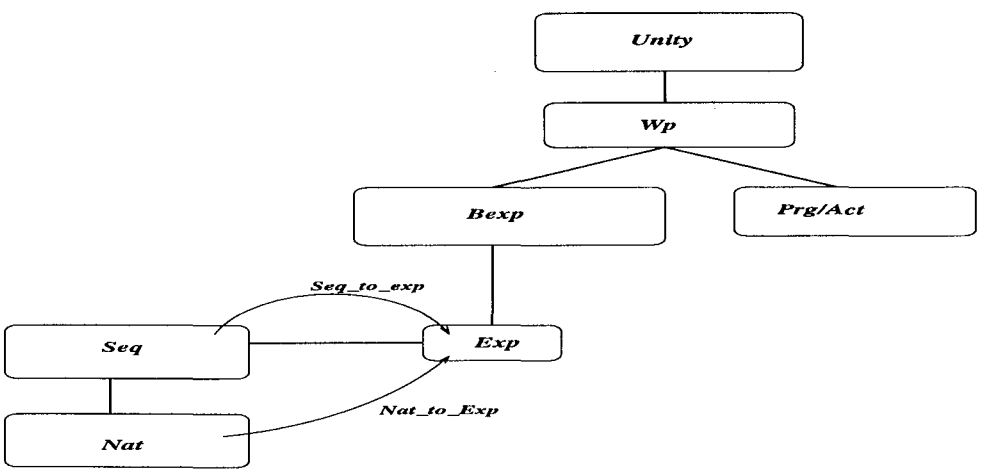



FIG. 3.4 – *La base de règles*

## 3.6 Conclusion

Nous avons décrit dans ce chapitre la construction de la base de faits, à savoir les entiers naturels, les séquences et les opérateurs temporels. Nous avons vu que cette construction est basée sur une méthode incrémentale, qui consiste à introduire une base initiale que nous enrichissons progressivement. Une étape intermédiaire réside dans la validation ou certification de la base, en vérifiant les propriétés que la spécification est supposée satisfaire.

Les preuves sur les spécifications des entiers naturels et les séquences sont essentiellement basées sur l'induction. La plupart des preuves sur les opérateurs temporels nécessitent moins d'interaction que pour les preuves sur les entiers ou bien sur les séquences, car les opérateurs temporels sont définis sur les expressions de prédicats (expressions booléennes) et les preuves se

ramènent dans la plupart des cas à éliminer les tautologies. Néanmoins dans tous les cas, une preuve mécanisée nécessite une étape de conception, puis la traduction en LP puis une validation. Par conséquent, la structure de la preuve mécanisée est généralement similaire à celle effectuée à la main. Il est donc nécessaire de connaître a priori la structure générale d'une démonstration, afin de choisir une méthode de preuve.
La réussite d'une preuve mécanisée n'est pas une garantie de la validité du fait prouvé, car cette preuve peut être fondée sur des hypothèses inconsistances, d'où la nécessité de valider par des preuves toutes les stratégies de preuves, comme par exemple les schémas d'induction dérivés d'un schéma principal. Néanmoins, il est toujours agréable pour l'utilisateur de laisser faire par le démonstrateur les phases fastidieuses, comme la simplification des expressions booléennes.

# Chapitre 4

# Une nouvelle axiomatisation

## 4.1 Introduction

Les améliorations que nous présentons dans ce chapitre concernent d'une part la formalisation de l'opérateur *ensures* et d'autre part, celle de *leads_to*. Ces améliorations proviennent de l'introduction du quantificateur existentiel, qui permet une formalisation plus intuitive et plus proche de la philosophie UNITY. Les améliorations décrites dans ce chapitre, ont été effectuées durant la rédaction de ce document et par conséquent, nous n'avons pas encore approfondi leur impact, en particulier pour les études de cas. Cependant, il nous a paru intéressant de les décrire, pour illustrer l'apport de la quantification existentielle pour l'expressivité du langage, pour montrer que les formalisations ne sont pas figées et enfin pour étudier la mécanisation des preuves manipulant les quantificateurs existentiels.

L'introduction de l'opérateur existentiel nous permet d'améliorer la formalisation des relations de progrès. La définition axiomatique de la relation *ensures* utilisait la relation *unless* et un prédicat qui testait l'existence d'une action dans le programme vérifiant la propriété. Ayant à notre disposition le quantificateur existentiel, nous pouvons alors proposer une autre formalisation de *ensures* plus intuitive. La relation de progrès, *leads_to* est définie par trois règles d'inférence, que nous avons formalisé comme des implications. Cette formalisation ne nous permettait pas de prouver les règles dérivées (théorèmes) de la relations *leads_to*. L'introduction de l'opérateur existentiel nous permettra d'améliorer la formalisation de cette relation de progrès, et notamment de prouver les règles d'inférence, à l'aide d'un schéma d'induction basé sur une relation nothérienne.

Il est certain que pour représenter un programme dont les instructions peuvent être exécutées de façon indéterministe, il est plus naturel de le formaliser par une structure de type *ensemble* plutôt que par celle d'*une liste* où il y a une notion implicite d'ordre. Nous modifions alors la formalisation présentée au chapitre 2 pour représenter un programme comme un ensemble d'actions et non plus comme une liste, de façon à pouvoir utiliser l'opérateur \in.

```
set name set_of_act
declare variables  x, y, z: Actset
declare operators
  {}    :              -> Actset
  {__}  : Act          -> Actset
  insert: Act, Actset -> Actset
  __\u__: Actset, Actset -> Actset
  __\in__:Act, Actset -> Bool
```

```
    __\<__:Actset, Actset -> Bool
    ..
  assert
  sort Actset generated by {},{__},\u;
  sort Actset generated by {},insert;
  sort Actset partitioned by \in;
  ac \u
..
```

## 4.2 Ensures

Pour formaliser l'opérateur *ensures*, nous allons utiliser le quantificateur existentiel. En effet, la différence entre les opérateurs *unless* et *ensures* réside dans le fait que le deuxième nécessite l'existence de l'action qui permet d'établir le prédicat $q$:

$$p \textit{ ensures } q \equiv (p \textit{ unless } q) \wedge \langle \exists s : s \in P :: \{p \wedge \neg q\} s \{q\} \rangle$$

Nous définissons *ensures* de la façon suivante:

```
assert
ensures(p,q,pgm)
  = (unless(p,q,pgm)
   /\  (\E a1 ((a1 \in pgm) /\  (((p^ nnot(q)) \=> wp(a1,q))=T))))
..
```

Pour l'opérateur *unless*, la seule modification par rapport à la formalisation précédente (cf. paragraphe 2.7) consiste à modifier la définition[23] de façon que le troisième argument, représentant le programme, soit une structure de donnée de type *ensemble* (de la sorte `Actset`) et non une structure de donnée de type *liste* (de la sorte `Actlist`):

```
Set name unless
assert
unless(p,q,{}) = true;
unless(p,q,{a1}) = (((p ^ nnot(q)) \=> wp(a1,p \or q))=T);
unless(p,q,insert(a1,a_set)) = (unless(p,q,{a1}) /\ unless(p,q,a_set));
..
```

Les preuves des règles d'inférence de l'opérateur *unless* (les théorèmes) sont identiques. En effet, dans la version précédente, nous avons utilisé une induction sur la structure du programme représenté par une liste d'actions `cons(`$a_1$`,cons(...,cons(`$a_n$`,nil)))`. Cette induction était basée sur le schéma d'induction:

```
sort Actlist generated by nil, cons
```

Le changement réside dans le schéma d'induction utilisé. Pour la sorte `Actset`, nous avons défini trois schémas d'induction et nous utilisons le suivant pour les preuves par induction sur la variable `pgm` de type `Actset`:

```
set name pgm_induct
sort Actset generated by {},insert
```

23. De même que pour la version précédente, bien que le deuxième axiome et le troisième axiome peuvent être écrits à l'aide d'un seul axiome, nous utilisons un axiome supplémentaire pour exprimer la propriété pour une action du programme.

Par conséquent, les preuves des propriétés *unless* ont la même structure mis à part le fait que nous précisons la règle d'induction à utiliser, à savoir `pgm_induct`. Nous remplaçons les termes `l(`$a\_i$`)`, où `l` avait la signature `Act -> Actlist`, par les termes `{`$a\_i$`}`, où `{}` a la signature `Act -> Actset`.

Comparons maintenant la formule `ensures(p,q,cons(a2,pgm))`, où `cons` était le constructeur des listes, à la formule `ensures(p,q,insert(a2,pgm))`, où `insert` est le constructeur des ensembles. Les formes normales respectives sont :

(1):
```
nnot(p) \or wp(a2,p \or q) \or q = T
 /\  unless(p,q,pgm)
 /\  ((nnot(p) \or wp(a2,q) \or q = T) \/  exist_Act(p,q,pgm))
```
(2)
```
nnot(p) \or wp(a2,p \or q) \or q = T
 /\  unless(p,q,pgm)
 /\  \E a1 ((a1 = a2 \/  a1 \in pgm) /\  nnot(p) \or wp(a1,q) \or q = T)
```

De façon intuitive, montrer une formule de la forme (1) nécessite une preuve par cas pour résoudre la disjonction. Ce traitement par cas n'apparaît pas dans les preuves des programmes particuliers du fait que LP essaye de réécrire l'un des termes

```
p ^ q \=> wp(a2,q)
```

à `true`, pour toutes les actions du programme. En effet, pour un programme $(a_2, \ldots, a_n)$, la formule (1) est réécrite en

```
nnot(p) \or wp(a2,p \or q) \or q = T
 /\  nnot(p) \or wp(a3,p \or q) \or q = T
 ...
 /\  nnot(p) \or wp(an,p \or q) \or q = T
 /\  (nnot(p) \or wp(a2,q) \or q = T
       \/ nnot(p) \or wp(a3,q) \or q = T
       ...
       \/ nnot(p) \or wp(an,q) \or q = T)
```

Puis LP utilise les définitions de *wp*, des affectations et substitutions pour réécrire cette formule à `true` et par conséquent, s'il existe une action dans le programme considéré vérifiant la partie `exist_act`, toute la disjonction sera réécrite à `true` de façon transparente.

Par contre, pour montrer une formule de la forme (2), il nous faut instancier de façon explicite la variable `a1` avec l'action vérifiant la propriété, à cause du quantificateur existentiel (cf. chapitre 3). Certes, cette méthode est moins automatique puisqu'elle nécessite plus d'interaction avec le démonstrateur, mais lorsqu'il s'agit de vérifier un programme particulier, l'action est connue et l'instanciation est évidente. Par contre, pour les preuves des théorèmes sur un programme quelconque, l'instanciation n'est pas triviale. C'est ce que nous allons détailler dans ce qui suit.

## 4.3 Preuves des règles d'inférence

### 4.3.1 Réflexivité

Soit le théorème de la *réflexivité*[24] de l'opérateur *Ensures*. Nous allons comparer les preuves avec les deux formalisations. Avec la formalisation utilisant les listes, la preuve est automatique

24. Voir la note 21.

par normalisation, car *unless* est réflexif et `(p ^ nnot(p))` se réduit à `T`.

```
Set name E_reflex
prove ensures(p,p,cons(a1,pgm))
qed
```

Par ailleurs, ce théorème est éliminé de la base car il est normalisé à `true`. Les principales étapes de la réécriture sont :

```
1.  ensures(p, p, cons(a1, pgm))
2.  unless(p, p, cons(a1, pgm)) /\ exist_Act(p, p, cons(a1, pgm))
3.  unless(p, p, cons(a1, pgm))
     /\ (nnot(p)  \or p \or wp(a1, p)= T \/ exist_Act(p, p, pgm))
...
6.  unless(p, p, cons(a1, pgm)) /\ (true \/ exist_Act(p, p, pgm))
...
12.  true /\ unless(p, p, pgm)
13.  unless(p, p, pgm)
14.  true
```

De même, la preuve avec la formalisation utilisant les ensembles est immédiate, mais moins automatique dû à la présence du quantificateur existentiel :

```
Set name E_reflex
prove (ensures(p,p,insert(a1,pgm)))
  resume by specializing a2 to a1
    <> specialization subgoal
    [] specialization subgoal
  [] conjecture
qed
```

Cette propriété revient à montrer la formule suivante :

```
\E a2 (a2 \in insert(a1,pgm))
```

En effet, le théorème est réécrit de la façon suivante :

```
ensures(p,p,insert(a1,pgm))
→ unless(p,p,insert(a1,pgm))
   /\  \E a2 (a2 \in insert(a1,pgm) /\  ((p ^ nnot(p)) \=> wp(a2,p)= T))
```

Or comme *unless* est réflexif et `(p ^ nnot(p)) -> T`, cette formule est normalisée à

```
\E a2 (a2 \in insert(a1,pgm))
→ \E a2 (a2 = a1 \/ a2 ∈ pgm)
```

Il suffit alors de "spécialiser" `a2` à `a1`, c'est à dire de "choisir" comme nom pour `a2`, `a1` (cf. Chapitre 1, paragraphe 1.3.2).

Dans les cas plus complexes, la représentation du programme par un ensemble est plus adéquate et intuitive que la représentation par une liste.

### 4.3.2 Affaiblissement

Prenons comme exemple la propriété d'*affaiblissement* de la relation *Ensures* :

$$\textbf{E\_consqWeak} \quad : \frac{ensures(p, q, pgm), q \Rightarrow r}{ensures(p, r, pgm)}$$

Nous avons vu, au chapitre précédent, que la preuve nécessite une induction, puis une preuve par cas pour montrer qu'il existe une action dans le programme vérifiant `ensures(p,r,pgm)` étant donné `ensures(p,q,pgm)`. Nous allons voir que la preuve où le programme est représenté par un ensemble est plus intuitive.

La propriété LP à prouver est

```
(ensures(p,q,pgm) /\ ((q \=> r)=T)) => ensures(p,r,pgm)
```

En procédant par "implication", nous obtenons les hypothèses :

```
E_consqweakImpliesHyp.1.1: nnot(qc) \or rc -> T
E_consqweakImpliesHyp.1.2: unless(pc,qc,pgmc) -> true
E_consqweakImpliesHyp.1.3:
    \E a1 (a1 \in pgmc /\  nnot(pc) \or qc \or wp(a1,qc) = T)
    -> true
```

et le but

```
unless(pc,rc,pgmc)
 /\  \E a1 (a1 \in pgmc /\  nnot(pc) \or rc \or wp(a1,rc) = T)
```

Pour la première partie de la conjonction, il suffit d'appliquer aux deux premières hypothèses de l'implication, la règle correspondante prouvée pour *unless*. Il nous reste donc à montrer qu'il existe une action $a$ dans le programme *pgmc* vérifiant $p \wedge \neg r \Rightarrow wp(a, r)$.

D'après la troisième hypothèse de l'implication, nous savons qu'il existe une action dans *pgmc* vérifiant $p \wedge \neg q \Rightarrow wp(a, q)$. Nous allons alors skolémiser la formule `E_consqweakImpliesHyp.1.3`, en remplaçant `a1` par `sk(x1,...,xn)` où `sk` est un symbole de fonction[25] et `x1,...,xn` sont les variables liées par le quantificateur existentiel.
En LP, cela revient à "fixer" (nommer) la variable quantifiée `a1` de l'hypothèse par une constante `skact` de type `Act` (le nombre de variables liées par le quantificateur étant nul, une fonction d'arité 0 suffit). Ce qui nous permet de déduire les deux propriétés suivantes, à partir de l'hypothèse `E_consqweakImpliesHyp.1.3`:

```
E_consqweak.2.1: nnot(pc) \or qc \or wp(skact,qc) = T
E_consqweak.2.2: skact \in pgmc
```

Pour résoudre le but

```
\E a1 (a1 \in pgmc /\  nnot(pc) \or rc \or wp(a1,rc) = T)
```

il suffit de choisir une action, en "spécialisant" la variable liée `a1` à `skact` pour obtenir le but :

```
nnot(pc) \or rc \or wp(skact,rc) = T
```

Ayant les hypothèses :

```
E_consqweakImpliesHyp.1.1: nnot(qc) \or rc -> T
E_consqweak.2.1: nnot(pc) \or qc \or wp(skact,qc) = T
```

25. qui n'apparaît dans aucun fait de la base.

Il suffit d'appliquer la règle **wp_impl** (instancier correctement les variables de cette règle) pour déduire le résultat.

$$\textbf{wp\_impl} : \frac{p \Rightarrow r, r \Rightarrow wp(s,q), q \Rightarrow r}{p \Rightarrow wp(s,r)}$$

En effet, les étapes de déduction[26] $q \Rightarrow r$, $\neg r \Rightarrow \neg q$, $p \wedge \neg r \Rightarrow p \wedge \neg q$, $p \vee q \Rightarrow p \vee r$ sont effectuées de façon transparente. □

Nous utilisons le même principe pour montrer la règle d'*impossibilité* de l'opérateur *Ensures*:

$$\textbf{E\_impo} : \frac{p \ ensures \ false}{\neg p}$$

dont la preuve en LP est:

```
set name E_impo
prove ensures(p,F,pgm) => (nnot(p)=T)
 resume by =>-m
  <> => subgoal
  declare operator skact:-> Act
  fix a1 as skact in *hyp
  [] => subgoal
 [] conjecture
qed
```

Les hypothèses sont:

```
E_impoImpliesHyp.1.1: unless(pc,F,pgmc) -> true
E_impoImpliesHyp.1.2: \E a1 (a1 \in pgmc /\  nnot(pc) = T) -> true
```

et le but

```
nnot(pc) = T
```

Il suffit alors de "nommer" la variable dans l'hypothèse pour avoir le résultat. La preuve avec la formalisation *liste* nécessitait une preuve par induction puis par cas.

Dans ce qui suit, nous allons détailler la preuve d'une formule plus complexe, qui contient deux opérateurs existentiels. La gestion de ces quantificateurs est délicate, et peut engendrer une preuve incorrecte, si on introduit des axiomes arbitraires non justifiés.

### 4.3.3 Conjonction

Soit la règle d'inférence suivante que nous désignons par la propriété de *conjonction*, afin de conserver la terminologie de Chandy et Misra:

$$\textbf{E\_conjonc} : \frac{p \ unless \ q \ , \ c \ ensures \ d}{(p \wedge c) \ ensures \ (p \wedge d) \vee (c \wedge q) \vee (q \wedge d)}$$

La preuve utilise les règles suivantes:

$$\textbf{U\_conjonc} : \frac{p \ unless \ q, c \ unless \ d}{(p \wedge c) \ unless \ (p \wedge d) \vee (c \wedge q) \vee (q \wedge d)}$$

26. Preuve du résultat correspondant pour la relation *unless* est détaillée au chapitre 3, paragraphe 3.5

$$\textbf{wp\_disj\_conj} \quad : \quad \frac{\{p\}s\{q\}, \{c\}s\{d\}}{\{p \wedge c\}s\{q \wedge d\}, \{p \vee c\}s\{q \vee d\}}$$

Décrivons tout d'abord la preuve "manuelle", dans laquelle nous ne détaillons pas la preuve du *unless* (cf. chapitre 3) :

(1) *ensures(c,d,pgm) : il existe une action s dans pgm telle que* $\{c \wedge \neg\ d\}\ s\ \{d\}$, *nous choisissons de l'appeler "skact", et pour toute action s de pgm, on a* $\{c \wedge \neg\ d\}\ s\ \{c \vee d\}$

(2) *D'après (1), l'action skact vérifie,* $\{c \wedge \neg\ d\}$ *skact* $\{d\}$

(3) *unless(p,q,pgm) : pour toute action de pgm, on a* $\{p \wedge \neg\ q\}\ s\ \{p \vee q\}$

(4) *D'après (3), on a* $\{p \wedge \neg\ q\}$ *skact* $\{p \vee q\}$

(5) *En appliquant wp_ disj_ conj à (2) et (4), on a,* $\{(p \wedge \neg\ q) \wedge (c \wedge \neg\ d)\}$ *skact* $\{(p \vee q) \wedge d\}$

(6) $((p \vee q) \wedge d) \Rightarrow ((p \wedge d) \vee (c \wedge q) \vee (q \wedge d))$

(7) $((p \wedge c) \wedge \neg((p \wedge d) \vee (c \wedge q) \vee (q \wedge d))) \Rightarrow ((p \wedge \neg q) \wedge (c \wedge \neg d))$

(8) *En appliquant wp_ impl à (7), (6) et (8) on a,*
$\{(p \wedge c) \wedge \neg((p \wedge d) \vee (c \wedge q) \vee (q \wedge d))\}$*skact*$\{(p \wedge d) \vee (c \wedge q) \vee (q \wedge d)\}$

(9) *Enfin, en simplifiant (8), on a le résultat cherché :*
$\{p \wedge c\}$ *skact* $\{(p \wedge d) \vee (c \wedge q) \vee (q \wedge d)\}$

Étudions maintenant la preuve mécanisée : pour effectuer les étapes utilisant *wp*, nous avons besoin de déplier la définition de *unless(p,q,pgm)*, afin d'obtenir *unless(p,q,a1)* $\wedge$ *unless(p,q,pgm2)* (en écrivant *pgm* comme étant *a1* \u *pgm2*). Nous avons alors deux cas : soit l'action vérifiant (1), que nous appelée *skact*, est *a1*, ou bien elle appartient à *pgm2*. Pour le premier cas, la preuve mécanisée utilise les mêmes étapes que celles de la preuve manuelle. Pour le deuxième cas, nous allons utiliser l'induction, puisque la définition de *unless* est inductive. En effet, l'étape (1) s'écrit (où $\rightarrow$ se lit "se réécrit") :

```
\E a2 (a2 \in {a1, pgm2}
     /\ {c /\ ~d} a2 {d}) /\ unless(c /\ ~ d,c \/ d,insert(a1,pgm2))
→ \E a2 (((a2 = a1) \/ (a2 \in pgm2)) /\ {c /\ ~ d} a2 {d})
      /\ unless(c /\ ~ d,c \/ d,insert(a1,pgm2))
```

L'étape (3) consiste à skolémiser la première partie de la formule précédente, pour obtenir deux faits, (31) et (32) :

```
((skact = a1) \/ (skact \in pgm2)) /\ {c /\ ~ d} skact {d})          (3)
→ ((skact = a1) \/ (skact \in pgm2))                                 (31)
   {c /\ ~ d} skact {d})                                             (32)
```

Le but à montrer est :

```
\E a2 ((a2 = a1) \/ (a2 \in pgm2))
   /\ {p /\ c}a2{(p /\ d) \/ (c /\ q) \/ (q /\ d)}
```

▶ Dans le cas où `(skact = a1)`, ce qui correspond à l'étape (4), nous utilisons le "dépliage" de la définition de *unless*:

```
 unless(p,q,a1) /\ unless(p,q,pgm2)
→ {p  /\ ~ q} a1 {p \/ q}/\ unless(p,q,pgm2)                    (4)
→ {p  /\ ~ q} skact {p \/ q}/\ unless(p,q,pgm2)
→ {p  /\ ~ q} skact {p \/ q}                                    (41)
   unless(p,q,pgm2)                                             (42)
```

Nous appliquons alors *wp_disj_conj* à (32) et (41), afin de déduire (5). Puis nous poursuivons la preuve comme la preuve manuelle, où nous spécialisons la variable quantifiée du but à `skact` (ou bien à `a1`).

▶ Dans le cas où `skact \in pgm2`, la formule (4) est inchangée:

```
{p  /\ ~ q} a1 {p \/ q} /\ unless(p,q,pgm2)                     (4)
```

et la formule (31) est simplifiée:

```
(skact \in pgm2)                                                (31)
{c /\ ~ d} skact {d})                                           (32)
```

Pour appliquer *wp_disj_conj*, il nous faut la propriété

```
{p  /\ ~ q} skact {p \/ q}
```

que l'on ne peut déduire de (4), car $\neg$`(skact = a1)`.

Conclusion: il nous faut appliquer l'induction sur le programme, car l'action `skact` appartient au reste du programme, c'est à dire `pgm2`.

Afin d'illustrer le déroulement de la preuve, donnons les étapes de la preuve telle que LP la détaille dans son script:

```
Set name E_conjonc
prove
   (unless(p,q,pgm) /\ ensures(c,d,pgm))
    => (ensures((p ^ c), (p ^ d) \or (c ^ q) \or (q^ d),pgm))
by induction on pgm using pgm_induct
..

Creating subgoals for proof by structural induction on 'pgm'
Basis subgoal:
 Subgoal 1: unless(p,q,{}) /\  ensures(c,d,{})
        => ensures(p ^ c,(c ^ q) \or (p ^ d) \or (q ^ d),{})
Induction constant: pgmc
Induction hypothesis:
 E_conjoncInductHyp.1:
   unless(p,q,pgmc) /\  ensures(c,d,pgmc)
    => ensures(p ^ c,(c ^ q) \or (p ^ d) \or (q ^ d),pgmc)
Induction subgoal:
 Subgoal 2: unless(p,q,insert(a1,pgmc)) /\ ensures(c,d,insert(a1,pgmc))
          => ensures(p ^ c,(c ^ q) \or (p ^ d) \or (q ^ d),insert(a1,pgmc))
```

Le cas de base est simplifié à `true`, car l'antécédent de l'implication se réduit à `false`.

Pour l'étape d'induction, nous procédons par implication, puis nous montrons le but *unless* en instanciant la règle `U_conjonc` avec les termes adéquats. L'état courant comprend les hypothèses suivantes:

```
E_conjoncImpliesHyp.1.1: nnot(cc) \or dc  \or wp(a1c,cc \or dc) -> T
E_conjoncImpliesHyp.1.2: nnot(pc) \or qc \or wp(a1c,pc \or qc) -> T
E_conjoncImpliesHyp.1.3: unless(cc,dc,pgmc) -> true
E_conjoncImpliesHyp.1.4: unless(pc,qc,pgmc) -> true
E_conjoncImpliesHyp.1.5: \E a2 ((a2 = a1c \/  a2 \in pgmc)
                                /\  dc \or nnot(cc) \or wp(a2,dc) = T)
                         -> true
```

et le but:

```
\E a2 (a2 = a1c \/  a2 \in pgmc)
    /\  (dc \or nnot(cc) \or nnot(pc) \or qc
      \or wp(a2,(cc \or dc) ^  (qc \or dc) ^  (pc \or qc)))
      = T
```

L'étape (2) de la preuve manuelle correspond à la skolémisation de la formule de l'hypothèse 5, c'est à dire de la prémisse de la règle d'inférence à prouver. Par conséquent, l'idée est de montrer que si on "pose" $s$ comme étant l'action `a1c` alors la preuve consiste à utiliser les mêmes étapes de déduction que celles utilisées dans la preuve manuelle. Sinon, l'action $s$, vérifiant la prémisse de la règle, se trouve dans le reste du programme, `pgmc`, auquel cas on utilise l'hypothèse d'induction.

La première étape consiste à éliminer le quantificateur existentiel de l'hypothèse 5. Pour cela, nous choisissons une constante de skolem, `skact` de type `Act`:

```
declare op skact:-> Act
fix a2 as skact in *hyp
```

Nous obtenons alors les faits (31) et (32) de la preuve manuelle (le but et les autres hypothèses sont inchangés):

```
E_conjonc.2.1: dc \or nnot(cc) \or wp(skact,dc) -> T
E_conjonc.2.2: skact = a1c \/ skact \in pgmc -> true
```

Conformément à la preuve informelle, nous allons procéder par cas, suivant que `skact = a1c` ou bien `skact \in pgmc`. Cette preuve par cas comprend donc trois sous-but, dont le premier consiste à justifier les deux autres cas (cf. chapitre 1, p. 22). Cette justification est triviale puisque'elle est fournie par le fait `E_conjonc.2.2`.

Le but est le même pour les deux cas:

```
\E a2 (a2 = a1c \/  a2 \in pgmc)
    /\  (dc \or nnot(cc) \or nnot(pc) \or qc
      \or wp(a2,(cc \or dc) ^  (qc \or dc) ^  (pc \or qc)))
      = T
```

▶ Cas `skact = a1c`: il suffit de spécialiser la variable quantifiée du but à `skact`, avec la commande

```
resume by specializing a2 to skact
```

nous obtenons le sous-but suivant :

```
dc \or nnot(cc) \or nnot(pc) \or qc
 \or wp(skact, (cc \or dc) ^  (qc \or dc) ^  (pc \or qc))
 = T
```

résolu suivant les mêmes étapes que celles utilisées dans la preuve informelle, en appliquant les deux règles `wp_disj_conj`, et `wp_impl`, aux faits `E_conjoncImpliesHyp.1.2` et `E_conjonc.2.2`.

▶ Cas `skact \in pgmc` : intuitivement, nous savons qu'il faut utiliser l'hypothèse d'induction. Or celle ci est sous la forme d'une implication dont l'antécédent et la conclusion contiennent un quantificateur existentiel :

```
E_conjoncInductHyp.1:
 unless(c,d,pgmc) /\  unless(p,q,pgmc)
  /\  \E a1 (a1 \in pgmc
          /\  nnot(c) \or wp(a1,d) \or d = T)
 => unless(p ^ c,(c \or d) ^ (q \or d) ^ (p \or q),pgmc)
    /\ \E a1 (a1 \in pgmc
           /\ (nnot(c) \or nnot(p) \or d \or q
           \or wp(a1,(c \or d) ^ (q \or d) ^ (p \or q)) = T))
-> true
```

alors que le but est:

```
  \E a2 (a2 = a1c \/  a2 \in pgmc)
         /\ (nnot(cc) \or nnot(pc) \or dc \or qc
        \or wp(a2,(cc \or dc) ^  (qc \or dc) ^ (pc \or qc)) = T)
```

Mis à part les quantificateurs existentiels, la conclusion de l'hypothèse d'induction est bien le but recherché. Il s'agit donc de "montrer" que les antécédents de l'implication sont vérifiés afin de déduire la conclusion de l'implication. Nous commençons par instancier correctement les variables libres (p par pc, q par qc,...) pour obtenir, après simplification, l'hypothèse d'induction suivante :

```
  \E a1 (a1 \in pgmc /\  dc \or nnot(cc) \or wp(a1,dc) = T)
    => \E a1 a1 \in pgmc
         /\  (dc \or nnot(cc) \or nnot(pc) \or qc
            \or wp(a1,(cc \or dc) ^  (qc \or dc) ^ (pc \or qc)) = T)
```

Comme nous avons le fait :

```
E_conjonc.2.1: dc \or nnot(cc) \or wp(skact,dc) -> T
```

C'est à dire que nous "savons" qu'il existe une action qui vérifie l'antécédent de l'implication, il nous suffit de le montrer :

```
  prove \E a1 (a1 \in pgmc /\  dc \or nnot(cc) \or wp(a1,dc) = T)
     res by specializing a1 to skact
      <>  specialization subgoal
      [] specialization subgoal
     [] conjecture
```

Ce qui nous permet d'inférer la conclusion de l'implication :

```
E_conjoncInductHyp.1.1:
 \E a1 (a1 \in pgmc)
    /\ (dc \or nnot(cc) \or nnot(pc) \or qc
       \or wp(a1, (cc \or dc) ^  (qc \or dc) ^  (pc \or qc)) = T)
 -> true
```

Pour conclure, il suffit de choisir un nom pour la variable `a1`, par exemple `skact2`, c'est à dire skolémiser la formule précédente. Puis, spécialiser la variable quantifiée du but à la même constante `skact2` nous permet de conclure :

```
    declare op skact2:-> Act
    fix a1 as skact2 in *InductHyp.1.1!          %la conclusion de l'hyp
    resume by spec a2 to skact2
□
```

En résumé, pour utiliser une hypothèse qui comprend deux quantificateurs existentiels, nous avons utilisé la stratégie suivante :
Ayant les hypothèses

$H1 : \exists v Q(v) \Rightarrow \exists r P(r)$
$H2 : Q(cst)$

et le but

$\exists x P(x)$

Nous commençons par montrer la propriété $\exists v Q(v)$ en spécialisant la variable $v$ à la constante $cst$. Nous obtenons alors le fait $\exists r P(r)$, pour lequel nous choisissons une constante de skolem, $skr$, pour nommer la variable $r$. Nous obtenons le fait $P(skr)$. Pour conclure, il suffit de spécialiser la variable $x$ du but à $skr$. La preuve correspondante est :

```
prove \E x P(x)
  set name lemme
  prove \E v Q(v)
    resume by specializing v to cst
  [] conjecture
  declare operator skr: -> S
  fix r as skr in Hyp
  resume by specializing x to skr
[] conjecture
qed
```

Bien que cette gestion des quantificateurs semble délicate, la démarche utilisée correspond au raisonnement effectué pour les preuves manuelles. En effet, ces preuves mécanisées doivent être considérées comme une validation du raisonnement mis en place pour la preuve manuelle. Il ne s'agit pas de "forcer" la preuve à réussir.

Voici un exemple où on introduit des axiomes arbitraires non justifiés dans le but de "forcer" la preuve à réussir :

Reprenons la preuve précédente, où le but à montrer est :

```
  \E a2 (a2 = a1c \/  a2 \in pgmc)
  /\ (dc \or nnot(cc) \or nnot(pc) \or qc
    \or wp(a2,(cc \or dc) ^ (qc \or dc) ^  (pc \or qc))
    = T)
```

Les hypothèses sont les suivantes après avoir skolémiser l'hypothèse (5) :

```
E_conjoncImpliesHyp.1.2: nnot(pc) \or qc \or wp(a1c,pc \or qc) -> T
E_conjonc.2.1: dc \or nnot(cc) \or wp(skact,dc) -> T
E_conjonc.2.2: skact = a1c \/ skact \in pgmc -> true
```

Nous poursuivons en spécialisant la variable `a2` à `a1c`, et ceci dans le but d'éliminer le quantificateur existentiel. Nous obtenons le but suivant :

```
dc \or nnot(cc) \or nnot(pc) \or qc
   \or wp(a1c,(cc \or dc) ^ (qc \or dc) ^ (pc \or qc))
= T
```

Pour forcer la preuve à réussir, nous "décidons" que `skact` *est* `a1c` en introduisant le fait `skact = a1c` et en utilisant `wp_disj_conj` pour obtenir :

```
wp_disj_conj.1.1.1:
   dc \or nnot(cc) \or nnot(pc) \or qc \or wp(a1c,(pc \or qc) ^ dc) = T
```

puis nous utilisons comme précédemment la règle `wp_impl` avec les mêmes termes, pour terminer la preuve.

L'erreur provient de l'axiome

```
assert skact = a1c
```

Sémantiquement, nous avons montré le théorème en choisissant l'action qui vérifie la propriété comme étant l'action de l'induction `a1c`, ce qui est évidement incorrect. D'après la preuve manuelle, comme $\exists s \in pgm : \{c \wedge \neg d\}s\{d\}$, nous avons affirmé que $s = first(pgm)$, ce qui est une étape inexistante dans la preuve manuelle.

De façon générale, l'utilisation des axiomes injustifiés dans les preuves annule leur validité. En effet, si nous introduisons des affirmations ("gratuites"), nous n'avons aucune garantie d'un résultat correct. Par conséquent, l'introduction des axiomes dans les preuves doit se faire de façon temporaire. En effet, il se peut que l'on ait besoin, lors de la vérification d'une preuve, d'une propriété qui n'existe pas encore dans notre base de faits. La stratégie consiste à l'introduire comme un axiome, puis de poursuivre la preuve. Une fois la preuve achevée, il faut reprendre les lemmes introduits et les prouver séparément. Une autre stratégie consiste à sauvegarder le statut courant de la preuve, à prouver les lemmes dont on a besoin puis de reprendre la preuve après avoir eu confirmation de la validité des lemmes utilisés. En conclusion, le démonstrateur nous aide à construire notre preuve et à la mettre au point et non pas à effectuer des preuves incorrectes.

## 4.4 Leads_to

La deuxième amélioration, et non la moindre, est une nouvelle formalisation de l'opérateur *leads_to*, utilisant le quantificateur existentiel, et les trois règles d'inférence définissant cette relation de *progrès*.

Un programme *pgm* vérifie la propriété *p leads_to q* si et seulement si cette propriété peut être dérivée en appliquant un nombre fini de fois les règles d'inférence suivantes :

$$LEADS\_TO : \quad \frac{p \ ensures \ q}{p \ leads\_to \ q}$$

$$TRANS\_LEAD : \quad \frac{p \ leads\_to \ r, r \ leads\_to \ q}{p \ leads\_to \ q}$$

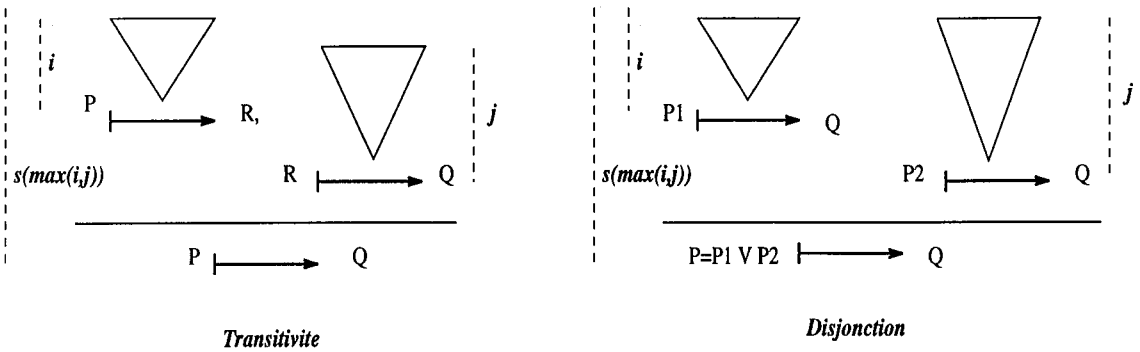



FIG. 4.1 – *Schématisation de l'axiome leads_to2*

$$DISJ\_LEAD : \frac{\forall m : p(m)\ leads\_to\ q}{\exists m : p(m)\ leads\_to\ q}$$

La disjonction est formalisée comme une disjonction finie dans notre approche :

$$DISJ\_FINIE : \frac{p_1\ leads\_to\ q, p_2\ leads\_to\ q}{p_1 \vee p_2\ leads\_to\ q}$$

Du point de vue opérationnel, un programme a la propriété $p$ *leads_to* $q$ si et seulement si dès que le prédicat $p$ est valide alors le prédicat $q$ sera tôt ou tard valide. Comme nous l'avons vu au chapitre 3, la preuve d'une propriété *leads_to* consiste à raisonner par induction sur la profondeur des dérivations.

L'idée est de simuler cette profondeur par un troisième paramètre de type entier : `leads_to(p,q,pgm,n)`. Ce terme doit être interprété comme le fait que la propriété `leads_to` a été obtenue en $n$ pas de dérivation.

La définition de la relation *leads_to* est alors la suivante :

```
set name leads_to2
assert
  ensures(p,q,pgm)
  \/ \E c \E d \E i \E j
        ((s(max(i,j)) = n)
           /\ p = c \or d
           /\ leads_to2(c,q,pgm,i)
           /\ leads_to2(d,q,pgm,j))
  \/ \E r \E i \E j
        (max(i,j) < n
           /\ leads_to2(p,r,pgm,i)
           /\ leads_to2(r,q,pgm,j))
  = leads_to2(p,q,pgm,n)
..
```

Cet axiome exprime les propriétés suivantes (Figure 4.1) :

- Si la propriété $p$ *ensures* $q$ est valide dans le système courant alors nous pouvons en déduire $p$ *leads_to* $q$, avec $n$ quelconque car $p$ *ensures* $q$ peut être valide à n'importe quel moment du calcul.

- Si la propriété $p$ *leads_to* $r$ a été obtenue à un pas $i$ et la propriété $r$ *leads_to* $q$ a été obtenue au pas $j$ alors nous pouvons en déduire $p$ *leads_to* $q$, à un pas $n$, $n$ étant le successeur du maximum de $i$ et de $j$.

- Si la propriété $c$ *leads_to* $q$ a été obtenue à un pas $i$ et la propriété $d$ *leads_to* $q$ a été obtenue au pas $j$ alors nous pouvons en déduire $c \vee d$ *leads_to* $q$, à un pas $n$, $n$ étant le successeur du maximum de $i$ et de $j$.

De plus, cette règle induit que la propriété $p$ *leads_to* $q$ ne peut être déduite qu'en appliquant ces trois règles d'inférence.

Nous allons à présent essayer de prouver les théorèmes que doit vérifier la relation *leads_to*, précédemment posés comme axiomes.

### 4.4.1 Impossibilité

Une des propriétés à vérifier est le théorème de l'*impossibilité*, précédemment posé comme axiome :

$$IMPO\_LEAD : \frac{p \ \ leads\_to \ \ false}{\neg p}$$

Nous rappelons la preuve "manuelle" : la preuve est par induction sur le nombre de règles d'inférence appliquées pour inférer $p \mapsto false$ (cf. Figure3.3).

**Cas de base:** La longueur de la preuve est 1, on a inféré $p \mapsto false$ avec l'application d'une seule règle d'inférence, en l'occurrence la règle *LEADS_TO*. Le but est déduit en utilisant le théorème correspondant pour *ensures*.

**Hypothèse d'induction:** La propriété est vraie si le nombre de règles utilisées est inférieur ou égal à $n$.

**Pas d'induction:** Le nombre de règles appliquées pour inférer $p \mapsto false$ est $n + 1$. A l'étape $n$, nous avons donc appliqué la règle de transitivité ou bien celle de la disjonction.
**Transitivité :**

| | |
|---|---|
| $r \mapsto false$ | (1) *Hyp*1 |
| $\neg r$ | (2) *Hyp d'induction* sur (1) |
| $p \mapsto r$ | (3) *Hyp*2 |
| $p \mapsto false$ | (5) *avec* (3) et (2) |
| $\neg p$ | *Hyp d'induction sur* (5) |

**Disjonction :**

| | |
|---|---|
| $p_1 \mapsto false$ | (1) *Hyp*1 |
| $\neg p_1$ | (2) *Hyp d'induction sur* (1) |
| $p_2 \mapsto false$ | (3) *Hyp*2 |
| $\neg p_2$ | (4) *Hyp d'induction sur* (2) |
| $p \equiv p_1 \vee p_2$ | (5) *Hyp*3 |
| $\neg p \equiv \neg p_1 \wedge \neg p_2$ | (7) *CP*1 *avec* (5) |
| $\neg p$ | *CP*1 *avec* (2),(4) |

Étudions maintenant la preuve mécanisée :

```
set name impo_lead
prove leads_to2(p,F,insert(a1,pgm),n) => (nnot(p)=T)
```

Nous devons donc utiliser l'induction sur n pour pouvoir effectuer les dérivations de $\neg p$ pour la transitivité et de $\neg p_1$ et $\neg p_2$ pour la disjonction. Mais quel schéma d'induction utiliser?

En effet, si nous utilisons le schéma "classique" :

```
sort Nat generated freely by 0,s:Nat-> Nat
```

Cela nous donne la structure de preuve suivante :

```
Creating subgoals for proof by structural induction on 'n'
Basis subgoal:
  Subgoal 1: leads_to2(p,F,insert(a1,pgm),0) => nnot(p) = T
Induction constant: nc
Induction hypothesis:
  impo_leadInductHyp.1: leads_to2(p,F,insert(a1,pgm),nc) => nnot(p) = T
Induction subgoal:
  Subgoal 2: leads_to2(p,F,insert(a1,pgm),s(nc)) => nnot(p) = T
```

Le cas de base est prouvé à l'aide du lemme sur le résultat correspondant pour *ensures* :

```
resume by =>-m
  <> => subgoal
  instantiate p by pc,q by F,pgm by insert(a1c,pgmc),n by 0 in leads_to2
  instantiate p by pc,pgm by insert(a1c,pgmc) in E_impo
  [] => subgoal
[] basis subgoal
```

Examinons le pas d'induction : le but à prouver est `nnot(p) = T` ayant comme hypothèses :

```
impo_leadImpliesHyp.1: leads_to2(pc,F,insert(a1c,pgmc),s(nc)) -> true
impo_leadInductHyp.1:leads_to2(p,F,insert(a1,pgm),nc)=> nnot(p)= T-> true
```

Il nous faut alors utiliser l'axiome de définition `leads_to2` avec les termes appropriés :

```
instantiate p by pc,q by F,n by s(nc),pgm by insert(a1c,pgmc) in leads_to2
```

L'axiome instancié est :

```
leads_to2.1.1: ensures(pc,F,insert(a1c,pgmc))
               \/ \E c \E d \E i \E j
                    (pc = c \or d
                       /\ (s(max(i,j)) = s(nc))
                       /\ leads_to2(c,F,insert(a1c,pgmc),i)
                       /\ leads_to2(d,F,insert(a1c,pgmc),j))
               \/ \E r \E i \E j
                    ((s(max(i,j)) = s(nc))
                       /\ leads_to2(pc,r,insert(a1c,pgmc),i)
                       /\ leads_to2(r,F,insert(a1c,pgmc),j))
               -> true
```

Ce qui "simule" le fait qu'on ait obtenu `leads_to2(pc,F,insert(a1c,pgmc),s(nc))` par application d'une des trois règles d'inférence.

Nous procédons par cas, suivant la règle d'inférence appliquée, *ensures*, *disjonction* ou bien *transitivité*. Le premier cas de la preuve consiste à montrer que les trois cas sont "justifiés", c'est à dire que la liste de cas donnée couvre bien tous les cas possibles. Ce qui est trivial avec la règle `leads_to2.1.1`. Le deuxième cas correspond à *ensures* ce qu'on résout comme le cas de base de l'induction.

Soit maintenant le cas de la disjonction :

```
impo_leadCaseHyp.1.2: \E c \E d \E i \E j
                        (pc = c \or d
                           /\ (s(max(i,j)) = s(nc))
                           /\ leads_to2(c,F,insert(a1c,pgmc),i)
                           /\ leads_to2(d,F,insert(a1c,pgmc),j))
                       -> true
```

Nous devons choisir deux constantes de skolem, `ccst,dcst`, inférer `nnot(ccst)` et `nnot(dcst)` en utilisant l'induction, puis comme `nnot(pc)= nnot(cst)^nnot(dcst)`, nous en déduirons `nnot(pc) = T`.

```
declare op ccst:-> Bexp
fix c as ccst in *hyp              %on obtient impo_lead.2
declare op dcst:-> Bexp
fix d as dcst in impo_lead.2       %on obtient impo_lead.3
declare op k:-> Nat
fix i as k  in impo_lead.3         %on obtient impo_lead.4
declare op t:-> Nat
fix j  as t   in impo_lead.4       %on obtient impo_lead.5
```

Ce qui correspond à l'élimination des quatre quantificateurs existentiels de l'hypothèse `impo_leadCaseHyp.1.2`. Nous obtenons quatre faits correspondants aux quatre conjonctions :

```
impo_lead.5.1: s(max(k,t)) = s(nc)
impo_lead.5.2: pc = ccst \or dcst
impo_lead.5.3: leads_to2(ccst,F,insert(a1c,pgmc),k)
impo_lead.5.4: leads_to2(dcst,F,insert(a1c,pgmc),t)
```

D'après le fait `impo_lead.5.1`, `nc` est égal au maximum de `k` et de `t`. Supposons que `max(k,t) = t`. Nous en déduisons (en utilisant un lemme sur les entiers) que `k < nc` et `t = nc`. Pour inférer `nnot(ccst)` (étape (2) de la preuve manuelle), il faut appliquer l'induction, à savoir l'hypothèse :

```
impo_leadInductHyp.1:leads_to2(p,F,insert(a1,pgm),nc)=> nnot(p)= T-> true
```

Puis il faut procéder par cas, suivant que `k = nc` ou bien `k < nc`. Dans le cas ou `k=nc`, la preuve se poursuit comme la preuve manuelle. Mais quand est-il lorsque `k < nc`? Nous ne pouvons appliquer l'hypothèse d'induction. Par conséquent le schéma d'induction utilisé n'est pas adéquat, car il ne peut être utilisé que dans le cas ou la transitivité (ou bien la disjonction) a été appliquée au pas $n$.

Nous disposons d'un deuxième schéma d'induction sur les entiers (cf. paragraphe 3.2) :

```
Nat.4: well founded < :Nat,Nat -> Bool
```

Ce schéma définit $<$ comme étant un ordre bien fondé (*nothérien*), logiquement équivalente à à l'ensemble infini de formules

$$\forall x(\forall y((y < x) \Rightarrow P(y)) \Rightarrow P(x)) \Rightarrow \forall x P(x)$$

Prouver la formule `P(x)` à l'aide de cette règle d'induction consiste à montrer `P(xc)` ayant l'hypothèse d'induction `x < xc => P(x)`. Ce schéma correspond bien aux preuves que nous nous proposons d'effectuer : on suppose la propriété vraie pour un nombre de dérivations inférieur à $n$ et on montre la propriété pour la dérivation suivante.

Examinons alors l'utilisation de ce schéma d'induction, pour la preuve du théorème d'*impossibilité* :

```
set name impo_lead
prove leads_to2(p,F,insert(a1,pgm),n) => (nnot(p)=T) by ind using nat.4
```

Nous obtenons le schéma suivant :

```
Creating subgoals for proof by well founded induction on 'n'
Hypothesis:
  impo_leadInductHyp.1:
     n < nc => (leads_to2(p,F,insert(a1,pgm),n) => nnot(p) = T)
Subgoal:
  leads_to2(p,F,insert(a1,pgm),nc) => nnot(p) = T
```

Nous poursuivons par implication, pour obtenir l'hypothèse :

```
impo_leadImpliesHyp.1: leads_to2(pc,F,insert(a1c,pgmc),nc) -> true
```

et le but : `nnot(pc) = T`.

Nous utilisons alors les mêmes étapes que précédemment, à savoir l'axiome `leads_to2` pour déduire les trois possibilités puis une preuve par cas, où le cas *ensures* est résolu comme précédemment.

▶ Étudions le deuxième cas, correspondant à la disjonction. Nous avons trois hypothèses :

- l'hypothèse que la propriété $c \mapsto false$ (respectivement $d \mapsto false$) a été inférée à l'étape `i` (respectivement `j`), pour $i$ et $j$ vérifiant $s(max(i,j)) = nc$ et $pc = c \vee d$.
- l'hypothèse que la propriété $pc \mapsto false$ a été inférée à l'étape `nc`.
- et enfin que la propriété d'impossibilité est valide pour tout entier naturel inférieur à `nc`.

```
Rewrite rules:
impo_leadCaseHyp.1.2: \E c \E d \E i \E j
                        ((s(max(i,j)) = nc)
                           /\ pc = c \or d
                           /\ leads_to2(c,F,insert(a1c,pgmc),i)
                           /\ leads_to2(d,F,insert(a1c,pgmc),j))
                       -> true
impo_leadImpliesHyp.1: leads_to2(pc,F,insert(a1c,pgmc),nc) -> true
impo_leadInductHyp.1: n < nc
                       => (leads_to2(p,F,insert(a1,pgm),n) => nnot(p) = T)
                       -> true
```

Nous éliminons les quantificateurs existentiels de l'hypothèse, en utilisant comme précédemment, quatre constantes de skolem `ccst`, `dcst`, `k` et `t`, et nous obtenons les faits suivants :

```
impo_lead.5.1: s(max(k,t)) = nc -> true
impo_lead.5.2: pc -> ccst \or dcst
impo_lead.5.3: leads_to2(ccst,F,insert(a1c,pgmc),k) -> true
impo_lead.5.4: leads_to2(dcst,F,insert(a1c,pgmc),t) -> true
```

en utilisant un lemme sur les entiers, nous obtenons les faits :

```
lemme.1.1.1: k < nc
lemme.1.1.2: t < nc
```

Nous pouvons alors "utiliser" l'hypothèse d'induction , en calculant ses paires critiques avec les quatre faits obtenus, qui correspond aux étapes (2) et (4) de la preuve manuelle :

```
set name critpair
crit-p impo_lead.5.1:5 with *InductHyp
```

Les paires critiques non triviales sont :

```
critpair.1: nnot(ccst) = T
critpair.2: nnot(dcst) = T
```

utilisées par LP pour réécrire le but :

```
  nnot(pc) = T
→ nnot(ccst \or dcst) = T
→ nnot(ccst) ^ nnot(dcst) = T
→ nnot(ccst) = T /\ nnot(dcst) = T
→ true
```

□

▶ Examinons à présent le dernier cas, c'est à dire la transitivité, les hypothèses sont:

```
Rewrite rules:
impo_leadCaseHyp.2.3: \E r \E i \E j
                        ((s(max(i,j)) = nc)
                           /\ leads_to2(pc,r,insert(a1c,pgmc),i)
                           /\ leads_to2(r,F,insert(a1c,pgmc),j))
                      -> true
impo_leadImpliesHyp.1: leads_to2(pc,F,insert(a1c,pgmc),nc) -> true
impo_leadInductHyp.1: n < nc
                      => (leads_to2(p,F,insert(a1,pgm),n) => nnot(p) = T)
                      -> true
```

Nous éliminons comme précédemment les trois quantificateurs existentiels en choisissant trois constantes de skolem, `rcst`, `k` et `t`. En utilisant le même lemme sur les entiers naturels, nous obtenons les faits suivants:

```
impo_lead.4.1: s(max(k,t)) = nc
impo_lead.4.2: leads_to2(pc,rcst,insert(a1c,pgmc),k)
impo_lead.4.3: leads_to2(rcst,F,insert(a1c,pgmc),t)
lemme.1.1.1: k < nc
lemme.1.1.2: t < nc
```

Nous appliquons l'hypothèse d'induction avec le fait `lemme.1.1.2` pour déduire `nnot(rcst) = T` (étape (2) de la preuve manuelle). Nous devons alors déduire que `rcst = F`, prouvé comme un lemme trivial avec la base définissant les expressions de prédicats `Bool_str` (cf. chapitre 2, paragraphe 2.3). Enfin, en utilisant l'hypothèse d'induction et le fait `impo_lead.4.2`, nous déduisons le résultat cherché.
□

### 4.4.2 Disjonction

Une autre propriété que la relation doit satisfaire est la disjonction "générale" :

$$DISJ\_leads : \frac{p \; leads\_to \; q, b \; leads\_to \; r}{p \vee b \; leads\_to \; q \vee r}$$

dont la preuve est :

| | |
|---|---|
| $q \Rightarrow q \vee r$ | (1) $CP1$ |
| $q \mapsto q \vee r$ | (2) $IMPL\_LEAD$ |
| $p \mapsto q$ | (3) $Hyp1$ |
| $p \mapsto q \vee r$ | (4) $TRANS\_LEAD$(2,3) |
| $r \Rightarrow q \vee r$ | (5) $CP1$ |
| $r \mapsto q \vee r$ | (6) $IMPL\_LEAD$ |
| $b \mapsto r$ | (7) $Hyp2$ |
| $b \mapsto q \vee r$ | (8) $TRANS\_LEAD$(6,7) |
| $p \vee b \mapsto q \vee r$ | $DISJ\_finie$(4,8) |

La preuve ne nécessite donc pas d'induction, elle utilise la propriété suivante (dont la preuve est triviale, car elle est vérifiée par la relation *Ensures*) :

$$IMPL\_leads : \frac{p \Rightarrow q}{p \; leads\_to \; q}$$

La difficulté de la preuve mécanisée réside dans la gestion des indices de l'opérateur `leads_to2`. Le but de la preuve est de montrer qu'il existe $m$ tel que $max(i,j) < m$ et tel que $leads\_to2(p \vee b, q \vee r, pgm, m)$, sachant $leads\_to2(p, q, pgm, i)$ et $leads\_to2(b, r, pgm, j)$. Nous choisissons alors une valeur précise pour $m$, $max(s(i), s(j)) + 1$ :

```
set name disj_leads
prove
  (leads_to2(p,q,insert(a1,pgm),i) /\ leads_to2(b,r,insert(a1,pgm),j))
     => leads_to2(p \or b,q \or r,insert(a1,pgm),max(s(i),s(j))+1)
  ..
```

En procédant par implication, nous obtenons les hypothèses :

```
disj_leadsImpliesHyp.1.1: leads_to2(bc,rc,insert(a1c,pgmc),jc) -> true
disj_leadsImpliesHyp.1.2: leads_to2(pc,qc,insert(a1c,pgmc),ic) -> true
```

et le but :

```
leads_to2(pc \or bc,qc \or rc,insert(a1c,pgmc),s(max(s(ic),s(jc))))
```

En instanciant correctement les variables par les termes adéquats dans le théorème `impl_lead`, nous obtenons les propriétés (2) et (6) :

```
impl_lead.1.1: leads_to2(qc,qc \or rc,insert(a1c,pgmc),n) -> true
impl_lead.1.2: leads_to2(rc,qc \or rc,insert(a1c,pgmc),n) -> true
```

que nous devons utiliser avec les deux hypothèses de l'implication pour appliquer la transitivité afin d'obtenir les propriétés (4) et (8). Appliquer la transitivité revient à montrer les deux lemmes suivants :

```
set name lemme1
prove  leads_to2(pc,qc \or rc,insert(a1c,pgmc),s(ic))
```

et

```
set name lemme2
prove  leads_to2(bc,qc \or rc,insert(a1c,pgmc),s(jc))
```

Le premier correspond à la propriété (4) et le second à la propriété (8).

• Étudions la preuve du premier lemme : nous commençons par instancier l'axiome de définition `leads_to2.1`, afin de déduire la règle suivante :

```
leads_to2.1.1: ensures(pc,qc \or rc,insert(a1c,pgmc))
               \/ \E c \E d \E i \E j
                    (pc = c \or d
                       /\ (s(max(i,j)) = s(ic))
                       /\ leads_to2(c,qc \or rc,insert(a1c,pgmc),i)
                       /\ leads_to2(d,qc \or rc,insert(a1c,pgmc),j))
               \/ \E r \E i \E j
                    ((s(max(i,j)) = s(ic))
                       /\ leads_to2(pc,r,insert(a1c,pgmc),i)
                       /\ leads_to2(r,qc \or rc,insert(a1c,pgmc),j))
               -> leads_to2(pc,qc \or rc,insert(a1c,pgmc),s(ic))
```

Pour déduire le résultat, c'est à dire la partie droite de la règle, nous utilisons la stratégie explicitée au paragraphe précédent afin d'utiliser une hypothèse qui comprend une quantificateur existentiel (cf. p. 129). Il s'agit de montrer le but suivant :

```
set name lemme1_1
 prove
 \E r \E i \E j
    ((s(max(i,j)) = s(ic))
     /\ leads_to2(pc,r,insert(a1c,pgmc),i)
     /\ leads_to2(r,qc \or rc,insert(a1c,pgmc),j))
 ..
```

Pour prouver ce lemme, nous devons exhiber un prédicat `r` et deux entiers `i` et `j` vérifiant la propriété. En utilisant les hypothèses de l'implication `disj_leadsImpliesHyp` et les faits `impl_lead`, nous spécialisons `r` à `qc`, `i` à `ic`. Nous obtenons le sous-but :

```
\E j ((s(max(ic,j)) = s(ic))
     /\ leads_to2(qc,qc \or rc,insert(a1c,pgmc),j))
```

Ayant la règle :

```
impl_lead.1.1: leads_to2(qc,qc \or rc,insert(a1c,pgmc),n) -> true
```

Il suffit de choisir une valeur pour `j`, puisque la propriété est vrai pour tout `n`. Nous choisissons `ic`, afin de terminer la preuve du lemme `lemme1_1`. Celui ci nous permet de déduire le membre droit de la règle `leads_to2.1.1` et par conséquent d'achever la preuve du lemme `lemme1`.

• Nous procédons de façon similaire pour le lemme `lemme2`, pour prouver

```
leads_to2(bc,qc \or rc,insert(a1c,pgmc),s(jc))
```

Nous spécialisons `r` à `rc`, `i` à `jc` et `j` à `jc` dans le lemme correspondant `lemme2_1`.
□

Nous avons alors les propriétés (4) et (8) :

```
lemme1.1: leads_to2(pc,qc \or rc,insert(a1c,pgmc),s(ic)) -> true
lemme2.1: leads_to2(bc,qc \or rc,insert(a1c,pgmc),s(jc)) -> true
```

et nous devons montrer le but :

```
leads_to2(pc \or bc,qc \or rc,insert(a1c,pgmc),s(s(s(jc + ic))))
```

D'après la preuve manuelle, nous savons que pour conclure, il nous faut appliquer la règle d'inférence de la *disjonction* aux deux lemmes. "Appliquer" la disjonction consiste à utiliser l'axiome de définition, en instanciant correctement les variables libres :

```
leads_to2.2.1: ensures(pc\or bc,qc \or rc,insert(a1c,pgmc))
               \/ \E c \E d \E i \E j
                     (pc \or bc = c \or d
                        /\ (s(max(i,j)) = s(max(s(ic),s(jc))))
                        /\ leads_to2(c,qc \or rc,insert(a1c,pgmc),i)
                        /\ leads_to2(d,qc \or rc,insert(a1c,pgmc),j))
               \/ \E r \E i \E j
                     ((s(max(i,j)) = s(max(s(ic),s(jc))))
                        /\ leads_to2(pc \or bc,r,insert(a1c,pgmc),i)
                        /\ leads_to2(r,qc \or rc,insert(a1c,pgmc),j))
             ->
           leads_to2(pc \or bc,qc \or rc,insert(a1c,pgmc),s(max(s(ic),s(jc))))
```

En comparant cette formule aux deux lemmes, nous connaissons la "valeur" des variables quantifiées `c`, `d`, `i` et `j`. La preuve consiste alors à "montrer" que les hypothèses de la *disjonction* sont vérifiées :

```
set name lemme3
prove
  \E c \E d \E i \E j
      (pc \or bc = c \or d
       /\ (max(i,j) < s(ic)+s(jc)+1)
       /\ leads_to2(c,qc \or rc,insert(a1c,pgmc),i)
       /\ leads_to2(d,qc \or rc,insert(a1c,pgmc),j))
  ..
```

afin de déduire le membre droit de la règle `leads_to2.2.1`, qui est le but recherché.

La preuve de lemme3 utilise le même principe que la preuve du `lemme1` et du `lemme2`. Elle consiste à spécialiser les variables quantifiées `c`, `d`, `i` et `j` à `pc`, `bc`, `s(ic)`, `s(jc)` respectivement.
□

### 4.4.3 Élimination

Le théorème suivant est la règle d'*élimination*:

$$CAN\_LEAD : \frac{p \ leads\_to \ (q \vee b), b \ leads\_to \ r}{b \ leads\_to \ r}$$

dont la preuve fait appel au théorème précédent:

| | |
|---|---|
| $q \Rightarrow q$ | (1) $CP1$ |
| $q \mapsto q$ | (2) $IMPL\_LEAD$ |
| $b \mapsto r$ | $Hyp2$ (3) |
| $q \vee b \mapsto q \vee r$ | (4) $DISJ\_leads$(2,3) |
| $p \Rightarrow q \vee b$ | (5) $Hyp1$ |
| $p \mapsto q \vee r$ | (6) $TRANS\_LEAD$(5,4) |

La preuve LP correspondante est la suivante, où comme pour la disjonction, nous choisissons une valeur pour l'indice:

```
Set name can_lead
prove
  (leads_to2(p,q \or b,insert(a1,pgm),i) /\ leads_to2(b,r,insert(a1,pgm),j))
   => (leads_to2(p,q \or r,insert(a1,pgm),max(i,s(s(j)))+1))
  ..
  res by =>m
    instantiate p by qc,q by qc,pgm by insert(a1c,pgmc)  in leads_to2   %(2)
    instantiate                                                         %(4)
      p by qc,r by rc,q by qc,b by bc,a1 by a1c,pgm by pgmc,
      j by jc,i by jc in disj_leads
    ..
    set name lemme1
    prove leads_to2(pc,qc \or rc,insert(a1c,pgmc),max(i,s(s(j)))+1)   %(6)
      ins
        p by pc,q by qc \or rc,n by max(i,s(s(j)))+1,
        pgm by insert(a1c,pgmc) in leads_to2
      ..
      set name lemme1_1
      prove
        \E r \E i \E j
           ((s(max(i,j)) = max(i,s(s(j)))+1)
              /\ leads_to2(pc,r,insert(a1c,pgmc),i)
              /\ leads_to2(r,qc \or rc,insert(a1c,pgmc),j))
        ..
        resume by spec r to qc \or bc
          resume by spec i to ic
            resume by spec j to s(s(jc))
qed
```

La première étape utilise la réflexivité de *ensures*, en instanciant dans l'axiome de définition `p` par `q`, ce qui est équivalent à l'utilisation de `impl_lead`. La deuxième étape consiste à utiliser la règle *DISJ_ leads* pour inférer $q \vee b \mapsto q \vee r$. Il suffit alors d'"appliquer" la transitivité:

- instancier les variables libres de l'axiome de définition `p`, `q`, et `n` par `pc`, `qc \or rc`, et `max(ic,s(s(jc)))+1` respectivement.

- prouver le membre gauche de l'axiome de définition pour déduire le membre droit, c'est à dire `leads_to2(p,q \or r,insert(a1,pgm),max(i,s(s(j)))+1)`.

### 4.4.4 Progrès-Sûreté-Progrès

La dernière règle à prouver est la règle de *progrès-sûreté-progrès*:

$$PSP\_LEAD : \frac{p \mapsto q, r \ unless \ b}{p \wedge r \mapsto (q \wedge r) \vee b}$$

Pour ce théorème, nous ne pouvons utiliser une valeur précise pour l'indice, comme pour les deux derniers théorèmes. En effet, cette valeur est déduite de la preuve manuelle suivant les règles appliquées. La preuve de la règle *PSP_LEAD* nécessite l'application de règles différentes suivant que l'on se trouve dans le cas de la disjonction ou dans celui de la transitivité. La preuve se fait alors par induction pour utiliser le schéma décrit au début de ce chapitre. La structure de la preuve est la suivante:

**Cas de base:**
Hyp: $ensures(p, q, pgm), r \ unless \ b$
But: $p \wedge r \mapsto (q \wedge r) \vee b$

**Hypothèses d'induction:**
Transitivité:
$p \mapsto c, c \mapsto q$
Disjonction:
$c \mapsto q$
$d \mapsto q$
$p \equiv (c \vee d)$

**But:** $p \wedge r \mapsto (q \wedge r) \vee b$

Étudions le cas de la transitivité:

| | |
|---|---|
| $p \mapsto c$ | (1) *Hyp* |
| $r \ unless \ b$ | (2) *Hyp* |
| $p \wedge r \mapsto (c \wedge r) \vee b$ | (3) *Induction avec* (1) *et* (2) |
| $c \mapsto q$ | (4) *Hyp* |
| $c \wedge r \mapsto (q \wedge r) \vee b$ | (5) *Induction avec* (4) *et* (2) |
| $p \wedge r \mapsto (q \wedge r) \vee b$ | (6) $CAN\_LEAD$(3,5) |

Étudions le cas de la disjonction:

| | |
|---|---|
| $c \mapsto q$ | (1) *Hyp* |
| $r \ unless \ b$ | (2) *Hyp2* |
| $c \wedge r \mapsto (q \wedge r) \vee b$ | (3) *Induction*(1,2) |
| $d \mapsto q$ | (4) *Hyp* |
| $d \wedge r \mapsto (q \wedge r) \vee b$ | (5) *Induction*(2,4) |
| $(c \vee d) \wedge r \mapsto (q \wedge r) \vee b$ | (6) $DISJ\_leads$(3,5) |
| $p \equiv (c \vee d)$ | (7) *Hyp* |
| $p \wedge r \mapsto (q \wedge r) \vee b$ | (8) $CP1$(6,7) |

En LP, le théorème à prouver est :

```
Set name PSP_lead
prove
  (leads_to2(p,q,insert(a1,pgm),n) /\ unless(r,b,insert(a1,pgm)))
    => (\E m ((n < m) /\ leads_to2(p ^ r,(q ^ r) \or b,insert(a1,pgm),m)))
  by induction on n using Nat.4
..
```

La structure de la preuve est :

```
Creating subgoals for proof by well founded induction on 'n'
Hypothesis:
  PSP_leadInductHyp.1:
  n < nc => (unless(r,b,insert(a1,pgm)) /\ leads_to2(p,q,insert(a1,pgm),n)
            => \E m (n < m
                     /\ leads_to2(p ^ r,(b \or r) ^ (b \or q),
                                  insert(a1,pgm),
                                  m)))
Subgoal:
  unless(r,b,insert(a1,pgm))
    /\ leads_to2(p,q,insert(a1,pgm),nc)
    => \E m (nc < m /\ leads_to2(p^ r,(b\or r) ^ (b \or q),insert(a1,pgm),m))
```

Nous procédons par implication, les hypothèses sont les prémisses de la règle d'inférence à prouver et l'hypothèse d'induction :

```
PSP_leadImpliesHyp.1.1: unless(rc,bc,insert(a1c,pgmc)) -> true
PSP_leadImpliesHyp.1.2: leads_to2(pc,qc,insert(a1c,pgmc),nc) -> true
PSP_leadInductHyp.1:
  n < nc
    => (unless(r,b,insert(a1,pgm))
        /\ leads_to2(p,q,insert(a1,pgm),n)
        => \E m (n < m
                 /\ leads_to2(p ^ r,(b \or r) ^ (b \or q), insert(a1,pgm),m)))
```

et le but:

```
\E m (nc < m
        /\ leads_to2(pc ^ rc,(bc \or rc) ^ (bc \or qc),insert(a1c,pgmc),m))
```

Les étapes sont les suivantes :

- Nous instancions l'axiome de définition pour avoir les trois cas, *ensures*, *transitivité* et *disjonction.*
- *Transitivité*: Nous choisissons des fonctions de skolem pour fixer les variables liées par le quantificateur existentiel de l'hypothèse du cas, afin de déduire les faits :

  ```
  PSP_lead.4.1: s(max(k,t)) = nc
  PSP_lead.4.2: leads_to2(pc,rcst,insert(a1c,pgmc),k)          %(1)
  PSP_lead.4.3: leads_to2(rcst,qc,insert(a1c,pgmc),t)          %(4)
  ```

  Puis nous instancions les variables libres dans l'hypothèse d'induction pour obtenir :

  ```
  PSP_leadInductHyp.1.1:
    \E m (k < m
          /\ leads_to2(pc^ rc,(bc \or rc)^ (bc\or rcst),insert(a1c,pgmc),m))
  ```

```
    -> true

PSP_leadInductHyp.1.2:
  \E m (t < m
       /\ leads_to2(rcst^ rc,(bc \or rc)^ (bc\or qc),insert(a1c,pgmc),m))
      -> true
```

Il nous faut alors choisir deux fonctions de skolem `skm1` et `skm2`, pour fixer la variable quantifiée `m` dans chaque hypothèse. Nous obtenons les deux faits :

```
PSP_lead.5.1: k < skm1 -> true
PSP_lead.5.2: leads_to2(pc ^ rc,                              %(3)
                (bc \or rc) ^ (bc \or rcst),
                insert(a1c,pgmc),
                skm1)
                -> true
PSP_lead.6.1: t < skm2 -> true
PSP_lead.6.2: leads_to2(rcst ^ rc,                            %(5)
                (bc \or rc) ^ (bc \or qc),
                insert(a1c,pgmc),
                skm2)
                -> true
```

Nous utilisons alors la règle `CAN_lead`, pour inférer:

```
can_lead.1.1: leads_to2(pc ^ rc,                              %(6)
                (bc \or rc) ^ (bc \or qc),
                insert(a1c,pgmc),
                s(max(skm1,s(s(skm2)))))
                -> true
```

Pour résoudre le but :

```
  \E m (nc < m
       /\ leads_to2(pc^ rc,(bc\or rc)^ (bc\or qc),insert(a1c,pgmc),m))
```

il faut spécialiser `m` à `s(max(skm1,s(s(skm2))))`, puis montrer que
`nc < s(max(skm1,s(s(skm2))))`,
sachant que `nc = s(max(k,t))`, `k < skm1` et `t < skm2`. Pour cela, nous utilisons le lemme suivant :

```
((i < n) /\ (j < m)) => (max(i,j) < max(n,s(s(m))))
```

que nous avons montré par cas suivant la valeur du maximum. Pour terminer la preuve, il suffit d'instancier `i` par `k`, `j` par `t`, `n` par `skm1` et `m` par `skm2`.

- *Disjonction*: Nous choisissons les variables de skolem pour fixer les variables quantifiées dans l'hypothèse du cas. Le processus est similaire au précédent. Nous obtenons les faits suivants :

```
  PSP_lead.6.1: s(max(k,t)) = nc
  PSP_lead.6.2: pc = ccst \or dcst
  PSP_lead.6.3: leads_to2(ccst,qc,insert(a1c,pgmc),k)          %(1)
  PSP_lead.6.4: leads_to2(dcst,qc,insert(a1c,pgmc),t)          %(4)
```

puis pour appliquer la disjonction, nous utilisons les hypothèses d'induction et la skolémisation pour obtenir la propriété :

```
disj_leads.2.1: leads_to2((ccst \or dcst) ^ rc,                %(6)
                  (bc \or rc) ^ (bc \or qc),
                  insert(a1c,pgmc),
                  s(max(s(skm1),s(skm2)))
                  -> true
```

et le but est :

```
\E m (nc < m
   /\ leads_to2((ccst \or dcst) ^ rc, (bc \or rc)^ (bc \or qc),
         insert(a1c,pgmc),m))
```

La preuve se poursuit de façon similaire au cas de la transitivité, en spécialisant `m` à `s(max(s(skm1),s(skm2))` et en montrant le lemme suivant :

```
((i < n) /\ (j < m)) => (max(i,j) < max(s(n),s(m)))
```

### 4.4.5 Le principe d'induction

Nous allons à présent montrer comment formaliser le principe d'induction associé à l'opérateur *leads_to* :

$$\textbf{Induction :} \quad \frac{\forall m : m \in W :: p \wedge M = m \ leads\_to \ (p \wedge M <_w m) \vee q}{p \ leads\_to \ q}$$

Comme nous l'avons décrit au chapitre 2, nous avons défini un ordre lexicographique sur les listes d'expressions arithmétiques. En utilisant la nouvelle définition de l'opérateur *leads_to*, ce principe est formalisé comme suit :

```
Set name induc_principle
assert
  leads_to2(p ^ (exp_list1 == exp_list2),
            (p ^  (exp_list1 ≪ exp_list2)) \or q,pgm,n)
  => \E m ((n < m) /\ leads_to2(p,q,pgm,m))
..
```

En utilisant ce principe, nous avons prouvé la règle suivante :

$$\textbf{IND\_coroll :} \quad \frac{p \wedge M = m \ leads\_to \ M < m}{true \ \ leads\_to \ \neg p}$$

dont la preuve utilise le corollaire suivant :

$$\textbf{leads\_to\_coroll :} \quad \frac{p \wedge q \ leads\_to \ r}{p \ \ leads\_to \ \neg q \vee r}$$

formalisé en LP :

```
Set name coro_lead1
assert
  leads_to2(p ^ q,r,pgm,n)
    => (\E m ((n < m) /\ leads_to2(p,nnot(q) \or r,pgm,m)))
```

La preuve de **IND_coroll** est la suivante :

| | |
|---|---|
| $p \wedge M = m \mapsto M < m$ | (1) *Hyp* |
| $M = m \mapsto M < m \vee \neg p$ | (2) *leads_to_coroll* |
| $true \mapsto \neg p$ | (3) *induct_principle* |

La preuve LP est :

```
Set name induc_princ2
prove
  leads_to2(p^ (exp_list1 == exp_list2),
            (exp_list1 ≪ exp_list2),pgm,n)
  => \E m ((n < m) /\ leads_to2(T,nnot(p),pgm,m))
  ..
  res by => -m
    <> => subgoal
    %----------------------(1)-(2)
    ins
      p by (exp_list1c == exp_list2c),
      q by pc,r by (exp_list1c ≪ exp_list2c),n by nc,
      pgm by pgmc   in coro_lead1
      ..
    dec op skm1:-> Nat                  %elimination in a hyp
    fix m as skm1 in coro_lead1.2.1
    %----------------------(2)-(3)
    instantiate
       p by T,exp_list1 by exp_list1c,exp_list2 by exp_list2c,
       pgm by pgmc,q by nnot(pc), n by skm1 in induc_principle
    ..
    dec op skm2:-> Nat                  %elimination in a hyp
    fix m as skm2 in induc_principle.2.1
    %----------------------conclusion of the principle
    res by spec m to skm2               %elimination in the goal
      <> specialization subgoal
      ins i by nc,j by skm1,n by skm2 in nat.22
      [] specialization subgoal
    [] => subgoal
  [] conjecture
qed
```

La preuve mécanisée possède la même structure que la preuve manuelle : Tout d'abord, nous utilisons le corollaire, par une instanciation explicite. Nous obtenons la conclusion de l'implication, qui comprend un quantificateur existentiel. Il faut donc skolémiser en utilisant une constante `skm1`, afin d'obtenir la conclusion de la règle d'inférence, représentée par le fait `coro_lead.2.1`. Toujours d'après la preuve manuelle, nous utilisons le principe d'induction en instanciant l'axiome correspondant avec les termes adéquats. La prémisse est alors le fait `coro_lead.2.1`, qui nous permet d'obtenir la conclusion du principe d'induction, `induc_principle.2.1`. Pour conclure il suffit d'éliminer le quantificateur existentiel de "l'hypothèse" avec une constante de skolem `skm2` et enfin spécialiser la variable quantifiée du but à cette constante.

## 4.5 Utilisation des règles d'inférence

Le but de ce paragraphe est d'étudier l'utilisation des règles d'inférence de la relation *leads_to*. Cette utilisation dépend de la présence du quantificateur existentiel dans la définition. Les règles,

prouvées aux paragraphes précédents, sont :

```
assert
((ensures(p,q,pgm))
\/ (\E r \E i \E j
    (leads_to2(p,r,pgm,i) /\ leads_to2(r,q,pgm,j) /\ (s(max(i,j))=n)))
\/ (\E c \E d \E i \E j
    (leads_to2(c,q,pgm,i) /\ leads_to2(d,q,pgm,j) /\ (s(max(i,j))=n)
     /\ (p = (c \or d)))))
= leads_to2(p,q,pgm,n)
..
Set name impl_lead
 ((p \=> q)=T) => (leads_to2(p,q,insert(a1,pgm),n))
..
set name impo_lead
  leads_to2(p,F,insert(a1,pgm),n) => (nnot(p)=T)
..
set name disj_leads
  (leads_to2(p,q,insert(a1,pgm),i) /\ leads_to2(b,r,insert(a1,pgm),j))
     => leads_to2(p \or b,q \or r,insert(a1,pgm),max(s(i),s(j))+1)
  ..
Set name can_lead
  (leads_to2(p,q \or b,insert(a1,pgm),i) /\ leads_to2(b,r,insert(a1,pgm),j))
    => leads_to2(p,q \or r,insert(a1,pgm),s(max(i,s(s(j)))))
  ..
Set name PSP_lead
  (leads_to2(p,q,insert(a1,pgm),n)/\ unless(r,b,insert(a1,pgm)))
     => (\E m ((n < m) /\ leads_to2(p ^ r,(q ^ r) \or b,insert(a1,pgm),m)))
  ..
```

Prouver qu'un programme *Prog* vérifie une propriété *p leads_to q* en utilisant ces règles d'inférence revient à montrer :

```
\E m leads_to2(p,q,prog,m)
```

Étudions les différentes possibilités :

- Le programme vérifie la propriété $ensures(p, q, prog)$, il suffit d'appliquer l'axiome de définition, pour obtenir $leads\_to2(p, q, prog, n)$, pour tout $n$.
- Le programme vérifie $leads\_to2(p, c, prog, k)$ et $leads\_to2(c, q, prog, t)$, alors on peut utiliser la transitivité de l'axiome de définition. Il faut alors **prouver** qu'il existe $r$,$i$ et $j$ tels que la formule logique correspondant à la transitivité soit vraie. Il suffit alors de spécialiser, dans la formule à prouver, les variables quantifiées par $c$, $k$ et $t$.

  Supposons que les deux faits suivants soient vrais dans le système courant :

  ```
  leads_to2(pc,cc,prog,kc)
  leads_to2(cc,qc,prog,tc)
  ```

  et que la preuve consiste à montrer que *pc leads_to qc*. La structure de la preuve est alors :

  ```
  prove \E m leads_to2(pc,qc,prog,m)
   res by spec m to s(max(kc,tc))
    % on applique la transitivité
    instantiate p by pc,q by qc,n by s(max(kc,tc)) in leads_to2
    prove
  ```

```
      \E r \E i \E j
         ((s(max(i,j)) = s(max(kc,tc)))
        /\ leads_to2(pc,r,prog,i)
        /\ leads_to2(r,qc,prog,j))
  ..
    res by spec r to cc,i to kc,j to tc
qed
```

Il est important de noter qu'il ne suffit pas d'instancier les variables libres de l'axiome de définition. En effet, dans ce cas, il faudrait fixer les variables quantifiées en utilisant de "nouvelles" constantes $c1$, $c2$ et $c3$ et affirmer que ces constantes sont équivalentes aux valeurs $c$, $k$ et $t$.

- Le programme vérifie *leads_ to2(p1,q,prog,k)* et *leads_ to2(p2,q,prog,t)*. De plus, $p \equiv p1 \vee p2$. On peut alors utiliser la disjonction dans l'axiome de définition. La structure de la preuve est similaire à la transitivité :

```
prove \E m leads_to2(pc,qc,prog,m)
  res by spec m to s(max(kc,tc))
    ins p by pc,q by qc,pgm by prog,n by s(max(kc, tc)) in leads_to2
    prove
      \E c \E d \E i \E j
         (pc \or rc = c \or d
            /\ (s(max(kc,tc))=s(max(i,j)))
            /\ leads_to2(c, qc, prog, i)
            /\ leads_to2(d, qc, prog, j))
      ..
      res by spec c to p1c,d to p2c,i to kc,j to tc
qed
```

- Le programme vérifie $pc \Rightarrow qc$, il suffit alors d'appliquer la règle d'implication. Cela nous permet de montrer *leads_ to2(p,q,prog,n)*, pour tout $n$, et par conséquent qu'il existe un $m$ tel que la propriété soit vraie.

- Le programme vérifie $leads\_to2(p1, q1, prog, k)$ et $leads\_to2(p2, q2, prog, t)$, et $p \equiv (p1 \vee p2)$, $q \equiv (q1 \vee q2)$, il faut appliquer la règle *disj_ leads*. Pour cela, on spécialise $m$ à $max(s(k), s(t)) + 1$ et on instancie les variables libres de la règle *DISJ_ leads*. Supposons qu'on ait les faits

```
leads_to2(p1c,q1c,prog,kc)
leads_to2(p2c,q2c,prog,tc)
pc = p1c \or p2c
qc = q1c \or q2c
```

La structure de la preuve est alors :

```
prove \E m leads_to2(pc,qc,prog,m)
 res by spec m to max(s(kc),s(tc))+1
  ins
    p by pc,q by qc, b by dc,r by rc,i by kc,j by tc
    in disj_lead
  ..
qed
```

- Le programme vérifie *leads_to2*$(p, q1 \vee b, prog, k)$ et *leads_to2*$(b, r, prog, t)$, et $q \equiv (q1 \vee r)$, il faut appliquer la règle d'élimination, *CAN_LEAD*. Pour cela, on spécialise $m$ à $s(max(k, s(s(t))))$ et instancie les variables libres de la règle *CAN_LEAD*.

```
leads_to2(pc,q1c \or bc,insert(ac,prog),kc)
leads_to2(bc,rc,insert(ac,prog),tc)
q \<=> q1c \or rc
```

et la preuve consiste à montrer que *pc leads_to qc*. La structure de la preuve est alors :

```
prove \E m leads_to2(pc,qc,insert(ac,prog),m)
 res by spec m to s(max(k,s(s(t))))
  ins
    p by pc,q by q1c, b by bc,r by rc,
    i by kc,j by tc, a1 by ac,pgm by prog in can_lead
  ..
qed
```

- Le programme vérifie *leads_to2(p1,q1,prog,k)*, *unless(r,b,prog)*, $p \equiv p1c \wedge r$, et $q \equiv (q1 \wedge r) \vee b$. On doit utiliser la règle `PSP_lead`. Pour cela, il faut instancier les variables libres de la règle `PSP_lead`, puis fixer le quantificateur existentiel de la conclusion de l'implication avec une nouvelle fonction de skolem puis spécialiser dans le but $m$ à cette constante.

Nous avons les faits suivants dans le système :

```
leads_to2(p1c,q1c,insert(ac,prog),kc)
unless(rc,bc,prog)
pc = p1c ^ rc
qc = (q1c ^ rc) \or bc
```

et nous voulons prouver

```
\E m leads_to2(pc,qc,insert(ac,prog),m)
```

La structure de la preuve est alors :

```
prove \E m leads_to2(pc,qc,insert(ac,prog),m)
  ins
    p by p1c,r by rc,b by bc,q by q1c, n by kc,
    a1 by ac,pgm by prog in psp_lead
  ..                                              %psp_lead.1.1
  dec op cst:-> Nat
  fix m as cst in PSP_leal.1.1
  res by spec m to cst
qed
```

## 4.6 Conclusion

Pour les spécifications abstraites, le quantificateur existentiel accroît l'expressivité du langage. L'utilisation du quantificateur existentiel a permis une définition plus intuitive des relations *ensures* et *leads_to*. Cependant, les preuves des théorèmes de la relation *ensures* sont moins

automatiques, à cause de la gestion des quantificateurs. En effet, l'utilisation des hypothèses comprenant des quantificateurs existentiels doit être explicite pour éliminer ces quantificateurs par la skolémisation et n'est donc pas automatique. On pourrait supposer une automatisation de la skolémisation, c'est la "spécialisation" qui est moins évidente, car il faut trouver la bonne valeur de spécialisation.

Néanmoins, le quantificateur existentiel permet un raisonnement plus intuitif, plus proche des preuves manuelles. Par conséquent, la certification de ces preuves est plus aisée en cas d'échec puisque nous pouvons déterminer l'action du programme qui ne vérifie pas la propriété. Comme nous le verrons dans les chapitres consacrés aux applications, les propriétés *ensures* sont plus naturelles car nous explicitons l'action qui vérifie la propriété que l'on cherche à prouver.

Par ailleurs, la formalisation précédente ne nous permettait pas de prouver les théorèmes de la relation *leads_to*. En utilisant les quantificateurs existentiels et un schéma d'induction nothérienne, nous avons pu formaliser le raisonnement effectué, pour dériver une propriété *leads_to*. Cependant, comme cette recherche d'une meilleure formalisation de cette relation est récente, nous pensons que cette formalisation de *leads_to* peut être encore améliorée, en simplifiant les axiomes :

```
set name leads_to2
assert
leads_to2(p,q,pgm,0) =  ensures(p,q,pgm);
leads_to2(p,q,pgm,s(i)) =
   \E c \E d \E j
    ((i < j)
     /\ p = c \or d
     /\ leads_to2(c,q,pgm,i)
     /\ leads_to2(d,q,pgm,j))
 \/  \E r \E j
    ((i < j)
    /\ leads_to2(p,r,pgm,i)
    /\ leads_to2(r,q,pgm,j))
..
```

Le schéma d'induction à utiliser serait toujours l'induction bien fondée. Nous pensons que cette formalisation simplifierait les preuves mécanisées, tout en utilisant une même sémantique de la relation *leads_to*. Ceci constitue une des perspectives intéressante de ce travail.

Dans les études de cas décrites dans la deuxième partie de ce document, nous utilisons la formalisation de UNITY décrite dans les chapitres précédents, en particulier la définition de *ensures* et *leads_to* sans les quantificateurs existentiels. Cependant, pour le problème des lecteurs rédacteurs, nous donnons quelques idées de l'impact de ces améliorations pour les preuves des propriétés du programme.

Dans le chapitre suivant, nous proposons une méthodologie de spécification et de vérification à l'aide de LP, basée sur notre expérience et sur les preuves décrites jusqu'à présent. En particulier, nous proposons une méthode incrémentale pour la spécification d'une base de travail ainsi que des principes de gestion et d'organisation des preuves complexes.

# Chapitre 5

# Méthodologie

## 5.1 Introduction

Un démonstrateur de théorèmes vient en général avec un manuel, mais rarement avec "un livre de recettes" qui expliquerait comment se tirer de telle ou telle situation. Ce livre de recettes dépendrait d'ailleurs du domaine d'utilisation. Nous allons donc dans ce chapitre faire part de notre expérience dans le domaine de la preuve mécanisée à l'aide d'un démonstrateur de théorèmes, et en particulier avec LP, en espérant qu'elle sera utile à d'autres.

Les techniques formelles ne sont pas faciles à comprendre, à apprendre et à utiliser. Pour être capable d'appliquer des techniques formelles particulières, il est nécessaire de comprendre la théorie de base. De plus, il faut apprendre comment appliquer les règles qui composent ces techniques. Enfin, il est nécessaire de comprendre comment appliquer une règle donnée pour résoudre un but particulier. Structurer le but en sous buts et organiser le travail en étapes pour résoudre ou pour atteindre ces sous buts nécessite une *méthode.*

Le but de ce chapitre est d'être à la fois une synthèse et une aide pour l'utilisation de LP. Il est certain qu'il serait inutile de publier tous les essais infructueux et tous les échecs auxquels nous avons été confrontés avant d'obtenir notre version finale. Néanmoins, ce n'est pas juste de présenter la version finale comme si elle a été obtenue de façon descendante directe et il serait incorrect d'affirmer que c'est la seule façon d'obtenir ces résultats (ou ces preuves). Un néophyte croirait qu'utiliser LP serait aisé et serait très surpris des difficultés qu'il aurait en face des preuves même les plus simples parfois. Il n'est pas surprenant que les ingénieurs du logiciel soient dépités lorsqu'ils essayent les techniques formelles en pratique, car c'est un art difficile.

Dans ce qui suit, nous décrivons une méthodologie essentiellement basée sur notre expérience de l'utilisation des techniques formelles pour des applications de tailles croissantes. Bien sûr, une méthode ne peut garantir de guider le développeur vers "la" solution, sinon le processus de conception pourrait être complètement automatisable. Néanmoins, les méthodes peuvent aider le développeur dans sa tâche.

## 5.2 Une méthode triviale

Lorsqu'on décide de vérifier formellement une application, les deux principales étapes sont la construction d'une description formelle des données (sur lesquelles porte la vérification) et des propriétés, ce qu'on appelle généralement le *type abstrait*, et la *vérification* de ce que l'application est censée faire. Cette vérification est donc basée sur la spécification de l'application, sa validité dépend donc de celle de sa spécification.

Une spécification est dite *contradictoire*, si toute formule peut être déduite à partir de ses axiomes. En particulier, en LP, lorsque la formule `true = false` est déduite d'un ensemble d'équations alors la spécification est contradictoire, ou inconsistante. Il n'y a aucun moyen de prouver la consistance d'une spécification, car prouver que $false \neq true$ pour une spécification ne suffit pas à prouver que la spécification est cohérente. Par conséquent, on peut prouver l'inconsistance mais non la consistance.

Néanmoins, nous considérons que cette liberté de définition est l'un des avantages de LP qui facilite l'écriture et le choix des définitions. Par conséquent, pour minimiser les erreurs, nous utilisons une méthode incrémentale, qui permet d'assurer un minimum de correction pour les spécifications définies. Cette méthode consiste d'une part à choisir des spécifications "usuelles", en particulier les spécifications LSL quand cela est possible, et d'autre part à prouver le maximum de propriétés sur les spécifications. C'est le but principal de notre méthode incrémentale, qui nous permet, non pas de garantir la cohérence mais tout au moins de minimiser les erreurs à l'aide des outils de preuves fournis par LP.

## 5.3 Une méthode incrémentale

La méthode de développement que nous utilisons comprend quatre phases cycliques : *Définir*, *Enrichir*, *Déduire*, et *Vérifier*. La première phase consiste à définir les objets et leurs opérations pour construire la spécification de base. La seconde consiste à enrichir cette spécification en introduisant de nouveaux objets en utilisant un même principe de définition. La troisième phase consiste à montrer des propriétés que la spécification est supposée satisfaire et enfin la dernière phase consiste à éliminer les redondances et à vérifier que la base ne contient pas d'inconsistance, si cela est possible.

### 5.3.1 Les objets prédéfinis

A la différence du système NQTHM [Boy79, Boy88] ou HOL [Gor93], mis à part la sorte `Bool`, le démonstrateur LARCH ne contient aucune théorie prédéfinie. LP comprend un ensemble de règles de réécriture et de règles de déduction sur les opérateurs booléens, `=>` , `<=>` , `/\`, `\/` et `~`, ainsi que sur les opérateurs `=` et `if then else`, automatiquement déclarés pour chaque sorte `S`. Cet ensemble de règles de réécriture pré-définies n'est pas complet, pour des raisons de temps de calcul et d'espace, mais généralement suffisant pour prouver un grand nombre de propriétés sur les booléens.

### 5.3.2 Définition des objets de base

Par analogie à un langage de programmation, mis à part la sorte `Bool` et les opérateurs logiques associés, tout ce qu'on utilise dans un script LP doit être déclaré : les sortes, les opérateurs et les variables.

Définir les objets de base consiste à définir une théorie en LP. Cette théorie est constituée de deux types d'objets, les sortes et les opérateurs (ou fonctions). Puis, on définit les propriétés de base des opérateurs, en d'autres termes, le pourquoi de leur utilisation. Le tout forme donc une spécification.

La plupart des définitions que nous utilisons font partie du répertoire classique. Vu le nombre d'exemples de spécification fournis par les spécification LSL, l'idée consiste à rechercher parmi ces spécifications celles qui nous semble la plus appropriée à nos besoins, sous réserve de modification si nécessaire. En effet, LP a été conçu à la base pour vérifier des spécifications LSL. Il semble alors

évident que le modèle de spécification défini par ce langage est adéquat pour utiliser l'outil LP. Si les objets à définir, ou le type de données, n'appartiennent pas aux types de données fournis par les spécifications LSL alors on peut les construire comme suit :

La première étape consiste à nommer la spécification, ou l'ensemble des objets que l'on est entrain de définir : **set name** *ma_specif*

Cette "identification" est importante lorsque nous définissions une base de règles de taille importante. Cela permet de structurer la spécification en plusieurs morceaux, afin de faciliter l'utilisation et la maintenance.

La deuxième étape consiste à déclarer les ensembles d'objets que l'on manipule, à savoir les sortes et les opérations qui utilisent ces objets.

**déclare sort** *sorte1, sorte2, ...*
**declare operators**
*op1*: —>
*op2*: *sorte1* —> *sorte2*
⋮

Enfin, la troisième étape consiste à définir ces opérations à l'aide d'équations ou plus généralement de formules, en particulier les opérateurs que l'on sait être associatifs et/ou commutatifs, les schémas d'induction, etc. La stratégie consiste à définir un minimum de propriétés, pour éviter les redondances et surtout les contradictions. Il faut remarquer qu'en général, on connaît les propriétés de base des opérations que l'on ait entrain de définir, c'est le but de leur définition!

Les bases de règles des entiers naturels et des séquences (Chapitre 3) sont basées sur les spécifications LSL [Gut93] correspondantes, modifiées pour les besoins de notre application, à savoir la vérification des programmes UNITY. Les expressions booléennes, c'est à dire les prédicats, et les opérateurs temporels ont été définis d'après les travaux de Dijkstra [Dij89] et des définitions UNITY [Cha88].

### 5.3.3 Enrichissement de la base

La deuxième phase du développement consiste à enrichir la spécification de base. En effet, une base de règles LP n'est pas figée, on peut à tout moment ajouter des sortes, des opérations et des propriétés.

Il faut noter que LP ne fait pas de différence syntaxique entre les définitions et les propriétés à prouver. En effet, toute équation (formule) introduite avec le mot clé **assert** sera considérée par LP comme un axiome. La différence sémantique entre une équation "de définition" et une équation spécifiant une propriété est gérée par l'utilisateur, qui a la responsabilité de "prouver" les formules spécifiant des propriétés.

Par conséquent, un premier type d'enrichissement peut être l'ajout d'une équation ne contenant aucun opérateur précédemment défini. Auquel cas, cette équation est considérée comme une définition et le principe est le même que celui de la phase précédente puisque les nouveaux opérateurs doivent être déclarés. En particulier, lorsqu'il s'agit de nouveaux objets, l'utilisation de nouveaux noms, ou bien de noms composés, est recommandée pour faciliter la mise au point. Un deuxième type d'enrichissement est la définition d'un opérateur à l'aide des opérateurs précédemment définis. Ce sont des équations de la forme

$$f_i(\ldots) = g_i(\ldots)$$

où $g_i(\ldots)$ ne contient pas le symbole $f_i$.

On peut aussi enrichir la base par une re-définition des opérateurs en fonction de ceux nouvellement introduits. Ces deux types d'enrichissement ne modifient pas la correction de la base de règles.
Enfin, si les équations introduites ne contiennent que des opérateurs précédemment définis alors ces équations sont considérées comme des propriétés que la spécification doit satisfaire et par conséquent, cela s'inscrit dans la phase suivante.

### 5.3.4 Déduction et Vérification

La troisième phase du développement peut être couplée avec la précédente. En effet, une fois les propriétés énoncées, il faut s'assurer que nous avons bien formalisé les besoins. Cela consiste à montrer que l'ensemble des axiomes satisfait un ensemble de propriétés ou qu'une certaine propriété est bien une conséquence d'un ensemble d'axiomes.
En effet, les définitions sont un ensemble d'axiomes dont on doit assurer la correction alors qu'une propriété est une conjecture que l'on doit essayer de prouver. Si nous utilisons les spécifications LSL alors les propriétés énoncées dans les clauses `implies` doivent être prouvées et en particulier toutes les obligations de preuves contenues dans cette spécification.

### 5.3.5 Validation et Amélioration

Cette phase consiste à éliminer les redondances et à vérifier que la base n'est pas inconsistante. Cette dernière est très importante. En effet, au cours du développement, on peut s'apercevoir qu'un axiome introduit ne satisfait pas les besoins que l'on s'est imposés ou plus simplement qu'une équation inconsistante a été introduite, auquel cas sa détection doit se faire au plus tôt. En effet, cette méthode incrémentale fait que les spécifications peuvent être définies de façon hiérarchiques (par enrichissement) et par conséquent, une inconsistance met en cause toute spécification descendante. Or cette remise en cause et les corrections qu'elle nécessite, peuvent être coûteuses en temps et en efficacité car on peut oublier comment et pourquoi tel fait a été introduit et surtout prouvé. En effet, une preuve basée sur une spécification inconsistance n'a aucune valeur, d'où l'intérêt de découvrir ces erreurs au plus tôt. Ceci peut être mis à profit avec les nombreuses facilités de LP pour la mise au point des preuves, à savoir commenter les preuves, l'identification des règles par leur nom et le retour arrière (*backtraking*).

## 5.4 Les stratégies de preuves

LP est un assistant de preuve et non un démonstrateur automatique dans le sens strict du terme, du fait qu'il n'effectue pas de façon automatique des preuves difficiles et qu'il n'utilise pas non plus d'heuristiques, comme le système NQTHM de Boyer & Moore, pour générer des conjectures utiles à la preuve. Par contre, il offre plusieurs facilités pour la mise au point des preuves. Néanmoins, dès lors que l'on considère LP comme un *correcteur de preuve*, la notion de *conception* d'une preuve devient évidente. En effet, mis à part les preuves des propriétés de base, une preuve en LP doit être conçue, écrite et corrigée. Or cette notion de conception sous-entend un choix de méthodes appropriées pour montrer la conjecture.

Comme nous l'avons vu au chapitre 1, les méthodes de preuves disponibles en LP sont nombreuses et de différentes natures. Une classification simpliste permet de les distinguer suivant qu'elles appartiennent à des techniques liées au système de réécriture, telle que la *normalisation*

ou le *calcul de paires critiques*, ou à des techniques standards, telles que la *preuve par l'absurde* ou bien *par cas*, ou bien à des techniques liées à la forme syntaxique de la conjecture, telle que l'*implication* ou la *conjonction*. Néanmoins, cette classification ne donne aucune méthode adéquate à toute conjecture. En effet, lorsqu'on conçoit une preuve, on s'intéresse généralement à choisir une méthode parmi celles fournies par le démonstrateur, et non pas à l'organisation des sous buts de la preuve. Or, les méthodes d'inférence *arrière* (guidées par le but) produisent pour un but donné, un ensemble de sous-buts à satisfaire. Les méthodes utilisées pour montrer les sous-buts dépendent de la méthode choisie au départ pour le but. Typiquement, prouver une implication provoque le remplacement des variables libres par des constantes sur lesquelles on ne peut appliquer l'induction.

Par conséquent, nous décrivons dans la suite quelques propriétés à appliquer pour décider de la méthode à utiliser pour une conjecture donnée.

### 5.4.1 L'induction

Les méthodes d'induction sont des mécanismes puissants du démonstrateur. En effet, bien que ce mécanisme existe dans d'autres démonstrateurs, tels que le système NQTHM, LP offre de plus la possibilité de définir (et de prouver) plusieurs schémas d'induction pour une même sorte, de définir un nombre quelconque de générateurs dans la même règle d'induction et enfin la possibilité de contrôler la profondeur d'une induction.
Il est évident que les méthodes de preuves par induction sont les plus immédiates lorsque la conjecture à prouver comprend des variables d'une sorte pour laquelle nous avons défini un schéma d'induction.

#### Choix d'une règle d'induction

Pour une même sorte $S$, plusieurs schémas d'induction peuvent être disponibles. Le problème consiste alors à choisir le schéma d'induction le plus adéquat. La stratégie consiste à "étudier" la conjecture à prouver et plus précisément les opérateurs qui la constituent.

- Si les opérateurs sont définis en fonction d'opérateurs avec lesquels nous avons défini une règle d'induction $R$ alors choisir cette règle d'induction.

- Si certains opérateurs $op1s$ sont définis en fonction des générateurs de $R$, et d'autres $op2s$, en fonction de ceux de $R'$, alors si les définitions de $op2s$ ont été complétées par rapport aux opérateurs de $R$, alors choisir comme règle d'induction $R$.

- Si la conjecture utilise un ensemble d'opérateurs dont certains sont définis avec les opérateurs d'une règle d'induction $R$ et d'autres avec ceux de la règle d'induction $R'$, alors la solution consiste à choisir une des deux règles et à compléter la définition des opérateurs par rapport aux opérateurs de la règle choisie.

- Enfin, si tous les opérateurs sont définis par rapport aux générateurs de $R$ et de $R\prime$, alors choisir celle de l'opérateur le plus interne, c'est à dire de celui qui s'applique directement sur la variable sur laquelle porte l'induction.
  Le principe est identique lorsque les opérateurs ne sont pas définis en fonction de générateurs mais en fonction d'opérateurs qui sont eux définis en fonction de générateurs (cf. preuve 3.11).

**Exemple 5.1** *Reprenons la spécification des séquences (cf. Chapitre 3, paragraphe 3.4), où les opérateurs* `head` *et* `tail` *ont été définis à l'aide de* `-|: Exp,Seq -> Seq` *et* `len` *a été définie avec* `|-: Seq,Exp -> Seq`*:*

```
head(exp1 -| seq1) = exp1;
tail(exp1 -| seq1) = seq1;
len(emptyseq)              = 0;
len(seq1 |- exp1) = len(seq1) + 1;
isempty(seq1)   <=> (seq1  = emptyseq);
sort Seq generated by emptyseq,|-;                    %seq_ind1
Sort Seq generated by emptyseq,-|                      %seq_ind2
```

*Soit la propriété à prouver:*

```
when isempty(s2) <=> isempty(s3),
     head(s2) = head(s3),
     tail(s2) = tail(s3)
yield s2 = s3
```

*Comme nous avons une règle d'induction sur* `Seq`, `seq_ind2`, *dont* `-|` *est un générateur, nous choisissons sans ambiguïté cette règle pour prouver la propriété par induction.*

*Soit maintenant la propriété:*

```
when len(s2) = len(s3), head(s2) = head(s3), tail(s2) = tail(s3)
  yield s2 = s3
```

*Nous avons alors deux règles d'induction car* `|-` *est un générateur de la règle d'induction* `seq_ind1`*. Dans ce cas, nous étudions la conjecture et notre base de règles (axiomes et théorèmes). A ce niveau du développement de la base des* séquences, `head` *et* `tail` *sont définis en fonction de* `-|` *(générateur de* `seq_ind2`*), mais n'ont pas encore été définis en fonction de* `|-` *(générateur de* `seq_ind1`*) alors que* `len` *a été "redéfinie" avec* `-|`*, car nous avons montré le lemme (trivial) suivant:*

```
len(exp1 |- seq1) = len(seq1) + 1
```

*Nous choisissons alors, pour prouver la conjecture, la règle d'induction* `seq_ind2`*.*

*La propriété suivante,*

```
¬isempty(seq1) => (count(last(seq1),seq1) > 0)
```

*est prouvée par induction en utilisant la règle* `seq_ind1`*, bien que* `count` *soit définie par rapport aux générateurs de* `seq_ind2`*. Nous avons choisi, seq_ind*1 *car l'opérateur le plus interne, ici* `last`*, est défini par les générateurs de seq_ind*1*.*

### Choix d'une profondeur d'induction

Un deuxième point qui nécessite réflexion est l'utilisation de d'une *profondeur* d'induction (cf. Chapitre 1, p. 21). En effet, comment déterminer pour une conjecture donnée, la profondeur de l'induction à appliquer?

Cela consiste à étudier la sémantique de la conjecture. En effet, lorsqu'on décide, pour montrer la conjecture $P(x)$, d'utiliser une induction sur la variable $x$, on voudrait vérifier que la propriété est satisfaite pour la "plus petite valeur" puis montrer que la propriété est préservée pour les

valeurs "plus grandes". Or cette "plus petite valeur", peut ne pas être le générateur de base de la sorte considérée. Par exemple, pour vérifier une propriété sur un tableau de taille $n$, dont les éléments sont $T[1]...T[n]$, en utilisant l'induction sur les entiers naturels, le cas de base ne sera pas significatif puisque l'élément $T[0]$ n'est pas défini. Par conséquent, l'induction de profondeur $p > 1$ nous permet de vérifier que la propriété est satisfaite pour "le cas de base significatif", c'est à dire si la propriété est satisfaite pour $T[1]$ (cf. paragraphe 3.4.6).

#### Choix de la variable d'induction

Lorsqu'une conjecture contient des variables de différentes sortes sur lesquelles on peut appliquer l'induction, la question est de savoir dans quel ordre invoquer l'induction. La solution consiste à étudier la forme de la conjecture, et surtout les cas de base de l'induction. Certains cas de base sont irréductibles car ils donnent des valeurs délibérément indéfinies.

### 5.4.2 Les méthodes syntaxiques

Les méthodes syntaxiques s'appuient sur la forme de la conjecture, telle que l'implication, l'équivalence, et la conjonction. Par conséquent, il pourrait sembler facile de décider du moment de leur utilisation. Néanmoins, sachant qu'une preuve à l'aide d'une méthode syntaxique, mis à part la conjonction, entraîne le remplacement des variables libres par des constantes, le choix des méthodes syntaxiques doit être fait en dernier recours. Par exemple, prouver par la règle d'implication, une assertion de la forme $P(x, y) \Rightarrow Q(y)$ introduit l'hypothèse $P(xc, yc)$ puis essaye de prouver le but $Q(yc)$. En particulier, on ne peut plus appliquer l'induction sur $Q(yc)$. Ce remplacement est dû au fait que les variables apparaissant dans les formules $P$ ne peuvent pas être quantifiées de façon universelle mais doivent représenter les mêmes variables que celles qui apparaissent dans $Q$. Or les variables libres dans une formule LP sont implicitement quantifiées de façon universelle [Gar91], d'où la nécessité de les remplacer par des constantes.
Par contre, si la formule à prouver contient un opérateur logique et est close, alors on peut appliquer la méthode syntaxique correspondante sans hésitation.

Si la conjecture à prouver est sous forme d'une négation, c'est à dire de la forme `~F`, alors il est conseillé d'utiliser la méthode par contradiction. C'est en général la méthode la plus adéquate pour cette forme du conjecture, sinon la négation est propagée dans les hypothèses et les sous buts de la méthode choisie.

**Exemple 5.2** *Soit la conjecture* $\tilde{}P_x(x)$, *où* $P_x$ *a la signature* $Nat \rightarrow Bool$. *Supposons que l'on choisisse d'utiliser l'induction sur* $x$, *le but introduit est le suivant :*

```
prove ~ Px(x) by induction on x
```

*Alors le cas de base sera de la forme* $\tilde{}P_x(0)$, *l'hypothèse de la forme* $\tilde{}P_x(c)$ *et le but* $\tilde{}P_x(s(c))$.

### 5.4.3 Factorisation des preuves

La méthode de conjonction, liée à la forme syntaxique de la conjecture, n'engendre pas de remplacement de variables par des constantes puisque la variable $x$ d'une formule de la forme $P(x) \wedge Q(x)$ contient deux quantificateurs universels implicites et est équivalente à $(\forall x)p(x) \wedge (\forall y)p(y)$. Un but de la forme `P(x) /\ Q(x)` est alors subdivisé en deux sous-buts `P(x)` et `Q(x)` où `x` ne représente pas forcément la même variable.
Par conséquent, si une conjecture est sous la forme d'une conjonction, on peut appliquer la

méthode `/\-m` et les méthodes de preuves, utilisées pour les sous-buts, sont complètement indépendantes.
Par contre, si la stratégie de preuve utilisée pour chaque sous-but est la même, par exemple un même lemme suffit pour résoudre chaque sous-but, alors il est plus judicieux de commencer par utiliser cette stratégie qui permet de simplifier l'ensemble des sous-buts en une seule étape. Un exemple typique correspond à la situation où la conjecture est de la forme

$$f(c_1) \wedge f(c_2) \wedge \ldots f(c_n)$$

où les $c_i$ sont des constantes de type `Nat` et nous disposons d'un lemme de la forme

$$j > 0 \Rightarrow f(j)$$

où $j$ est une variable de type `Nat` et des hypothèses

$$(c_1 > 0), (c_2 > 0), \ldots (c_n > 0)$$

Dans ce cas, les hypothèses et le lemme nous permettent de prouver la conjecture en une seule étape (par exemple avec un calcul de paires critiques), en utilisant la possibilité qu'offre LP pour nommer les faits que l'on manipule.

Ayant les hypothèses suivantes :

```
nomhyp.c1 : 0 < c1 -> true
nomhyp.c2 : 0 < c2 -> true
⋮
nomhyp.cn : 0 < cn -> true
nomlemme.1: 0 < j => f(j)  -> true
..
```

Si on utilise la méthode de la conjonction, la preuve aurait la structure[27] :

```
prove f(i1) /\ f(i2) ... f(in)
  resume by /\
    <> /\ subgoal
    critical-pairs nomhyp.c1 with nomlemme.1
    [] /\ subgoal
    <> /\ subgoal
    critical-pairs nomhyp.c2 with nomlemme.1
    [] /\  subgoal
    <> /\ subgoal
    ⋮
    <> /\ subgoal
    critical-pairs nomhyp.cn with nomlemme.1
    [] /\  subgoal
  [] conjecture
qed
```

En utilisant directement les hypothèses et le lemme, on factorise les sous-preuves en une preuve unique :

```
prove f(i1) /\ f(i2) ... /\ f(in)
    critical-pairs nomhyp.c1:cn with nomlemme.1
  [] conjecture
qed
```

27. Les lignes <>... et []... sont générées par LP. Pour chaque but considéré, le premier signe correspond au début de la preuve et le second correspond à la fin de la preuve.

où la notation `nom.i:j` de LP représente tous les faits de la base dont le nom est `nom` et le numéro est compris entre $i$ et $j$ inclus.

Cette factorisation des preuves est d'autant plus importante que la formule à montrer est complexe et que la stratégie à utiliser dans chaque sous-preuve comprend un grand nombre de commandes.

### 5.4.4 Le calcul de paires critiques

Le calcul de paires critiques est l'une des méthodes les plus utilisée pour une base de règle de taille importante. Outre son utilisation pour vérifier que la base ne contient pas d'axiomes contradictoires, elle permet de déduire les faits dont nous avons besoin mais pour lesquels on ne connaît pas la stratégie à utiliser pour les obtenir, en particulier quel axiome appliquer et à quelle hypothèse. En effet, le calcul de paires critiques, utilisé comme méthode de preuve et donc invoqué dans le contexte d'une preuve d'un but donné, s'arrête dès qu'une paire critique permettant de simplifier le but à `true` est générée[28]. Néanmoins ce calcul peut être très long, notamment sur un ensemble de règles de taille importante. Il existe alors quelques stratégie à adopter pour minimiser le coût de ce calcul.

Associer un nom à un ensemble de faits est l'un des moyens possible pour réduire ce coût. En particulier, il permet le calcul de paires critiques entre des sous-ensembles de règles spécifiés par les paramètres de la commande de calcul. Ainsi, lorsqu'on a plusieurs hypothèses dans un contexte courant et que l'on ne sait pas quelle est l'hypothèse qui permet de résoudre la but, il suffit de calculer les paires critiques entre l'ensemble des hypothèses courantes, spécifié par `Hyp*` et un sous ensemble de la base de règles.
Parfois un seul calcul de paires critique ne suffit pas pour générer le fait dont nous avons besoin, mais l'une des paires critiques, utilisée avec le même sous ensemble de règle le permet. Dans ce cas, il est préférable de changer le nom du contexte avant d'invoquer le premier calcul de paires critiques, afin que le deuxième ne se fasse qu'avec les paires critiques générées par le premier. Ce processus correspond à la structure suivante :

```
set name nom_preuve
prove ...
  ⋮
  set name les_paires_critq
  critical-pairs *hyp with nom_ens_regles*
  critical-pairs les_paires_critq* with nom_ens_regles*
  ⋮
```

Utiliser plusieurs calculs de paires critiques peut se ramener à invoquer une complétion. Néanmoins, la complétion utilise toutes les règles de réécriture de la base, qui généralement comprend plusieurs sous bases qui spécifient des théories incomplètes. Par exemple, on a défini *les entiers naturels* dont la base de faits correspondante est `Nat*`, puis on a défini les *séquences*, dont la base de faits correspondante est `seq*`. Supposons que l'on est entrain de prouver une propriété sur les séquences, et que l'on a besoin d'un calcul de paires critiques, alors si on demande une *complétion* avec la commande `complete`, le processus de complétion peut "s'engager" dans le calcul de paires critiques de la base `Nat*` et peut ne pas s'arrêter.

28. On rappelle que lors d'un calcul de paires critiques, le processus utilisé par LP consiste à simplifier tous les faits présents dans le système par la paire critique générée avant de calculer la suivante (Chapitre 1).

Enfin, la règle `critical-pair`, qui calcule les paires critiques, s'applique quand l'axiome dont on a besoin ne s'utilise pas dans le sens de la simplification (gauche-droit) mais plutôt dans le sens contraire (droite-gauche). Le calcul des paires critiques a pour effet de forcer cette utilisation.

### 5.4.5 Les informations sur l'orientation

Il est très fréquent que l'on introduise un ensemble d'équations et que l'on soit persuadé qu'elles seront orientées dans le sens attendu, puis que l'on s'aperçoive que certaines équations ont été inversées. Ceci est peut être du aux informations contenues dans le registre et manipulées par LP pour orienter les équations ou bien lorsque le démonstrateur a le choix, c'est à dire que les deux sens sont possibles, il choisit toujours un sens de simplification ou de réduction, du membre de l'équation le plus complexe vers le plus simple.
Les situations les plus fréquentes correspondent à la définition structurelle ou conditionnelle des opérateurs.

$t4$ = `if` $t1$ `then` $t2$ `else` $t3$

si $t_4$ ne commence pas par un `if`.

Une définition structurelle est très utile pour définir des opérations sur les objets de type *liste* de la forme

`f(x,y,cons(`*premier*`,`*reste*`))` = `f(x,y,sing(`*premier*`))` /\ `f(p,q,`*reste*`)`

Enfin la troisième situation est l'utilisation des fonctions d'arité 0, c'est à dire des constantes que l'on définit. Dans ce cas, l'orientation d'une équation de la forme `Cst = 0`, où `cst` est une constante entière, peut être `0 -> Cst`, due au fait que LP choisit une précédence quelconque entre les deux symboles `0` et `Cst`[29]. Cette orientation n'est pas très utile car le symbole `0` sera remplacé par `Cst` dans tous les faits courants.

Néanmoins, cette situation, qui dans une preuve par cas est très gênante, fréquente dans la précédente version du démonstrateur a été minimisée dans la version actuelle.

Deux solutions sont possibles pour remédier à ces situations :

- Soit on recommence le processus en modifiant l'ordre par défaut, en proposant des informations sur la précédence et le statut, de telle sorte que les équations que l'on introduira soient orientées dans le bon sens (sous réserve que ces modifications soient acceptées, c'est à dire qu'elles sont compatibles avec les informations contenues dans le registre).

  **Exemple 5.3** *Pour les exemples précédents, les informations à introduire sont*

  ```
  register height f > /\
  register height cst > 0
  ```

- Soit on utilise l'ordre interactif `manual`, où pour chaque équation introduite, LP demande le sens de l'orientation, ou bien l'ordre "gauche-droite" (paragraphe 1.3.4).

  **Exemple 5.4** *Pour les règles conditionnelles précédentes, on demande généralement l'orientation de gauche à droite.*

  ```
  set ordering left-to-right
  ```

29. Si l'ordre utilisé est noeq-dsmpos, où l'égalité entre les symboles n'est pas possible.

*puis on re-positionne l'ordre à* `noeq-dsmpos` *après l'introduction de toutes les équations.*

Ces possibilités sont à utiliser de façon rigoureuse car elles peuvent engendrer un système de réécriture non confluent, et il est préférable d'essayer de modifier le registre, car ces modifications demeurent sous le contrôle du démonstrateur puisque il vérifie que ces informations ne contredisent pas celles déjà présentes dans le registre.

### 5.4.6 Preuve par cas

Une preuve par cas est à utiliser lorsque on peut simplifier la preuve d'une conjecture en procédant par cas. Une des exigences est que les cas soient exhaustifs, c'est à dire que la liste des cas que l'on propose au démonstrateur couvre bien tous les cas possibles.

Les preuves par cas sont très utiles lorsque la forme de la conjecture ne nous permet ni une induction, ni l'utilisation d'une méthode syntaxique. En effet, si une hypothèse de la base de faits du contexte courant est sous forme d'une disjonction et non la conjecture, alors on procède par cas. De même, si une hypothèse contient une implication, on procède aussi par cas, en essayant de montrer que lorsque l'antécédent de l'hypothèse est faux, le but se réduit à un axiome de la base de faits.

### 5.4.7 Une stratégie de preuve classique

En résumé, un preuve classique d'une conjecture comprenant deux entiers est la suivante: la preuve débute par une induction sur le premier, où la règle d'induction est choisie suivant le principe décrit précédemment (cf. p. 155), puis on procède par induction sur la deuxième variable, et ceci pour le cas de base ainsi que pour l'étape d'induction. Alors, pour chaque sous cas, si la conjecture n'est pas triviale, étudier la forme de la conjecture et appliquer une des tactiques suivantes:

- Si la conjecture contient un opérateur logique alors utiliser la méthode syntaxique correspondante.
- Si la conjecture est sous forme d'une négation alors on poursuit par contradiction puis par calcul de paire critique pour déduire une inconsistance. Si ce calcul ne suffit pas (très rare) pour générer l'inconsistance alors appliquer une complétion.
- si une des hypothèses est sous la forme d'une disjonction ou d'une implication alors utiliser une preuve par cas.
- Sinon, si la conjecture n'est pas sous forme de négation et ne contient pas d'opérateur logique, ni de variable libre alors demander un calcul de paires critiques entre les hypothèses et la base courante.

**Exemple 5.5** *La structure d'une telle preuve est la suivante ou la méthode syntaxique choisie est l'équivalence:*

```
prove P(i,j) by induction on i using regle1_ind
    <> basis subgoal
   resume by induction on j using regle1_ind
      <> basis subgoal
      resume by contradiction
        <> contradiction subgoal
```

```
        cr-p *hyp with nat*
        [] contradiction subgoal
      [] basis subgoal
      <> induction subgoal
      [] induction subgoal
    [] basis subgoal
    <> induction subgoal
    resume by induction on j using regle1_ind
      <> basis subgoal
      resume by <=>
        <> <=> subgoal
        critical-pairs *hyp with nat*
        [] <=> subgoal
        <> <=> subgoal
        critical-pairs *hyp with nat*
        [] <=> subgoal
      [] basis subgoal
      <> induction subgoal
      [] induction subgoal
    [] induction subgoal
  [] conjecture
qed
```

*On peut remarquer que l'interaction survient la plus part du temps pour les cas de base de l'induction alors que pour les étapes d'induction, le sous but est résolu automatiquement par* LP *en utilisant l'hypothèse d'induction.*

### 5.4.8 Preuves paramétrées

Bien que l'objet *preuve* n'existe pas en LP et que LP ne fournisse pas de méta-langage pour écrire des tactiques, on peut tout de même simuler la conception et l'utilisation de preuves paramétrées.

En effet, une stratégie de preuve à utiliser lorsque la forme de la conjecture ne nous permet pas de décider d'une méthode de preuve, est la suivante :

```
set name lemme
critical-pairs *hyp with spec*
critical-pairs lemme* with spec*
```

Cette sous-preuve, ou suite de commandes, peut être vue comme une preuve paramétrée dont le paramètre est le nom de la base *spec* et éventuellement le nom à donner aux paires critiques, ici *lemme*. Elle consiste à déduire les conséquences de l'introduction des hypothèses courantes avec la base de faits, et ceci en utilisant deux calculs de paires critiques où le deuxième utilise les paires non triviales calculées par le premier.

Une deuxième preuve paramétrée est la preuve vue plus haut pour une conjecture sur deux variables qui peuvent être d'une même sorte ou de deux sortes différentes sur lesquelles des schémas d'induction ont été définis. Dans ce cas, la stratégie de preuve est une double induction et les paramètres sont les noms des variables libres et les noms des règles d'induction.

```
prove P(v1, v2) by induction on v1 using nom_regle1_ind
    <> basis subgoal
```

```
   resume by induction on v2 using nom_regle2_ind
      <> basis subgoal
         ⋮
      [] basis subgoal
      <> induction subgoal
         ⋮
      [] induction subgoal
    [] basis subgoal
    <> induction subgoal
    resume by induction on v2 using nom_regle2_ind
      <> basis subgoal
         ⋮
      [] basis subgoal
      <> induction subgoal
         ⋮
      [] induction subgoal
    [] induction subgoal
  [] conjecture
qed
```

Un troisième cas de preuve paramétrée est la preuve des schémas d'induction, où les noms des hypothèses sont générés par LP. De plus, le nombre d'hypothèses générées est égal au nombre de constructeurs de la règle à prouver. On peut donc considérer ces preuves comme étant paramétrées par le nom de la règle d'induction définie sur la sorte considérée pour montrer que cette sorte admet un deuxième schéma d'induction.

La structure d'une preuve d'un schéma d'induction *nom_regle_ind* qui définit trois générateurs $g1, g2, g3$ pour la sorte $S$ est :

```
prove Sort S generated by g1,g2,g3
    <> induction rule
    resume by induction using nom_regle_ind
      <> basis subgoal
      [] basis subgoal
      <> induction subgoal
      set name nom_lemme
      critical-pairs *GenHyp with *GenHyp
      critical-pairs *InductHyp with nom_lemme
      [] induction subgoal
    [] induction rule
  [] conjecture
qed
```

Enfin le dernier cas est la preuve d'une règle d'extension de la forme

```
Sort S partitioned by g1,g2,g3
```

traduite par LP en une règle de déduction et dont la preuve utilise une double induction sur les variables dont le nom est généré par LP. Par conséquent, la preuve est paramétrée par le nom de la sorte, la liste des opérateurs et le nom des variables et dont la structure est :

```
prove Sort S partitioned by g1,g2,g3
    <> deduction rule
```

```
    resume by ind on v1 using nom_regle_ind
      <> basis subgoal
      resume by ind on v2 using nom_regle_ind
        <> basis subgoal
        [] basis subgoal
        <> induction subgoal
        [] induction subgoal
      [] basis subgoal
      <> induction subgoal
      resume by ind on v2 using nom_regle_ind
        <> basis subgoal
        [] basis subgoal
        <> induction subgoal
        [] induction subgoal
      [] induction subgoal
    [] deduction rule
  [] conjecture
qed
```

**Remarque 5.1** *Bien sûr, la structure effective de ces preuves peut être légèrement différentes en raison d'une interaction possible nécessaire pour résoudre un sous cas. Néanmoins, cette interaction est généralement triviale, car on a pu le remarquer, l'interaction est généralement requise pour les cas de base. Pour les étapes d'induction* LP *utilise de façon automatique l'hypothèse d'induction.*

### 5.4.9 Gestion des quantificateurs existentiels

Pour les spécifications abstraites, le quantificateur existentiel accroît l'expressivité du langage et peut être très utile. Il permet notamment un raisonnement plus intuitif. Néanmoins, la gestion de ces quantificateurs est complexe. En résumé, pour utiliser une hypothèse qui comprend deux quantificateurs existentiels, nous avons utilisé la stratégie suivante : ayant les hypothèses

$H1 : \exists v Q(v) \Rightarrow \exists r P(r)$

$H2 : Q(d)$

et le but

$\exists x P(x)$

Nous commençons par montrer la propriété $\exists v Q(v)$ en spécialisant la variable $v$ à la constante $d$. Nous obtenons alors le fait $\exists r P(r)$, pour lequel nous choisissons une constante de skolem, $skr$, pour fixer la variable $r$. Nous obtenons le fait $P(skr)$. Pour conclure, il suffit de spécialiser la variable $x$ du but à $skr$. La preuve correspondante est :

```
prove \E x P(x)
  set name lemme
     prove \E v Q(v)
       resume by specializing v to d
  declare operator skr: -> S
  fix r as skr in Hyp
  resume by specializing x to skr
qed
```

Cette gestion des quantificateurs est importante pour la validité des preuves effectuées, néanmoins ces preuves doivent être intuitives dans le sens où la démarche utilisée ne doit pas aller à l'encontre du raisonnement effectué pour les preuves manuelles.

### 5.4.10 Utilisation des lemmes

De façon générale, l'utilisation abusive des clauses **assert** dans les preuves, pour en hâter la conclusion, annule leur validité. En effet, si nous introduisons des affirmations ("gratuites"), nous n'avons aucune garantie d'un résultat correct. Par conséquent, nous déconseillons d'introduire des clauses **assert** dans les preuves de façon définitive (ou systématique). Néanmoins, lors de la vérification d'une preuve, un lemme peut s'avérer nécessaire, qui spécifie généralement une propriété que l'on a pas encore vérifiée ou bien qui n'existe pas encore dans notre base de faits et dont on espère que l'introduction conduira à la conclusion. La stratégie consiste à l'introduire comme un axiome, puis à poursuivre la preuve. Une fois la preuve achevée, il faut reprendre les lemmes introduits pour les vérifier puis réécrire la preuve sans les **assert**. Une autre stratégie consiste à sauvegarder le statut courant de la preuve, à prouver les lemmes puis à reprendre la preuve après avoir eu confirmation de la validité des lemmes utilisés.

Il est crucial pour la certification des preuves que les propriétés utilisées soient correctes (relativement à la base de faits). Cette situation est fréquente car elle permet les "raccourcis", la propriété manquante paraissant triviale, par exemple une propriété sur les entiers ou sur les listes. Pour cela, on l'introduit comme axiome. Néanmoins, ces propriétés relèvent généralement de l'intuition lorsqu'il s'agit de traiter les cas de base, $n = 0$, $l = nil$,etc. Or notre expérience sur la vérification de preuves complexes nous a montré que c'est précisément sur ces cas de base que repose la validité de la propriété à prouver.

### 5.4.11 L'immunité et la passivité

Lors de la vérification d'une preuve, dont on a établi au préalable la structure, il arrive que LP ne puisse terminer la preuve alors que les lemmes dont il a besoin ont été prouvés et introduits dans la base.
Mis à part la situation où la preuve est incorrecte, ce blocage peut être dû à la normalisation automatique de la conjecture à prouver et des faits présents dans la base.

Un exemple simple est le suivant, relatif aux expressions booléennes : soit une règle d'inférence, formalisée en implication :

`R: (p \=> q) => f(p,q)`

Supposons qu'on ait montré le fait

```
P1: (a ^ b) \=> ((a ^ b ^ c) \or (a ^ b ^ d))
```

et que l'on veuille prouver

```
P2: f(a ^ b, ((a ^ b ^ c) \or (a ^ b ^ d))
```

Il suffit d'appliquer la règle d'inférence `R`. Néanmoins, la formule `P1` est simplifiée en

```
Ps:nnot(a) \or nnot(b) \or c \or d
```

La règle `R`, dont le membre droit est aussi simplifié en `nnot(`$p$`) \or` $q$, n'est pas directement applicable sur `Ps`. Certes, l'exemple présenté est simple, néanmoins cette situation est possible quelque soit la complexité des formules, d'où la difficulté à cerner le problème.

Trois solutions sont possibles. La première consiste à immuniser `P1` avant d'essayer de la prouver. En effet, l'immunité permet de récupérer la formule sous sa forme originale une fois sa preuve effectuée. On obtient la structure de preuve suivante :

```
set immunity on
```

```
prove (a ^ b) \=> ((a ^ b ^ c) \or (a ^ b ^ c))
  :
qed
set immunity off

prove f(a ^ b, ((a ^ b ^ c) \or (a ^ b ^ c))
  instantiate
    p by a ^ b, q by ((a ^ b ^ c) \or (a ^ b ^ c)) in R
qed
```

Une deuxième solution consiste à prouver, dans la preuve de `P2`, la "forme immunisée" de la propriété `P1`, que LP résout de façon triviale car la propriété a été prouvée précédemment et introduite comme un fait dans la base. Par conséquent, LP ne fait que la réécrire en sa forme normale, qui est une formule valide dans le système. La structure de ce processus est :

```
set name P1
prove (a ^ b) \=> ((a ^ b ^ c) \or (a ^ b ^ c))
  :
qed
:
%P1 est sous la forme nnot(a) \or nnot(b) \or c \or d
:
prove f(a ^ b, ((a ^ b ^ c) \or (a ^ b ^ c))
  set immunity on
  prove (a ^ b) \=> ((a ^ b ^ c) \or (a ^ b ^ c))
  set immunity off

  instantiate
    p by a ^ b, q by ((a ^ b ^ c) \or (a ^ b ^ c)) in R
qed
```

Enfin, la troisième solution consiste à dés-activer les règles de réécriture qui simplifient la propriété `P1` en `Ps`. Néanmoins cette solution complique les preuves, car les règles de simplification sont très utiles, et une fois dès-activées, leur utilisation doit être explicitement invoquée avec la commande **`rewrite ... with ...`**.

Le choix de la solution à adopter dépend principalement de l'utilité de la propriété `P1`. Si c'est une propriété très utilisée sous sa forme originale dans l'ensemble de la preuve, la première solution est plus adéquate que la seconde, car elle évite de refaire le même traitement dans chaque preuve où la propriété est nécessaire. Par contre, si son utilité est ponctuelle, la deuxième solution peut être utilisée.

De façon générale, l'immunité et la passivité permettent à l'utilisateur de contrôler l'application de certaines règles problématiques, pour effectuer des preuves complexes. Elles permettent d'améliorer les performances du système, en immunisant les faits irréductibles ou bien en dés-activant les faits qui ne peuvent être appliqués. Néanmoins, ces possibilités doivent être utilisées de façon rigoureuse car elles peuvent engendrer des systèmes non confluents et/ou ralentir le système avec un nombre considérable de règles de réécriture redondantes, et par conséquent diminuer les performances du système à cause du processus de normalisation.

## 5.5 Conclusion

La méthodologie que nous avons décrite dans ce chapitre repose essentiellement sur notre expérience dans la mise en oeuvre et l'utilisation des techniques de vérification à l'aide du démonstrateur LP. Bien sûr, nous ne prétendons pas être exhaustifs ni garantir à tout utilisateur de ces méthodes une réussite certaine sans embûches. Néanmoins, ces méthodes peuvent faciliter l'utilisation des techniques formelles, pour spécifier et résoudre un problème particulier, et pour vérifier la correction de la solution.

Par ailleurs, l'utilisation d'un démonstrateur permet, outre l'acquisition d'une expertise dans son utilisation propre, de comprendre et de se convaincre de l'utilité de la vérification assistée. En particulier, traduire une spécification informelle en une spécification formelle, structurer une preuve en sous-buts et organiser le travail en étapes pour résoudre ces sous-buts ne sont pas des activités propres au démonstrateur utilisé. De plus, la mise au point des preuves et la gestion des évolutions nous semblent cruciales dès le moment où l'on a décidé de valider une application.

Spécifier les objets de base sur lesquels porte le raisonnement, décomposer le problème en sous problèmes, vérifier chaque étape de vérification et enfin documenter la certification sont des activités nécessaires quel que soit l'outil utilisé pour la vérification.

# Deuxième partie

# Les études de cas

# Avant-propos

Cette deuxième partie est consacrée aux études de cas. Nous illustrons à travers trois programmes UNITY, la vérification mécanisée à l'aide du démonstrateur LP. Nous décrivons des problèmes de différentes complexités et nous illustrons l'utilisation des méthodes formelles de vérification lorsqu'on s'intéresse à des problèmes conséquents, afin de détecter des erreurs dans la spécification initiale ou dans la preuve informelle. Pour chaque programme, nous donnons la spécification, les étapes de développement, les preuves informelles, ainsi que la description des principales étapes de la preuve mécanisée.

Nous commençons par la vérification d'un *protocole de communication à travers des canaux défectueux*, décrit en UNITY. La spécification avec LP et la vérification mécanisée de ce protocole ne sont pas décrites dans leur totalité, à cause de la taille des preuves. En effet, la preuve *complète* comprend un nombre important d'obligations de preuves, dont trois invariants pour la sûreté. La propriété de vivacité, quant à elle, se décompose en dix huit étapes. Par contre, nous allons décrire la formalisation du protocole, et les principales étapes de la preuve informelle et de la preuve mécanisée. Nous montrons comment l'utilisation d'un démonstrateur nécessite la vérification des hypothèses ou suppositions basées sur des arguments intuitifs et que la complexité et le nombre de propriétés à vérifier fait que les preuves manuelles, même détaillées, ne sont pas suffisantes pour s'assurer de la correction. Par contre, l'utilisation d'un outil d'aide à la preuve nécessite la formalisation et la vérification de ces suppositions et permet de détecter des imperfections ou imprécisions dans la spécification [Che95b].

La deuxième étude de cas décrit le problème classique des *lecteurs-rédacteurs*. Nous illustrons à travers cet exemple ce que peut être une preuve mécanique par rapport à une preuve informelle structurée, et nous montrons qu'il ne suffit pas d'écrire la structure générale d'une preuve pour la vérifier mécaniquement. Par ailleurs, cet exemple nous permettra d'illustrer des aspects relatifs à l'ensemble de nos preuves, à savoir la réutilisation des preuves et le niveau de détails requis pour les preuves mécanisées [Che95a].

Enfin, la dernière application consiste à vérifier un *protocole de contrôle d'un ascenseur* et cet exemple nous permet une comparaison avec d'autres démonstrateurs traitant des preuves UNITY. La preuve originale ayant été effectuée avec le démonstrateur d'ordre supérieur HOL, il nous semblait intéressant de vérifier cette même preuve avec un démonstrateur du premier ordre. Nous décrirons les principales étapes de la vérification d'une propriété de sûreté et nous montrerons que les structures des preuves sont similaires dans les deux approches et enfin nous mettrons en évidence le caractère semi-automatique des preuves de sûreté.

Ces études de cas ont été menées avec la version 2.4 de LP et sont actuellement en cours de vérification avec la version 3.1. Pour une estimation approximative, la preuve des lecteurs-rédacteurs (la plus simple des trois) comprend une vingtaine de propriétés "principales" à vérifier et son temps d'exécution est de 15 minutes sur une station serveur *alpha* (bi-processeur 128 M de mémoire, 275 Mhz) et de 30 minutes sur une station *Sun4* (64 M de mémoire).

De part leurs complexités et leurs tailles, et leurs évolutions permanentes, ces preuves ne

peuvent être encore être mises à la disposition d'un large public, mais nous espérons qu'elles le soient dans un futur proche.

# Chapitre 6

# Un protocole de communication à travers des canaux défectueux

## 6.1 Introduction

Vérifier la correction d'un programme parallèle consiste essentiellement à prouver des propriétés de **vivacité** et de **sûreté.** Les propriétés de sûreté sont d'autant plus cruciales que le problème traité est complexe. Pour un protocole de communication, il est important de montrer que les données échangées ne soient pas perdues en cours de transmission, et que les données soient reçues dans l'ordre d'émission, ce sont donc des propriétés de sûreté. Les propriétés de vivacité, quand à elles, nous assurent qu'il y a bien un échange de données entre l'émetteur et le récepteur, ou que ces derniers ne se trouvent jamais dans une situation d'*inter-blocage.* Le protocole de communication que nous allons traiter spécifie un échange de données entre un émetteur et un récepteur reliés par un canal de transmission défectueux, c'est à dire que les données peuvent être perdues (respectivement dupliquées), mais un nombre fini de fois (non indéfiniment). Le but de la preuve est alors de montrer que malgré la nature défectueuse du canal de transmission, la communication est correcte.

Ce chapitre décrit la vérification d'un protocole de communication à travers des canaux défectueux. Plus précisément, nous décrivons la formalisation du problème ainsi que la mécanisation de la vérification avec le démonstrateur de théorèmes du LARCH. Nous montrons comment utiliser la méthodologie du démonstrateur pour prouver les propriétés de *sûreté* et de *vivacité* d'un protocole de communication et en particulier, comment un tel outil peut être utilisé pour détecter des imperfections ou imprécision dans la spécification d'un système. Pour cela, nous présentons les principales étapes de la preuve, en mettant l'accent sur les imprécisions détectées, les solutions choisies, et la mécanisation en LP. Le but est de montrer qu'on peut tirer avantage à la fois de la simplicité et de la puissance du démonstrateur et d'UNITY et d'illustrer l'utilisation de ces méthodes axiomatiques afin de corriger une spécification, même simple, présentée dans des textes.

Le protocole étudié dans ce chapitre est tiré du chapitre 17 du livre de Chandy et Misra [Cha88]. Le protocole en question est une simplification du protocole de *la fenêtre coulissante* de TCP dans le cas où la fenêtre est de taille 1 [Cor92]. Malgré sa simplicité, il comprend toutes les difficultés et problèmes que peut présenter un protocole réel et nous permet d'illustrer les solutions que nous proposons. Notre but est de montrer le type d'imprécision que peut contenir une spécification, même celle d'un protocole de base.

Tout d'abord, pour les propriétés de sûreté, la mécanisation de la preuve nous a amené à

développer les plus bas niveaux de la spécification. Ce raffinement nécessaire nous a forcé à vérifier des assertions dont la preuve est omise ou laissé au lecteur, et comme prévu, nous avons trouvé là une sous estimation de la difficulté réelle de la preuve. Plus précisément, la preuve de Chandy et Misra est basée sur l'invariance de trois prédicats $I_1$, $I_2$ et $I_3$, représentant trois propriétés d'invariance que le protocole doit satisfaire. Par conséquent, l'invariance de chaque prédicat peut être prouvé indépendamment. Cette expérience avec LP a montré que l'invariance de $I_1$ requière celle d'$I_3$ (et réciproquement), plus celle d'un invariant supplémentaire. Cette erreur n'a pu être détectée dans la preuve manuelle, d'une part en raison du nombre d'obligations de preuve, et des détails à manipuler, et d'autre part, à cause de l'évidence apparente de la propriété nécessaire pour conclure. Par contre, la vérification mécanisée nécessite une formalisation rigoureuse et la certification des suppositions et des hypothèses prises, d'où la détection des imperfections qui auraient pu être fatales dans un véritable protocole.

Pour les propriétés de vivacité, nous avons mis en avant le fait que les étapes de bas niveaux laissées au lecteur concernent les aspects les plus obscurs du protocole, à savoir la communication des données à travers des canaux défectueux. De plus, cette expérience nous a montré que l'aspect méthodologique dans le développement de la preuve de vivacité ne peut en aucun cas être négligé.

Ce chapitre est organisé comme suit. Nous commençons par une brève description du protocole tel qu'il a été proposé par Chandy et Misra [Cha88], et de sa représentation en LP. Par la suite, nous présentons les étapes principales de la mécanisation de la preuve de correction. Nous expliquons les défauts détectés dans les preuves de sûreté et nous décrivons la stratégie utilisée pour prouver les propriétés de *progrès*. Nous conclurons ce chapitre par une discussion sur cet expérience.

## 6.2 Un protocole de communication à travers des canaux défectueux

Le protocole de communication que nous décrivons constitue une étude de cas proposée par Chandy et Misra [Cha88]. Le canal défectueux et le protocole de communication sont représentés par des programmes de manière à ce que la conception d'un protocole de communication à travers des canaux défectueux soit vue comme une composition de programmes. L'union du programme représentant le protocole et de celui représentant le canal de transmission défectueux est un programme qui garantit une communication correcte. Dans ce qui suit, seuls les concepts de base sont étudiés.

### 6.2.1 Présentation du problème

Un processus, appelé *émetteur* a accès à une séquence infinie de donnée *ms*. Un autre processus, le *récepteur* doit délivrer une séquence *mr* satisfaisant la spécification suivante, où $|mr|$ est la longueur de *mr* :

**invariant** *mr is a prefix of ms*
$|mr| = n$ **leads_to** $|mr| = n + 1$

Ce sont les deux propriétés de base que doit vérifier toute transmission fiable sur un canal qui ne l'est pas. La première est une propriété de sûreté , à savoir que les données délivrées par le récepteur doivent avoir été transmises par l'émetteur. Néanmoins, un protocole qui ne délivre aucune donnée vérifie cette propriété, d'où le besoin d'une propriété de vivacité: les données

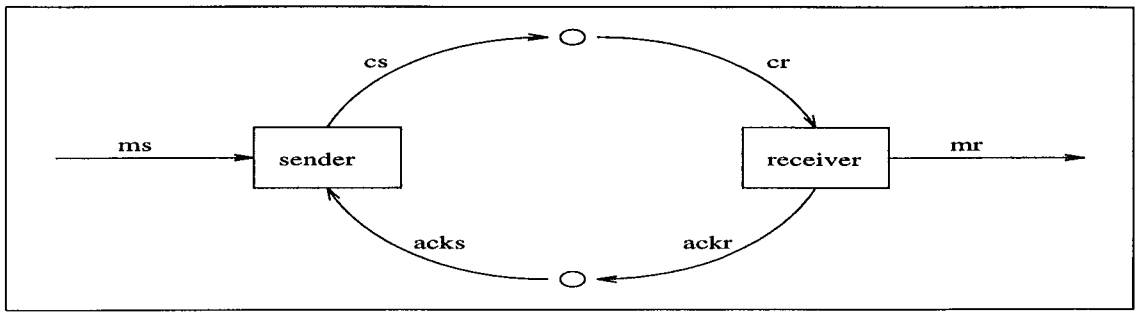



FIG. 6.1 – *Les composants du protocole de communication*

émises doivent être reçues tôt ou tard par le récepteur.

L'émetteur et le récepteur communiquent à travers un canal modélisé par deux paires de canaux de transmission non défectueux $(cs, cr)$ et $(ackr, acks)$, et par un programme (représentant le comportement défectueux du mode de transmission) manipulant le contenu de $cs$ et $cr$ (Fig 6.1). Ce canal pourrait être INTERNET et son protocole IP qui, comme l'on sait, peut perdre des messages. L'émetteur ajoute des messages à $cs$ et reçoit des acquittements de $acks$. Le récepteur reçoit les messages de $cr$ et transmet les acquittements par $ackr$. L'émetteur transmet par $cs$ une donnée de $ms$ et son index $ks$, donc un message est un couple $(ks, ms[ks])$. Par conséquent, à tout moment, $ks$ représente le numéro du dernier message émis par l'émetteur. D'une façon similaire, le récepteur reçoit de $cr$ une paire $(n, d)$. Il ajoute alors la donnée $d$ à $mr$ et envoie $n$ comme acquittement (utilisant le canal $ackr$). A chaque moment, $kr$ est le numéro du dernier message reçu par le récepteur. L'émetteur n'envoie un nouveau message que s'il a reçu un acquittement du précédent.

### 6.2.2 Spécification du canal de transmission défectueux

Il est important de spécifier un canal de transmission défectueux de telle façon que la perte ou la duplication des messages ne soit pas sous le contrôle de l'émetteur ou du récepteur mais se produit de façon autonome. Exactement comme IP, qui est le protocole "défectueux" de transmission des messages et TCP, le protocole au dessus de IP qui assure une transmission correcte.

Les propriétés de **sûreté** du canal garantissent que le canal défectueux n'a accès qu'au message situé en tête de $cs$ et ne peut transmettre un message qu'en l'ajoutant à la queue de $cr$; De plus, un canal défectueux peut perdre des messages de $cs$ et/ou dupliquer des messages dans $cr$. Une propriété de **progrès** d'un canal défectueux garantit que si un message est ajouté infiniment à $cs$, il finira par apparaître dans $cr$. Cette propriété est spécifiée à l'aide d'une *propriété conditionnelle*[30] :

**COND_Prop :**
*Hypothèses*:

$$p.m \wedge p.n \Rightarrow m = n \quad (1)$$

$$p.m \wedge cs = null \ \textbf{unless} \ \neg p.m \vee cs = \ll m \gg \quad (2)$$

30. Un propriété conditionnelle est composée d'un ensemble d'hypothèses et d'une conclusion. Sa sémantique est que le programme vérifie la conclusion si ayant les hypothèses, on peut la montrer la conclusion à partir du texte ou de la spécification du programme

$p.m \wedge (cs = cs^0) \wedge (cs \neq null)$ **unless** $\neg p.m \vee (cs = tail(cs^0)) \vee (cs = cs^0; m)$ (3)
$p.m \wedge (|\overline{cs}| = k)$ **leads_to** $\neg p.m \vee (|\overline{cs}| > k)$ (4)

*Conclusion*:

$p.m$ **leads_to** $\neg p.m \vee (m \in cr)$ (5)

où $\overline{s}$ représente la séquence rajoutée à la séquence $s$ et $p.m$ est un ensemble de prédicats en $m$ tel qu'aucun autre message que $m$ ne peut être ajouté à $cs$ tant que $p.m$ est valide.

Cette spécification exprime les propriétés suivantes :

- (1): A tout moment, $p.m$ est valide pour une seul message $m$.
- (2): Dès que $p.m$ est valide et la séquence $cs$ est vide, alors ils le restent à moins que $p.m$ ne le soit plus ou bien que $m$ apparaisse dans $cs$.
- (3): Dès que $p.m$ est valide et la séquence $cs$ est non vide, alors ils le restent à moins que $p.m$ ne le soit plus ou que $m$ apparaisse dans $cs$, ou bien l'élément en tête de $cs$ est retiré.
- (4): Dès que le programme atteint un état où $p.m$ est vrai et où la taille de la séquence rajoutée à $cs$ est égale à $k$, alors fatalement il atteindra un état dans lequel $p.m$ est non valide, ou bien la taille de la séquence rajoutée à $cs$ est supérieur à $k$.
- (5): Dès que $p.m$ est vrai, il est soit mis à faux soit $m$ apparaît dans $cs$.

Chandy et Misra donnent le programme UNITY, *FC*, qui représente le canal défectueux, et la preuve de sa correction (c'est à dire qu'il satisfait la spécification précédente). Cette démonstration n'est pas nécessaire pour l'étude des protocoles de communication conçus sur ce canal; seule sa spécification est requise pour développer et vérifier ces protocoles. Nous supposerons alors la correction de la preuve du canal défectueux et nous introduirons sa spécification comme des axiomes dans nos preuves. Pour reprendre notre paradigme INTERNET, nous nous appuyons sur une spécification de type IP, pour prouver la correction d'un protocole de type TCP.
Il est important de noter que cette étude ne porte pas sur la correction de la formalisation (modélisation) du protocole de communication et du canal. Il s'agit d'étudier la correction du protocole dans le contexte d'UNITY, et s'inscrit particulièrement dans la vérification d'une preuve de correction. Les lecteurs intéressés par cette modélisation peuvent toujours se reporter à [Cha88]. En fait, nous focalisons notre étude sur la correction du programme qui spécifie le protocole afin de vérifier la preuve proposée, en considérant le programme *FC* comme une "boite noire" dont on connaît juste la propriété de sûreté et la propriété de progrès. Dans une approche fondée sur une pile de protocoles, cela signifie que nous prouvons la correction d'un protocole de niveau $n+1$ par rapport à un protocole de niveau $n$.

### 6.2.3 Spécification du protocole de communication

Un protocole de communication *fiable* est un programme qui satisfait les contraintes d'une transmission correcte lorsqu'il est composé avec le programme *FC* qui satisfait les contraintes d'un canal défectueux.

**Program** {Proto}
**Declare** $ks, kr$ : *integer*
**initially**
$ks, kr = 1, 0 \,[]\, cs, cr, acks, ackr = null, null, null, null \,[]\, mr = null$
**assign**

| | | | | |
|---|---|---|---|---|
| $\{transmit\}$ | $cs$ | $:=$ | $cs; (ks, ms[ks])$ | |
| $[]\{rcv\_ack\}$ | $ks, acks$ | $:=$ | $ks + 1, tail(acks)$ | $if\ acks \neq null \wedge ks = head(acks)$ |
| $[]\{rmv\_old\_ack\}$ | $acks$ | $:=$ | $tail(acks)$ | $if\ acks \neq null \wedge ks \neq head(acks)$ |
| $[]\{receive\}$ | $kr, mr, cr$ | $:=$ | $kr + 1, mr; head(cr).val, tail(cr)$ | $if\ cr \neq null \wedge kr \neq head(cr).dex$ |
| $[]\{send\_ack\}$ | $ackr$ | $:=$ | $ackr; kr$ | |
| $[]\{rmv\_old\_msg\}$ | $cr$ | $:=$ | $tail(cr)$ | $if\ cr \neq null \wedge kr = head(cr).dex$ |

**end**{Proto}

Ce programme est légèrement différent de celui donné dans [Cha88], en ce que nous n'avons pas formalisé les affectations parallèles ||. Les deux programmes sont néanmoins équivalents. Les trois premières instructions constituent le processus émetteur et les trois suivantes, le processus récepteur.

La première action, *transmit*, du processus émetteur, émet un nouveau message en ajoutant la paire $(ks, ms[ks])$ au canal de transmission $cs$. La seconde, *rcv_ack*, modélise la réception d'un acquittement, la valeur contenue en tête du canal des acquittements est égale au numéro du dernier message émis ($ks = head(acks)$). L'action, *rmv_old_ack*, prend en charge la re-transmission : étant donné que ($ks \neq head(acks)$) (ce n'est pas un acquittement du message dont le numéro est $ks$), la valeur de $ks$ n'est pas modifiée et l'élément en tête du canal d'acquittement est supprimé (car c'est un ancien acquittement)[31].

La première action, *receive* dans le processus récepteur, correspond à la réception d'un nouveau message. Le numéro du message en tête du canal de transmission diffère du numéro du dernier message reçu ($kr \neq head(cr).dex$). Le récepteur délivre un message en retirant l'élément en tête du canal de transmission et en ajoutant la donnée reçue à la séquence de donnée $mr$. L'action *send_ack* modélise l'envoi de l'acquittement et la dernière, *rmv_old_msg*, le retrait du message du canal de transmission si c'est un message qui a déjà été reçu ($kr = head(cr).dex$).

**Remarque:** Le protocole du *Bit Alterné* est une simplification de ce protocole, où les index des messages sont calculés modulo 2. Le protocole TCP est une généralisation où les index des messages doivent appartenir à un intervalle appelé "fenêtre".

31. Il faut noter que ce protocole n'utilise pas la notion de séquentialisation où la re-transmission d'un message ne se fait que si l'acquittement n'as pas été reçu après un certain temps fini. La demande de retransmission est indéterministe

**Correction du protocole:** Pour prouver la correction du protocole, les propriétés d'invariance et la propriété de progrès qui suivent doivent être satisfaites :

| | | | |
|---|---|---|---|
| **Invariant** | $\langle \forall y : y \in acks \vee y \in ackr :: y \leq kr \wedge ks \leq y+1 \rangle$ $\wedge$<br>$\langle \forall x : x \in cs \vee x \in cr :: kr \leq x.dex \leq ks \rangle$ $\wedge$<br>$kr \leq ks \leq kr+1$ | | (1) |
| **Invariant** | $\langle \forall x : x \in cs \vee x \in cr :: x.val \in ms \rangle$ $\wedge$<br>$mr$ *is a prefix of* $ms$ $\wedge$<br>$\mid mr \mid = kr$ | | (2) |
| **Invariant** | $cs.dex, cr.dex, acks, ackr$ *are* sequences *of* nondecreasing integers in Proto. | | (3) |
| $\mid mr \mid = n$ | **leads_to** | $\mid mr \mid = n+1$ | $(P)$ |

Le premier invariant contraint la valeur de la composante *entière* des messages dans le canal de transmission. Le second exprime le fait que tout message délivré par le récepteur a bien été envoyé par l'émetteur. Enfin le troisième donne une information sur l'ordre des messages dans les canaux.

## 6.3 Représentation avec LP

Dans ce qui suit, nous décrivons les principales étapes de la traduction du programme UNITY *proto* en LP. Le programme UNITY, *proto*, est représenté par une constante de type `Actlist`, à laquelle on affecte la liste des *actions* du programme de la partie *assign*.

```
declare operators
 proto :-> Actlist
..
assert
proto = cons(transmit,...,cons(rmv_old_msg,nil))
```

Une expression booléenne `cond_init` (constante de type `Bexp`) représente la section *Initially* du program *proto* :

```
cond_init \= (id_to_exp(kr) \= nat_to_exp(0))
          ^ (id_to_exp(ks) \= nat_to_exp(succ(0)))
          ^ (seq_to_exp(id_to_seq(ackr)) \= seq_to_exp(emptyseq))
          ....
```

Le terme `id_to_exp(kr) \= nat_to_exp(0)` représente la condition initiale $kr = 0$. La fonction `id_to_exp` convertit un élément de la sorte `Id` en un élément de la sorte `exp` et `id_to_seq` transforme un élément de la sorte `Seq` en un élément de la sorte `Exp` (cf. paragraphe 2.4.1). Le terme `(seq_to_exp(id_to_seq(ackr)) \= seq_to_exp(emptyseq))` formalise la condition initiale $ackr = null$.

Les variables d'états, `acks`, `ackr`, `cr`, ... etc, sont représentées comme des identificateurs de séquence de la sorte `id` , apparaissant dans le texte du programme. `yvar` et `xrec` sont des

identificateurs de variables de la sorte `id_of_var` nécessaires dans les obligations de preuves (cf. paragraphe 2.4.1).

**Exemple 6.1** *L'action rcv_ack est traduite en* LP *en utilisant la fonction* `cond_mult_assg(l,bexp)`, *qui représente une affectation conditionnelle multiple:*

```
transmit = cond_mult_assg({ks.(id_to_exp(ks) + nat_to_exp(1))}
                           \u ({acks.seq_to_exp(tail(id_to_seq(acks)))}\u {}),
                          nnot(id_to_exp(acks) \= seq_to_exp(emptyseq))
                          ^ (id_to_exp(ks) \= head(id_to_seq(acks))))
```

**Remarque:** Dans ce qui suit, nous n'allons pas écrire les fonctions de conversion. Par exemple, une expression de la forme

`id_to_exp(ks)\= head(id_to_seq(acks))`

sera écrite

`ks \= head(acks).`

Par ailleurs, nous utiliserons l'expression abrégée *un acquittement n* pour l'expression *un acquittement de valeur n* et *un message n* pour *un message dont l'index est égal à n.*

## 6.4 Obligations de preuve: Sûreté

Les obligations de preuves décrivent les propriétés que le protocole doit vérifier. Pour les propriétés de sûreté, ce sont les spécifications équationnelles des trois invariants $I_1$, $I_2$, $I_3$.

### 6.4.1 Invariant 1

Pour analyser les propriétés que le protocole doit vérifier, nous allons utiliser le compteur des messages envoyés ($ks$) et celui des messages reçus ($kr$) ainsi que les numéros des messages en transit.

- Le canal de transmission ne doit pas contenir un message qui n'a pas été transmis : $(x \in cr \vee x \in cs) \Rightarrow x.dex \leq ks$

- Le récepteur ne doit recevoir que des messages qui ont été envoyés : $(x \in cr \vee x \in cs) \Rightarrow kr \leq x.dex$

$\Rightarrow$ Le récepteur reçoit seulement ce qui a été envoyé : $(kr \leq ks)$

- Le canal d'acquittement ne doit pas contenir un acquittement d'un message qui n'a pas été reçu : $(y \in acks \vee y \in ackr) \Rightarrow y \leq kr$

- Le récepteur reçoit seulement ce qui a été émis : $(kr \leq ks)$

$\Rightarrow$ Les canaux d'acquittement ne peuvent contenir un acquittement d'un message qui n'a pas été envoyé : $(y \in acks \vee y \in ackr) \Rightarrow y \leq ks$

- Considérons maintenant la situation où un acquittement $y$ se trouve dans le canal des acquittements (Fig 6.2). Que peut être alors la valeur du dernier message envoyé, à savoir la valeur de $ks$? Cette valeur dépend de la "nature" de l'acquittement : s'il s'agit de l'acquittement d'un message bien reçu (ancien acquittement), la valeur de $ks$ est alors égale à $y+1$, sinon elle est égale à $y$. Comme l'émetteur n'envoie un nouveau message que s'il reçoit l'acquittement pour le dernier, c'est à dire un acquittement dont la valeur est égale à $ks$, $y$ est égale à $ks-1$ ou bien à $ks$ : $(y \in acks \vee y \in ackr) \Rightarrow ks \leq y+1$.

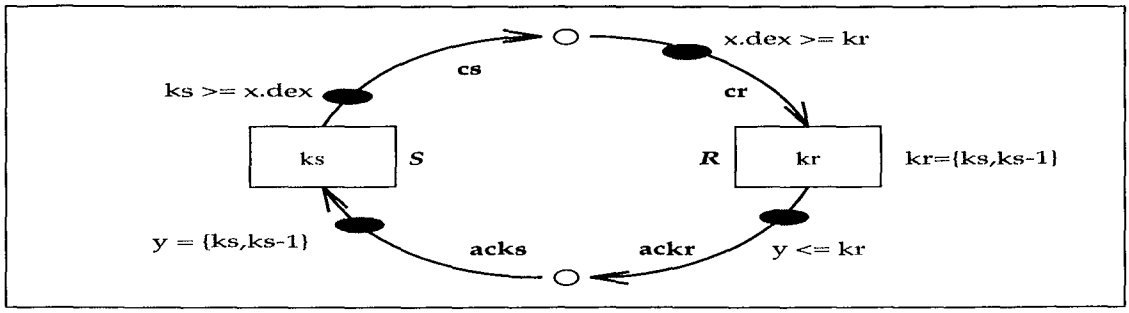



FIG. 6.2 – *Description de l'invariant 1*

- Enfin à tout moment, $kr$, le numéro du dernier message reçu, est égal au numéro du dernier message émis $ks$, ou bien à celui du précédent : $kr \leq ks + 1$.

Ces contraintes sont formalisées par l'invariant $I_1$ [32] :

$$\begin{array}{ll} \textbf{Invariant} & \langle \forall y : y \in acks \vee y \in ackr :: y \leq kr \wedge ks \leq y + 1 \rangle \quad \wedge \\ & \langle \forall x : x \in cs \vee x \in cr :: kr \leq x.dex \leq ks \rangle \quad \wedge \\ & kr \leq ks \leq kr + 1 \end{array}$$

### La preuve mécanisée en Lp

Pour prouver cet invariant dans notre système, nous devons prouver le théorème suivant en LP:

```
Inv(I1, proto, cond_init) = true
```

Pour formaliser le prédicat `I1`, nous avons défini le prédicat `inseq(e,seq)` qui teste si l'élément `e` appartient à la séquence `seq`. La quantification universelle de la variable $y$ (respectivement $x$) est implicite en utilisant une variable LP, `yvar`, (respectivement `xrec`)) de la sorte `id_of_var`.

**Exemple 6.2** *La première conjonction de l'expression booléenne* `I1` *est formalisée avec* LP *comme suit :*

```
(inseq(yvar,ackr) \or inseq(yvar,acks)) \=> ((yvar <= kr) ^ (ks <= (yvar+1)))
```

Montrer qu'un programme satisfait une propriété d'invariance *invariant* $P$ revient à montrer que $P$ est valide dans l'état initial et que toutes les actions du programme préservent $P$ : *init* $\Rightarrow P$ et toute action $s$ du programme doit satisfaire $\{P\}s\{P\}$.

Nous allons décrire la preuve en LP utilisant une seule action *rcv_ack*, c'est à dire en prouvant l'assertion $\{I1\}rcv\_ack\{I1\}$. Nous montrons que LP nous permet de construire des preuves dont la structure est identique à celle des preuves manuelles. Plus précisément, nous explicitons les étapes de preuves qui reposent sur des arguments intuitifs, non acceptés comme tels par le démonstrateur.

### Preuve de $I_1$ :

```
inv(I1, cons(rcv_ack,nil),cond_init) = true
```

32. "La propriété (1) spécifie l'invariance du prédicat $I_1$"

LP réécrit cette équation en utilisant les définitions de `inv` et de `unless`, en introduisant `wp` (cf. p. 61) :

```
(cond_init \=> I1) = T) /\ unless(I1, F, proto) = T)
→  (cond_init \=> I1) = T) /\ ((I1 \=> wp(rcv_ack,I1)) = T)
```

La commande `resume by /\-m` permet à LP de poursuivre par la méthode de conjonction (cf p. 24).

Le but est alors subdivisé en deux sous-buts :

```
(1.1): cond_init \=> (A ^ B ^ C) = T
```

```
(1.2): (A ^ B ^ C)
        \=> wp(cond_mult_assg({(ks,ks+1),(acks,tail(acks))},
                              (ks \= head(acks)) ^ nnot(acks \= emptyseq)),
               (A ^ B ^ C))
       = T
```

où $A$, $B$, $C$ sont respectivement le premier, second et troisième sous-terme de `I1` :

```
A :(inseq(yvar,ackr) \or inseq(yvar,acks))\=> ((yvar <= kr) ^ (ks <= (yvar +1)))
B :(inseq(xrec,cr) \or inseq(xrec,cs))\=> [(kr <= ind(xrec)) ^ (ind(xrec) <= ks))
C :((kr <= ks) ^ (ks <= kr+1))
```

**Preuve de** (1.1) : LP résout de façon automatique ce sous-but en utilisant la définition de `cond_init` représentant les conditions initiales. Il utilise la base de règles définissant les expressions de prédicats (cf. Chapitre 2) et les différents lemmes sur les séquences (cf. Chapitre 3), tel que `(inseq(exp1,seq1) \=> nnot(seq1 \=emptyseq)) = T` afin de réduire le but à `T=T`.

**Preuve de** (1.2) :

```
[((A ^ B ^ C) \=> wp(cond_assg((ks, acks:=ks+1, tail(acks)),
                               (ks \= head(acks) ^ nnot(acks \= emptyseq))),
                    (A ^ B ^ C)))] = T
```

LP applique alors la définition des substitutions, pour substituer `ks+1` à `ks`, `tail(acks)` à `acks` et normalise le but avec la base des règles définissant les opérateurs `<=`,`+`, etc :

```
(A ^ B ^ C) \=> [(((ks \= head(acks)) ^ nnot(acks \= emptyseq))
                  \=> ([inseq(yvar, ackr) \or inseq(yvar, tail(acks))]
                         \=> [(yvar <= kr) ^ (ks <= yvar)]
                       ^ [inseq(xrec, cr) \or inseq(xrec, cs)]
                         \=> [(kr <= ind(xrec)) ^ (ind(xrec) <= (ks+1))]
                       ^ [(kr <= (ks+1)) ^ (ks <= kr)]))
                ^ ((nnot(ks \= head(acks)) \or nnot(acks \= emptyseq))
                   \=> (A ^ B ^ C))]
   = T
```

Le but est normalisé à sa forme normale en utilisant la base des règles définissants les structures booléennes. La forme normale est une formule de la forme $C_1$ `/\` ... `/\` $C_n$ `= T`. Nous poursuivons alors par la méthode `resume by /\-m`. LP simplifie les sous-buts en utilisant la base de règles. Par exemple, un sous-terme de la forme :

```
A \or nnot(kr <= ks) \or (kr <= ks + 1) \or B
```

est normalisé à `T` à l'aide de l'axiomatisation des entiers naturels, la transitivité et les lemmes appropriés sur les *séquences*, tels que :

```
inseq(exp1,tail(seq)) \=> inseq(exp1,seq) = T
```

que nous avons prouvés et introduits dans la base de règle des séquences.

Néanmoins, LP ne peut terminer la preuve et s'arrête avec le sous-but :

```
[(inseq(yvar,ackr)\or inseq(yvar,tail(acks)))^ (yvar <= kr)^(ks <= yvar+1)
  ^ (kr <= ind(xrec)) ^ (ind(xrec) <= ks)^(kr <= ks)^(ks <= kr+1)
  ^ (ks \= head(acks)) ^ nnot(acks \= emptyseq)]
\=> (ks <= yvar)
```

### La preuve manuelle

Étudions alors la preuve manuelle : nous devons montrer l'assertion $\{I1\}rcv\_act\{I1\}$. En utilisant la définition de *wp* et en appliquant les substitutions, nous obtenons la formule suivante :

$$
\begin{array}{ll}
\langle \forall y : y \in acks \vee y \in ackr :: y \leq kr \wedge ks \leq y+1 \rangle & (H1) \\
\wedge \langle \forall x : x \in cs \vee x \in cr :: kr \leq x.dex \leq ks \rangle & (H2) \\
\wedge (kr \leq ks \leq kr+1) & (H3) \\
\wedge (ks = head(acks)) \wedge (acks \neq \emptyset) & (H4) \\
\Rightarrow & \\
\langle \forall y : y \in tail(acks) \vee y \in ackr :: y \leq kr \wedge ks \leq y \rangle & (B1) \\
\wedge \langle \forall x : x \in cs \vee x \in cr :: kr \leq x.dex \leq (ks+1) \rangle & (B2) \\
\wedge (kr \leq ks+1 \wedge ks \leq kr) & (B3)
\end{array}
$$

La preuve consiste alors à utiliser les hypothèses $H1$, $H2$, $H3$ et $H4$ pour montrer les sous-buts $B1$, $B2$ et $B3$ :

Preuve de (B2): $\forall x : (x \in cs \vee x \in cr) \Rightarrow (kr \leq x.dex \leq (ks+1))$

$x \in cs \vee x \in cr$ $(Hyp)$

$\{H2\}$

$kr \leq x.dex \leq ks$

$\{ks \leq ks+1\}$

$kr \leq x.dex \leq ks+1$

□

Preuve de (B3): $(kr = ks+1) \vee (kr = ks))$

$(ks = head(acks)) \wedge (acks \neq \emptyset)$ $(H4)$

$\{H1 \ \ avec \ \ y = head(acks) = ks\}$

$ks \leq kr \wedge ks \leq ks+1$

$\{H3\}$

$ks = kr$

□

Preuve de (B1): $\forall y(y \in tail(acks) \vee y \in ackr) \Rightarrow (y \leq kr \wedge ks \leq y)$

$y \in tail(acks) \vee y \in ackr$ $(Hyp)$

$\{H1\}$

$y \leq kr \wedge ks \leq y+1$

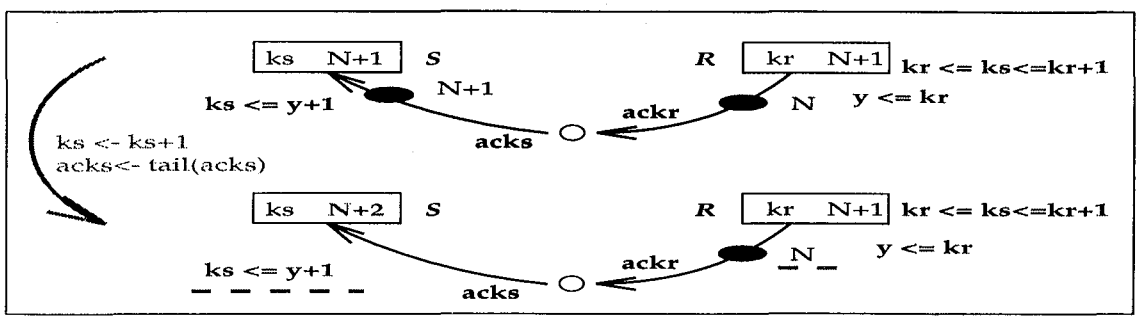



FIG. 6.3 – *Un état non accessible*

$\{kr = ks\}$
$\quad y \leq ks \wedge ks \leq y + 1$
$\{propriété\ des\ entiers\ naturels\}$
$\quad y = ks \vee ks = y + 1$

A ce niveau, nous devons procéder par cas : si $ks = y$, le but (B1) est résolu, sinon le cas $ks = y+1$ ne nous permet pas de conclure car il simplifie (B1) à $faux$. c'est donc un cas impossible. C'est précisément à ce niveau que la preuve mécanisée a échouée. Les sous buts (B2) et (B3) ont été résolus en LP en utilisant les mêmes étapes que celles utilisées dans les preuves manuelles correspondantes. La preuve mécanisée a mis en lumière un problème passé inaperçu dans la preuve manuelle décrite par Chandy et Misra.

### Le problème

Pour illustrer le problème, considérons un état $e_k$ (figure 6.3) dans lequel le numéro du dernier message envoyé est égal à celui du dernier message reçu, $ks = kr = N + 1$ et dans lequel l'acquittement de ce message est présent à la tête du canal d'acquittement $head(acks) = N + 1$. De plus, supposons qu'un ancien acquittement $N$ est encore en transit dans le canal $ackr$. Il est évident que la propriété $I_1$ est satisfaite dans cet état.

Comme $ks = head(acks)$ et $(acks \neq \emptyset)$, l'action $rcv\_ack$ peut être exécutée. L'état résultant $e_{k+1}$ satisfait :

- $ks = N + 2$
- $kr = N + 1$
- $kr \leq ks \leq (kr + 1)$

Mais la propriété $I_1$ n'est pas satisfaite dans l'état $e_{k+1}$, car $\langle \exists y \in ackr :: not(ks \leq (y + 1)) \rangle$, en prenant $y = N$.

Si on se base sur des arguments intuitifs du comportement du protocole, on s'aperçoit que $e_k$ n'est pas *accessible* (cf chapitre 2) à partir de l'état initial. C'est à dire que le programme ne peut atteindre un tel état par une séquence d'actions exécutées à partir de l'état initial. L'existence d'états non accessibles est un problème bien connus pour d'autres raisons dans UNITY (cf. paragraphe 2.8.2).

Les informations dont nous avons besoin concernent l'ordre des éléments dans les séquences. En fait, nous avons besoin d'information pour exprimer le fait que les valeurs des acquittements transitants dans *acks* sont inférieures à celles des acquittements dans *ackr*. Si nous considérons que dans l'état $e_k$ l'élément $N$ est dans *acks*, de nouveau l'état $e_{k+1}$ ne satisfait pas l'invariant 1. Par conséquent, nous avons besoin de l'information que *acks* et *ackr* sont des séquences non décroissantes d'entiers, ce qui est précisément la propriété spécifiée par l'invariant 3. Donc, afin de prouver invariance de $I_1$, on besoin de celle de $I_3$.

### 6.4.2 Invariant 3

Comme le canal de transmission est modélisé par deux paires de canaux non défectueux, les messages transmis ou leurs acquittements ne peuvent se "chevaucher" dans ces canaux[33]. Par conséquent, ces canaux contiennent des messages ou des acquittements dans l'ordre croissant de leur numéro.

Pour prouver l'invariant 3, nous devons montrer que le programme *proto* vérifie la propriété $I_3$ :

**invariant** *cs.dex, cr.dex, acks, ackr are sequences of nondecreasing integers*

Prouver cet invariant en LP revient à prouver le théorème suivant pour chaque séquences *acks*, *ackr*, *cs* et *cr*:

```
Inv(issorted(seq),proto,cond_init) = true
```

Le prédicat `issorted(seq)` est défini[34] comme suit :

```
issorted(emptyseq)              = T
issorted(emptyseq |- exp)       = T
issorted(exp -| emptyseq)       = T
issorted((seq |- exp1) |- exp2) = issorted(seq |- exp1) ^ (exp1 <= exp2)
issorted(exp1 -| (exp2 -| seq)) = issorted(exp2 -| seq) ^ (exp1 <= exp2)
```

• Pour les séquences *acks* et *cr*, seules les actions *rcv_ack*, *receive*, *rmv_old_ack* et *rmv_old_msg* sont pertinentes. Or la suppression de l'élément en tête ne modifie pas l'ordre des éléments de la séquence. Ce lemme est prouvé par induction :

```
set name lemme_seq
prove issorted(s)\=> issorted(tail(s)) = T  by ind depth 2 using seq_ind2
 qed
```

Cette commande indique à LP de prouver la formule `F(s)` avec une induction structurelle de profondeur 2. LP construit et prouve de façon automatique les sous-buts appropriés en utilisant la définition de `issorted`.

Prouver une formule `F(s)` par induction structurelle de profondeur 2 en utilisant la règle d'induction

```
seq_ind2: sort Seq generated freely by emptyseq,-|
```

33. Sous l'hypothèse de correction du programme qui manipulent ces canaux, c'est à dire la propriété de sûreté du canal défectueux, p. 175

34. Les canaux de transmission et les canaux d'acquittement sont représentés par des files à double entrée (cf. paragraphe 3.4) dont les deux opérateurs sont -| et |-.

requière la preuve de deux sous buts de base, `F(emptyseq)` et `F(e -| emptyseq )`, et d'un but correspondant au pas d'induction, `F(e -| (e1 -| sc))`, ayant `F(sc)` et `F(e -| sc)` comme hypothèses d'induction.

En utilisant ce lemme, les invariants suivants sont prouvés de façon automatique avec LP :

```
Inv(issorted(acks),proto,cond_init) = true
Inv(issorted(cr),proto,cond_init)   = true
```

Par exemple, les principales étapes pour le canal *cr* et l'action *receive* sont :

```
inv(issorted(cr),proto,init) = T
→ (init \=> issorted(cr))= T /\ unless(issorted(cr),F,proto) = T
→  unless(issorted(cr),F,receive)= T /\ unless(issorted(cr),F,proto)= T
→  issorted(cr) \=> wp(receive,issorted(cr)) = T
→  issorted(cr) \=> issorted(tail(cr)) = T
```

LP utilise alors le lemme `lemme_seq` pour réduire ce but à `true`.

• Pour les canaux *cs* et *ackr*, LP ne peut achever la preuve de cet invariant, car le lemme précédent est inutile pour les actions *transmit* et *send_ack*, qui elles, ajoutent des éléments à la séquence.

Ayant comme seule hypothèse `issorted(ackr)`, le but `issorted(ackr |- kr)` ne peut être prouvé. En utilisant la définition de `issorted`, ce but est simplifié à `last(ackr) <= kr`.
En effet, les principales étapes sont :

```
unless(issorted(ackr),F,send_ack) = T /\ unless(issorted(ackr),F,proto)= T
→ issorted(ackr) \=> wp(send_ack,issorted(ackr)) = T
→ issorted(ackr) \=> issorted(ackr |- kr) = T
→ issorted(ackr) \=> (issorted(ackr) /\ (last(ackr) <= kr)) = T
→ issorted(ackr) \=> (last(ackr) <= kr) = T
```

Cependant, comme $last(ackr) \in ackr$, en utilisant invariant 1, on peut déduire $last(ackr) \leq kr$. De même, pour montrer que *cs.dex* est un séquence non décroissante, le but est réduit à `ind(last(cs)) <= ks`. Comme $last(cs) \in cs$, en utilisant l'invariant 1 on en déduit $last(cs).dex \leq ks$.

**Conclusion** : L'invariance de $I_1$ requière celle de $I_3$ et réciproquement.

### 6.4.3 Invariant 2

Pour un protocole de communication, il est crucial que les données délivrées par le récepteur soient celles envoyées par l'émetteur. De plus, ces données doivent être délivrées suivant l'ordre de leur émission. Par conséquent, la séquence de donnée à la sortie du protocole doit être un préfixe de la séquence en entrée et la taille de la séquence de donnée en sortie doit être égale au nombre de messages reçus. Ce sont les propriétés spécifiées par le prédicat $I_2$.

Essayons maintenant la preuve de cet invariant : cela consiste à montrer $\{I_2\}s\{I_2\}$ pour toute action $s$ du programme *proto*.

$$\begin{array}{lll} \textbf{Invariant} & \langle \forall x : x \in cs \vee x \in cr :: x.val \in ms \rangle & \wedge \\ & isprefix(mr, ms) & \wedge \\ & len(mr) = kr & \end{array}$$

où $len(mr) \stackrel{def}{=} |mr|$ et $isprefix(q,s) \stackrel{def}{=} q$ *is a prefix of* $s$.

De même que pour l'invariant $I_1$, les seules actions pertinentes pour cet invariant sont *transmit*, *receive* et *rmv_old_ack*. Pour les actions *transmit* et *rmv_old_ack*, la preuve est triviale, nous allons donc nous intéresser à l'action *receive* pour laquelle le but à prouver est le suivant :

$\langle \forall x : x \in cs \vee x \in cr :: x.val \in ms \rangle$ $(H1)$
$\wedge isprefix(mr, ms)$ $(H2)$
$\wedge len(mr) = kr$ $(H3)$
$\wedge (kr \neq head(ackr)) \wedge (cr \neq \emptyset)$ $(H4)$
$\Rightarrow$
$\langle \forall x : x \in cs \vee x \in tail(cr) :: x.val \in ms \rangle$ $(B1)$
$\wedge isprefix(mr; head(cr).val, ms)$ $(B2)$
$\wedge len(mr; head(cr).val) = kr + 1$ $(B3)$

Preuve de B1 : $\langle \forall x : x \in cs \vee x \in cr :: x.val \in ms \rangle$
$x \in cs \vee x \in tail(cr)$ $(Hyp)$
$\{$*propriété des séquences*$\}$
$x \in cr$
$\{H1\}$
$x.val \in ms$
□

Preuve de B3 : $len(mr; head(cr).val) = kr + 1$
$len(mr; head(cr).val) = len(mr) + 1$ $(Hyp)$
$\{H3\}$
$len(mr; head(cr).val) = kr + 1$
□

Preuve de B2 : $isprefix(mr; head(cr).val, ms) \equiv [prefix(ms, kr + 1) = mr; (head(cr).val)]$
$isprefix(mr, ms)$ $(Hyp)$
$\{isprefix(s, q) \equiv [prefix(s, len(q)) = q]\}$
$prefix(ms, len(mr)) = mr$
$\{$*propriété des séquences*$\}$
$prefix(ms, len(mr)); head(cr).val = mr; head(cr).val$
$\{H3\}$
$prefix(ms, kr); head(cr).val = mr; head(cr).val$
$\{H1 \Rightarrow (\exists j (j \leq len(ms)) \wedge (ms.j = head(cr).val) \wedge (head(cr).dex = j))\}$
$prefix(ms, kr); ms[head(cr).dex] = mr; head(cr).val$

A ce niveau, nous ne pouvons conclure car nous ne pouvons pas montrer que $head(cr).dex = kr + 1$. Par contre, en utilisant la propriété $I_1$, on a :

$head(cr) \in cr$
$\{I_1\}$
$kr \leq head(cr).dex \leq ks$
$\{I_1 : kr \leq ks \leq kr + 1\}$
$kr \leq head(cr).dex \leq kr + 1$
$\{H4 : kr \neq head(cr).dex\}$
$head(cr).dex = kr + 1$
□

**Conclusion** : L'invariance de $I_2$ requiert celle de $I_1$.

### 6.4.4 Construction d'un invariant

Nous allons utiliser des arguments intuitifs pour construire un prédicat "plus fort" que les trois propriétés $I_1$, $I_2$ et $I_3$ et qui exclut les états non accessibles. Ce prédicat sera un invariant du programme et il nous permettra de renforcer la propriété à prouver afin qu'elle soit assez forte pour être un invariant. Cette recherche se fera de façon incrémentale jusqu'a obtenir un prédicat assez fort.

Intuitivement, nous devons étudier le comportement du protocole lorsqu'il y a perte de message ou d'acquittement. On rappelle que le canal de transmission des messages et celui des acquittements sont de même nature.

Considérons un état $e_N$, où $N-1$ messages ont été reçus (Figure 6.4) et où les variables d'états satisfont :

- $ks = kr = N$
- $cs = cr = ackr = acks = \emptyset$

Nous sommes dans l'état où le message $N$ vient d'être reçu par le récepteur (action *receive*). Quelles sont les actions susceptibles d'être exécutées dans l'état $e_N$?

Seules les actions *transmit* et *send_ack* peuvent être exécutées. Nous pouvons remarquer que si l'action *transmit* est exécutée, dont l'effet est l'envoi d'une deuxième copie du message $N$, elle sera "compensée" par l'action *rmv_old_msg* qui elle, élimine les copies (dont le numéro est à la valeur $kr$ du dernier message reçu par le récepteur). Nous allons donc considérer que ces deux actions ne sont pas pertinentes pour notre exemple.

L'action qui nous intéresse est *send_ack*. Son exécution résulte en un état $e_{N+1}$, où l'acquittement $N$ est ajouté à *ackr*. Supposons que cet acquittement soit perdu (le passage *ackr* vers *acks* échoue), alors l'action *send_ack* peut être exécutée un nombre fini de fois, disons $i$, car la propriété de sûreté du canal défectueux garantit que cet acquittement ne peut être perdu indéfiniment. Par conséquent, parmi ces $i$ occurrences de l'acquittement $N$, $j$ occurrences ($j \leq i$) apparaissent dans *acks* (état $e_{n_j}$). Nous pouvons alors poursuivre avec l'action *rcv_ack* qui permet à l'émetteur de recevoir l'acquittement du message $N$ (état $e_r$). Il n'est pas nécessaire que l'état $e_r$ soit le successeur immédiat de $e_{n_j}$, il est possible que d'autres copies de message $N$ soient émises dans *cs* avec *transmit*, éliminées avec *rcv_old_msg*, ou bien que d'autres acquittements de valeur $N$ soit émis dans *ackr* avec l'action *send_ack*. Puis l'action *transmit* permet à l'émetteur d'envoyer le message $N+1$, correspondant à l'état $E_s$, enfin le message

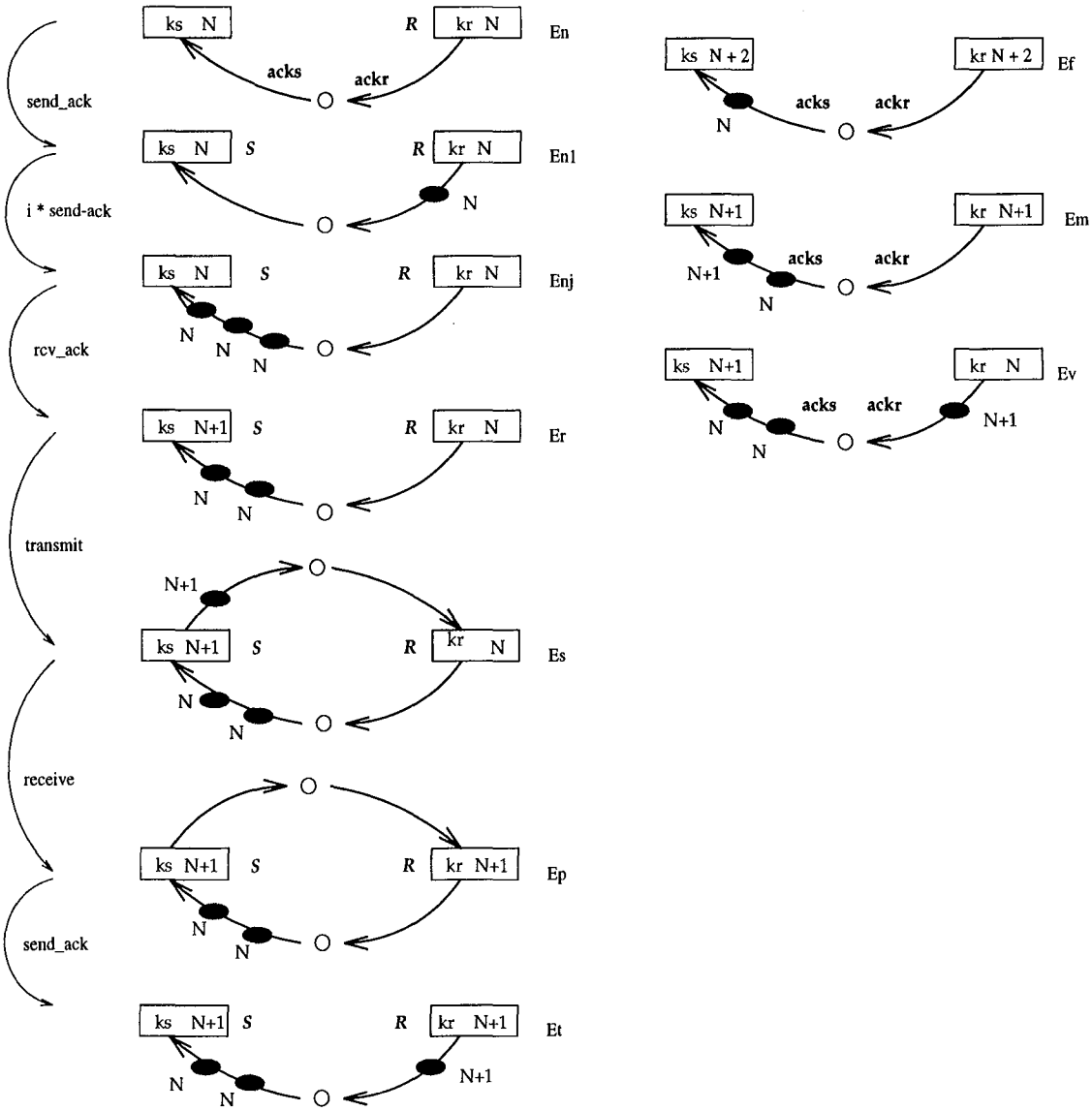



FIG. 6.4 – *Transmission avec perte d'acquittement et états non accessibles*

$N+1$ est reçu par le récepteur avec l'action *rcv_ack* et l'acquittement correspondant est envoyé avec *send_ack*.

Nous pouvons alors renforcer la propriété $I_1$ avec les informations manquantes : tout d'abord, pour exclure l'état $e_k$, on pourrait penser que les acquittements sont toujours égaux au numéro du dernier message reçu par le récepteur ($kr$). Or cette propriété est trop forte, elle exclut des états accessibles, comme l'état $e_p$ de la figure 6.4.

Toujours pour exclure $e_k$, les acquittements dans *acks* doivent être inférieurs à ceux dans *ackr*,

$$(\forall y \in acks) \wedge (\forall z \in ackr) :: y \leq z$$

et le numéro du dernier message reçu doit être égal au dernier message envoyé ou à celui du précédent.

Nous obtenons le prédicat :

$$\begin{array}{ll} \langle kr \leq ks \leq kr+1 \rangle & \wedge \\ \langle \forall y, z : y \in acks \wedge z \in ackr :: y \leq z \rangle & \wedge \end{array} \qquad (P_1)$$

Cette propriété est trop faible, l'état $e_f$ de la figure 6.4 satisfait cette propriété alors qu'il est évident qu'il n'est pas accessible. L'information manquante est que la différence entre le numéro du dernier message envoyé et la valeur des acquittements est de 1. Nous renforçons alors ce prédicat par une propriété supplémentaire :

$$\begin{array}{ll} \langle \forall y : y \in acks \vee y \in ackr :: y \leq ks \wedge ks \leq y+1 \rangle & \wedge \\ \langle kr \leq ks \leq kr+1 \rangle & \wedge \\ \langle \forall y, z : y \in acks \wedge z \in ackr :: y \leq z \rangle & \end{array} \qquad (P_2)$$

De nouveau, l'état $e_v$ de la figure 6.4 satisfait cette propriété alors qu'il n'est pas accessible, car l'acquittement $N+1$ ne peut être dans *ackr* puisque le numéro du dernier message reçu est $N$. C'est à dire que seuls les messages reçus peuvent être acquittés. Par conséquent, la première propriété du prédicat précédent doit être renforcée (en remplaçant l'inégalité $y \leq ks$ par $y \leq kr$):

$$\begin{array}{ll} \langle \forall y : y \in acks \vee y \in ackr :: y \leq kr \wedge ks \leq y+1 \rangle & \wedge \\ \langle kr \leq ks \leq kr+1 \rangle & \wedge \\ \langle \forall y, z : y \in acks \wedge z \in ackr :: y \leq z \rangle & \end{array} \qquad (P_3)$$

De nouveau, l'état $e_m$ de la figure 6.4 satisfait cette propriété alors qu'il est évident qu'il n'est pas accessible. L'information manquante est l'ordre des éléments dans une même séquence, c'est à dire que les séquences doivent être non décroissantes. En complétant par les propriétés symétriques pour le canal de transmission, on obtient le prédicat :

$$\begin{array}{ll} \langle \forall y : y \in acks \vee y \in ackr :: y \leq kr \wedge ks \leq y+1 \rangle & \wedge \\ \langle \forall x : x \in cs \vee x \in cr :: x.val \in ms \rangle & \wedge \\ \langle kr \leq ks \leq kr+1 \rangle & \wedge \\ \langle \forall y, z : y \in acks \wedge z \in ackr :: y \leq z \rangle & \wedge \\ \langle \forall x, t : x \in cs \wedge t \in cr :: t.dex \leq x.dex \rangle & \wedge \\ \langle cs.dex, cr.dex, acks, ackr \textit{ are sequences of} & \\ \quad \textit{nondecreasing integers} \rangle & \end{array} \qquad (I)$$

Enfin, nous montrons que le programme *proto* satisfait les propriétés :

*Invariant* $I$
$I \Rightarrow I_1$
$I \Rightarrow I_2$
$I \Rightarrow I_3$

Conclusion : $I_1$, $I_2$ et $I_3$ sont des prédicats *toujours-vrai*, ou bien des propriétés *restreintes* (sous entendu aux états accessibles [Pet91, Pet92]), ou bien des propriétés *indicées* (sous entendu par $I$, [San91]).

En effet, dans la spécification du premier invariant, le prédicat $I1$ est *toujours-vrai* , c'est à dire que les transitions d'un état dans lequel $I1$ est valide à un état dans lequel $I1$ n'est pas valide sont possibles, alors qu'elles ne le sont pas pour un *invariant*. Un prédicat est *toujours-vrai*

pour un programme si et seulement si il est valide dans tout état initial et il est préservé par toute action si elle est exécutée dans un état accessible.
Pour montrer les propriétés restreintes aux états accessibles, il faut construire le "plus fort" invariant du programme, $A$, qui identifie de façon unique les états accessibles du programme. Formellement,

**Définition 6.1 (invariant)** *Un prédicat p est un invariant d'un programme Prg si et seulement si il est valide dans tous état initial et p est préservé par toutes les actions du programme, c'est à dire,* $\forall s, \in Prg\{p\}s\{p\}$.

**Définition 6.2 (Always-true)** *Un prédicat p est always-true dans un programme Prg si et seulement si il est valide dans tous état initial et p est préservé par toute action du programme, quand elle est exécutée dans un état accessible, c'est à dire un état où A est valide, c'est à dire,* $\forall s \in Prg\{p \wedge A\}s\{p\}$.

**Théorème 6.1 ( [Pet92])** *p est always-true est équivalent à* $p \wedge A$ *est un invariant, où A caractérise de façon unique l'ensemble des états accessibles.*

Dans notre exemple de la figure 6.3, $e_k$ n'est pas un état accessible. Un état est *accessible* s'il est atteint à partir de l'état initial par une séquence d'actions du programme, alors que les états *possibles* du programme sont définis par le domaine des variables apparaissant dans le texte du programme. En effet, l'espace d'état du programme est défini comme le produit cartésien des domaines des variables.
Néanmoins, le plus "fort" invariant est difficile à calculer. Beverly Sanders a montré dans [San91] qu'un prédicat plus "faible" suffisait pour montrer les propriétés restreintes aux états accessibles. Formellement, ce résultat est basé sur le théorème suivant :

**Théorème 6.2 ( [Pet92])**

$$\frac{Invariant \;\; I, I \Rightarrow p}{invariant_R \;\; p}$$

*où* $invariant_R \;\; p \stackrel{def}{=} always\text{-}true \;\; p$

Pour illustrer la différence entre un invariant et un *always-true*, nous reprenons l'exemple utilisé dans [Pet91].

**Exemple 6.3** *Soit le programme*
**Program** *{Expl}*

**Declare** 0 : *integer*
**initially** $k = 0$
**assign**
$\{kact\} \;\; k := 2 \quad if \; (k = 1)$
**end***{Expl}*

*Le prédicat* $k = 0$ *caractérise de façon unique les états accessible de Expl. En utilisant la définition, on peut déduire que* $\neg(k = 1)$ *est un invariant de Expl. D'après le théorème 6.2,* $\neg(k = 1)$ *est always − true dans Expl. Comme* $\forall s \in expl$, $\{k < 2 \wedge k = 0\}s\{k < 2\}$, *d'après la définition 6.2,* $k < 2$ *est always-true. Mais* $k < 2$ *n'est pas un invariant puisque l'assertion* $\{k < 2\}s\{k < 2\}$ *n'est pas valide.*

### 6.4.5 Remarques sur la mécanisation

Ce problème était difficile à détecter dans la preuve manuelle correspondante, en raison du nombre important de détails engendrés par les obligations de preuve (une pour chaque action du programme) ainsi que la taille et la complexité des prédicats manipulés. Ainsi, la dernière étape de la preuve du sous-but (B1) de l'invariant $I_1$ est passée inaperçue. Ce phénomène est courant pour des preuves complexes où on utilise plusieurs hypothèses et suppositions basées sur des "réflexes" qu'on ne vérifie pas et que le démonstrateur ne peut pas accepter. D'où l'intérêt de vérifier mécaniquement les preuves manuelles que nous pensons être correctes. Néanmoins, la preuve manuelle nous a permis de structurer la preuve mécanisée, de choisir les lemmes nécessaires et les faits pertinents à appliquer. De plus, il nous semble difficile d'effectuer des preuves mécanisées sans avoir leur structure globale.

Les preuves des propriétés de sûreté sont semi-automatisées. En effet, pour prouver ces propriétés, on utilise un nombre fixe de commandes LP. Ce dernier prend en charge les étapes fastidieuses de la preuve manuelle, à savoir quelle hypothèse utiliser et à quel moment, comment simplifier les expressions de prédicats[35], comment simplifier les expressions arithmétiques, etc. Ceci nous permet de nous concentrer sur les aspects de la preuve relevant du problème considéré. Ainsi, les imperfections et incertitudes de cette spécification ont pu être détectées grâce aux facilités de mise au point des preuves offertes par LP, qui permet d'utiliser des techniques standards pour effectuer les preuves mais surtout pour comprendre pourquoi elles échouent à l'aide des traces et du retour arrière. En particulier, pour la preuve du but $B1$, "le problème" était que $(ks \leq (y+1))$ n'implique pas que $(ks \leq y)$. Bien que cette propriété semble évidente, elle peut passer inaperçue dans une preuve manuelle où le nombre d'obligations de preuves et de détails est considérable. En effet, même si LP "ne propose pas de solution", il fournit un ensemble d'informations sur l'état de la preuve, les hypothèses actuelles, le but recherché et la raison technique de l'échec (par exemple, l'inconsistance générée en utilisant l'hypothèse $n$ et l'hypothèse $k$).

## 6.5 Obligations de preuve: Vivacité

Dans la section précédente, nous avons montré à l'aide de LP, que le protocole de communication satisfait sa propriété de sûreté, à savoir que les données délivrées par le récepteur doivent avoir été transmises par l'émetteur, spécifiée par une propriété sur les séquences de données à l'entrée et à la sortie du protocole: la séquence en sortie doit être un préfixe de la séquence en entrée. Cependant, un protocole qui ne délivre aucune donnée vérifie cette propriété, puisque toute séquence de donnée *vide* est un préfixe de la séquence d'entrée. Par conséquent, nous avons besoin d'une propriété de vivacité: les données émises doivent être reçues tôt ou tard par le récepteur, c'est à dire que la taille de la séquence de donnée en sortie augmente.

$$|mr| = n \quad \textit{leads\_to} \quad |mr| = n+1 \qquad (P)$$

Les preuves de vivacité sont de nature différente de celles des propriétés de sûreté. Pour prouver une propriété *unless* ou *ensures*, nous appliquons la définition, qui nous permet de montrer que la propriété est satisfaite pour toutes les actions du programme (cf. Chapitre 2, paragraphe 2.7). Ceci n'est pas le cas pour la preuve d'une propriété **leads_to**, qui elle, est définie par trois

35. Pour les actions qui ne modifient pas les variables apparaissant dans les propriétés, la formule à prouver se ramène à des tautologies de la forme $(P \wedge \neg Q) \Rightarrow (P \vee Q)$.

règles d'inférence. Ceci est dû au fait que les deux premières ont une définition opérationnelle qui permet la vérification pour toutes les transitions (de l'une à l'autre).

Pour montrer une propriété de la forme $p$ *leads_to* $q$, la stratégie consiste à "étudier" si c'est une propriété "statique", c'est à dire qu'elle découle directement de $p$ *ensures* $q$. Sinon, nous devons trouver des "prédicats" intermédiaires pour pouvoir utiliser les règles d'inférence définissant *leads_ to*, telle que la transitivité par exemple. Pour contourner cette difficulté, nous procédons par décomposition pour simplifier la conjecture, afin de montrer des propriétés plus "simples". Nous exhibons des propriétés intermédiaires à montrer et des règles à utiliser pour les prouver.

L'une des règles les plus utilisées pour prouver une propriété *leads_ to* est la règle d'*élimination*:

$$\textbf{CAN\_rule}: \quad \frac{leads\_to(p, q \vee b), leads\_to(b, r)}{leads\_to(p, q \vee r)}$$

Dans ce qui suit, nous décrivons la preuve de la propriété de progrès que le programme *proto* doit satisfaire. La preuve proposée par Chandy et Misra est la suivante[36], en utilisant d'après l'invariant 2, $kr$ pour $|mr|$. ( Nous utiliserons $\mapsto$ pour *leads_ to*) :

$$\begin{array}{lll}
kr = n & \mapsto (kr = n+1) \vee (n \in acks) & (P_1) \\
n \in acks & \mapsto ks = n+1 & (P_2) \\
kr = n & \mapsto (kr = n+1) \vee (ks = n+1) & (P_3) \\
 & \text{CAN_rule avec } (P_1) \text{ et } (P_2) & \\
ks = n+1 & \mapsto kr = n+1 & (P_4) \\
kr = n & \mapsto kr = n+1 & (P) \\
 & \text{CAN_rule avec } (P_3) \text{ et } (P_4) &
\end{array}$$

Nous devons compléter la preuve proposée par les auteurs, en particulier, comme nous allons le voir, les preuves de $(P_2)$ et $(P_4)$.

**Preuve de** $(P_1)$ : Cette propriété est prouvée en utilisant la *propriété conditionnelle* (COND_prop) d'un canal défectueux (cf. p.175). En particulier, on considère le canal défectueux des acquittements (transmission du récepteur vers l'émetteur), en posant $p.m \equiv (kr = m)$.

**Preuve de** $(P_2)$ : Nous devons montrer que lorsqu'un élément $n$ (l'acquittement du énième message) est en transit dans le canal des acquittements *acks*, alors tôt ou tard $ks$ sera égal à $n+1$. Nous exhibons d'abord le fait que lorsqu'un élément $n$ est dans *acks*, les actions *rcv_ack* et *rcv_old_ack* de l'émetteur garantissent que cet élément arrivera tôt ou tard à la tête de *acks* ou bien que $ks$ sera égal à $n+1$ ( l'émetteur incrémente $ks$ lorsqu'il reçoit l'acquittement du énième message).
Cette progression est exactement la sémantique du principe d'induction (cf. p.60, paragraphe 2.7.0.0.0), en utilisant comme métrique la position de l'élément dans la séquence :
Pour tout état du programme dans lequel le prédicat $(n \in acks)$ est valide, l'exécution du programme finit par atteindre un état dans lequel le prédicat $((n \in acks \wedge head(acks) = n) \vee ks = n+1)$ est valide, ou bien il atteint un état dans lequel $(n \in acks)$ est valide et la position de $n$ par rapport au début de la séquence *acks* a diminuée. Or, la position d'un élément ne peut

36. Les règles d'inférence utilisées dans ce qui suit sont énoncées au le chapitre 4, paragraphe 4.4.

décroître à l'infini $pos(head(seq)) = 1, pos(head(tail(seq))) = 2, \ldots ect$, donc il atteindra fatalement $head(acks)$.
De plus, lorsqu'un élément $n$ est à la tête de *acks*, on sait que *ks* sera égale à $n+1$ (d'après le texte du programme *proto*), nous pouvons utiliser *ensures* pour le montrer. Par conséquent, la preuve de $n \in acks \ \mapsto \ ks = n+1$ est décomposée suivant ces trois étapes :

| | | |
|---|---|---|
| $n \in acks$ | $\mapsto ks = n+1 \vee (n \in acks \wedge head(acks) = n)$ | IND |
| $n \in acks \wedge head(acks) = n$ | $\mapsto ks = n+1$ | par *ensures* |
| $n \in acks$ | $\mapsto ks = n+1$ | CAN_rule |

**Preuve de** $(P_4)$: Nous avons à prouver la propriété suivante: $ks = n+1 \ \mapsto \ kr = n+1$, qui exprime la propriété que lorsque l'émetteur envoie le $(n+1)ieme$ message, le récepteur finit par le recevoir. Le protocole est conçu de telle façon que si le message est perdu la première fois qu'il est envoyé, l'émetteur le ré-émet jusqu'a ce qu'il reçoive l'acquittement correspondant. La propriété de sûreté assure qu'un message transitant sur le canal peut être perdu mais seulement un nombre fini de fois.

Pour prouver $(P_4)$, nous devons décomposer le problème, en utilisant la propriété conditionnelle d'un canal défectueux (COND_prop), qui est la seule façon de montrer le passage des éléments entre les canaux de transmission *cs* et *cr*.

**Preuve:**

| | | | |
|---|---|---|---|
| $ks = n+1$ | $\mapsto p.m \vee kr = n+1$ | par *ensures* | (1) |
| $p.m$ | $\mapsto \neg(p.m) \vee (m \in cr)$ | COND_prop | (2) |
| $p.m$ | *unless* $(m \in cr) \vee head(cr) = m \vee kr = n+1$ | | (3) |
| $p.m$ | $\mapsto (p.m \wedge m \in cr) \vee head(cr) = m \vee m \in cr \vee kr = n+1$ | | (4) |
| | | PSP_rule(2,3) | |
| $ks = n+1$ | $\mapsto (p.m \wedge m \in cr) \vee head(cr) = m \vee m \in cr \vee kr = n+1$ | | (5) |
| | | CAN_rule(1,4) | |
| $p.m \wedge (m \in cr) \wedge pos(m, cr) = i$ | $\mapsto (p.m \wedge (m \in cr) \wedge pos(m, cr) < i)$ $\vee (head(cr) = m \wedge m \in cr) \vee kr = n+1$ | par *ensures* | (6) |
| $p.m \wedge (m \in cr)$ | $\mapsto (head(cr) = m \wedge m \in cr) \vee kr = n+1$ | IND(6) | (7) |
| $ks = n+1$ | $\mapsto head(cr) = m \vee m \in cr \vee kr = n+1$ $\vee (m \in cr \wedge head(cr) = m)$ | CAN_rule(7,5) | (8) |
| $head(cr) = m$ | $\mapsto m \in cr \wedge head(cr) = m$ | IMPL_rule | (9) |
| $ks = n+1$ | $\mapsto kr = n+1 \vee m \in cr \vee (m \in cr \wedge head(cr) = m)$ | CAN_rule(9,8) | (10) |
| $m \in cr$ | $\mapsto (m \in cr \wedge head(cr) = m)$ | IND | (11) |
| $ks = n+1$ | $\mapsto kr = n+1 \vee (m \in cr \wedge head(cr) = m)$ | | (12) |
| | | CAN_rule(10,11) | |
| $head(cr) = m \wedge m \in cr$ | $\mapsto kr = n+1$ | par *ensures* | (13) |
| $ks = n+1$ | $\mapsto kr = n+1$ | CAN_rule(12,13) | |

**fin.**

**(1)** En posant $m$ comme étant $(n+1, ms[n+1])$ et $p.m$ comme étant $(ks = n+1 \wedge ((n+1, ms[n+1]) \in cs))$, nous commençons par montrer que partant d'un état où $ks = n+1$, on atteint un état où le message $m$ est en transit dans *cs* ou bien un état où $kr = n+1$, c'est à dire qu'il a été reçu.

**(2)-(5)** Poursuivant la décomposition, nous nous plaçons dans le premier cas, c'est à dire la situation où $p.m$ est valide (le message $m$ est en transit dans *cs*). On montre alors que

cette situation mène à une situation où le message apparaît dans le canal de transmission $cr$.

**(6)-(12)** Une fois le message dans le canal de transmission $cr$, nous utilisons le principe d'induction pour monter qu'il progresse dans le canal jusqu'a ce qu'il atteigne $head(cr)$.

**(13)** Une fois que l'élément ait atteint la tête du canal, l'action *rcv_msg* et l'hypothèse d'équité garantissent la réception du message et la mise à jour du compteur $kr$ par la valeur de l'index de $m$.

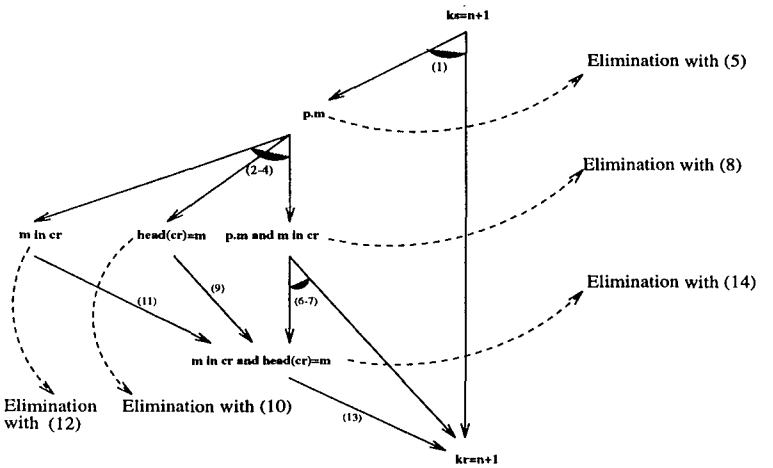



FIG. 6.5 – *Diagramme des applications de la règle d'élimination*

Nous avons construit cette preuve en utilisant un "arbre de recherche" similaire à ceux décrits dans [Lam82], appelés *proof lattices*, où la racine est le prédicat valide dans l'état de départ (Figure 6.5).

**Définition 6.3 ( [Lam82])** *Un arbre de preuve pour un programme $P$ est un graphe orienté acyclique dans lequel chaque noeud est étiqueté par une assertion et tel que*

1. *Il existe un noeud unique qui n'admet pas de noeud père,*
2. *Il existe un noeud unique qui n'a pas de descendants,*
3. *Si un noeud $N$ admet $k$ fils, $N_1, \ldots N_k$ alors la propriété $N \mapsto N_1 \vee \ldots \vee N_k$ est valide pour le programme.*

**Théorème 6.3 ( [Lam82])** *S'il existe un arbre de preuve pour un programme $Prg$ avec un noeud entrée $P$ et un noeud de sortie $Q$, alors $P \mapsto Q$ est valide pour le programme $prg$.*

Partant d'un noeud, nous étudions (déplions) les différentes possibilités en utilisant des arguments intuitifs du comportement du protocole, afin d'obtenir des propriétés plus simples puis réduire l'arbre à une seule feuille, qui représente le prédicat que l'on cherche à prouver.
L'importance de la règle CAN_rule réside dans le fait que lorsque on veut prouver $p \mapsto q$, nous partons de $p$ et nous essayons de "comprendre" ce que l'on veut obtenir : soit un état dans lequel $q$ est valide, soit un état où un autre prédicat $r$ est valide. Par conséquent, il est crucial de trouver $r$, il représente un prédicat valide dans un état intermédiaire *précédant* l'état où $q$ est valide, et la règle CAN_rule permet de l'éliminer. Chaque triangle de la figure 6.5 représente l'application de la règle de CAN_rule, et la fermeture du triangle veut dire que son application réussit.

### La preuve mécanisée

Nous n'allons pas détailler la mécanisation de ces preuves de vivacité, puisque d'une part, la complexité de la décomposition produit un nombre considérable d'obligations de preuves et d'autre part, comme nous l'avons vu, elles sont décomposées en propriétés *unless* et *ensures*. La mécanisation des preuves de ces propriétés est illustré par les preuves d'invariance du paragraphe précédent.

Il est évident que la décomposition n'est pas automatisable, puisqu'elle est basée essentiellement sur une compréhension intuitive du comportement du protocole. Néanmoins, la preuve mécanisée nous oblige, d'une part à structurer les différentes étapes, et d'autre part à vérifier toutes les étapes, aussi infimes soient elles. En effet, la vérification des preuves manuelles de ces propriétés a révélé que nombreuses preuves étaient erronées. La plupart des erreurs étaient dues aux cas de base, c'est à dire quand les variables ont des valeurs "limites". En particulier, l'erreur la plus fréquente résidait dans la simplification de la propriété à prouver sur les séquences quand celles ci sont vides (plusieurs actions du programme ont cette condition). Certes, ce type d'erreur ne semble pas grave à première vue mais si nous considérons des protocoles plus réels, ces erreurs peuvent compromettre le fonctionnement du système global. Bien sûr, ces erreurs sont d'autant plus graves lorsque le protocole en question met en jeu des vies humaines. D'où l'écart considérable entre la nature de l'erreur et son impact sur le système.

## 6.6 Discussion et Conclusion

Nous avons décrit dans ce chapitre la spécification et la vérification d'un protocole de communication à travers des canaux défectueux en utilisant la théorie UNITY et son système de preuve ainsi que sa mécanisation en utilisant le démonstrateur LP. Le protocole étudié, proposé par Chandy et Misra [Cha88], est conçu pour une communication entre émetteur et récepteur à travers un canal défectueux, à savoir que les données peuvent être perdues ou dupliquées. Les propriétés de sûreté du canal garantissent qu'aucun message n'est perdu indéfiniment, ni indéfiniment dupliqué. La propriété de progrès garantit que si un message est re-transmis plusieurs fois, il sera reçu tôt ou tard. Le canal défectueux et le protocole de communication sont spécifiés comme des programmes UNITY dont le composition spécifie une communication correcte de données entre émetteur et récepteur à travers des canaux défectueux. D'autres expériences pour la vérification mécanisée de protocole de communication ont utilisé UNITY comme environnement de spécification, notamment la vérification d'un protocole à *fenêtre coulissante* (sliding windows) [Pet93] avec HOL. Le protocole vérifié est une généralisation de celui considéré dans cette étude, à savoir qu'une séquence de messages est émis simultanément. La transmission utilise plusieurs canaux de communication défectueux qui peuvent perdre, et/ou dupliquer et/ou réordonner les messages, car la propriété de sûreté n'exige pas que les messages soient reçus suivant l'ordre d'émission.

Nous avons montré que le protocole vérifie ses propriétés de sûreté et de progrès, sous l'hypothèse que la spécification du canal est correcte. Ces preuves étaient basées sur des schémas de preuves proposés par les auteurs, qu'il a fallu étendre et compléter. En effet, notre but initial était de vérifier ces preuves à l'aide du démonstrateur LP. Cette vérification mécanisée comporte deux étapes, d'une part développer toutes les étapes de la preuve et d'autre part les vérifier à l'aide du démonstrateur.

Nous avons montré que LP permet de construire une preuve dont la structure est similaire à celle d'une preuve manuelle minutieuse. Néanmoins, les étapes de base des preuves manuelles sont généralement résolues en utilisant des arguments intuitifs. Comme ces arguments sont sou-

vent basés sur des aspects critiques du problème, ils doivent être formalisés et mécanisés afin de rechercher les erreurs et les fautes.

Le raisonnement formel sur la correction des protocoles est nécessaire pour garantir que les protocoles fonctionnent par rapport à leurs spécifications, puisque le raisonnement informel n'est pas fiable et le test n'est pas complet. Néanmoins, ce raisonnement formel est une tâche complexe et sujette à des erreurs. Cette complexité est due au fait que les protocoles eux mêmes sont complexes et exhibent des comportements compliqués. L'utilisation d'un outil de preuve pour vérifier la correction des preuve est donc nécessaire pour le succès d'une telle tâche.

Bien que l'exemple étudié ici soit classique, il illustre bien le type de preuves qui sont généralement effectuées pour les systèmes parallèles. Dans ce type de preuve, nous sommes souvent confrontés à un choix entre des arguments intuitifs et des preuves minutieuses. L'utilisation d'un démonstrateur nous oblige à vérifier toutes les suppositions basées sur des arguments intuitifs, et fournit donc un degré élevé de certification de la vérification. Par ailleurs, la semi-automatisation des preuves nous permet d'éviter des détails fastidieux des preuves manuelles afin de se concentrer sur les aspects profonds du problème. Cette expérience nous a montré que même un protocole "aussi simple" comprend une complexité inhérente à la plupart des protocoles de communication, due au nombre d'obligations de preuves et aux propriétés qui en découlent. En effet, nous avons vu que la complexité et le nombre de propriétés à vérifier font que les preuves manuelles, même détaillées, ne sont pas suffisantes pour s'assurer de la correction. Les preuves manuelles comprennent des hypothèses et des suppositions non vérifiées basées sur une compréhension hâtive du problème ou bien à des simplifications hasardeuses. Par contre, l'utilisation d'un outil d'aide à la preuve nécessite la formalisation et la vérification de ces suppositions. Par conséquent, il est facile d'imaginer le nombre d'erreurs qui peuvent s'introduire dans la vérification de protocoles plus complexes, où la quantité d'information à manipuler augmente l'utilisation d'hypothèses arbitraires posées pour un gain de temps. L'impact de telles erreurs est alors difficiles à quantifier, et inacceptables pour des protocoles mettant en jeu des vies humaines.

L'avantage de cette approche réside dans les optimisations ultérieures du protocole. Ces optimisations nécessitent peu de modification de la formalisation et de la stratégie de preuve, relativement à d'autres approches [Bor93]. Comme le protocole est spécifié par un programme et son comportement comme une exécution de programme, une optimisation telle que le protocole du *Bit alterné* requière la modification du programme (uniquement les conditions des actions) et non de la stratégie de preuve. Ceci est dû en partie au fait que la plupart de nos preuves de sûreté sont calculatoires dans le sens où elles consistent à effectuer un certain nombre de transformation syntaxique et des étapes de raisonnement sémantique.

Il serait intéressant de généraliser la stratégie utilisée pour prouver les propriétés de vivacité, en particulier étudier dans quelle mesure elle peut être appliquée à des systèmes plus importants, et surtout comment rejouer les preuves en réponse au changement dans la spécification initiale pour traiter des problèmes de taille réelle.

# Chapitre 7

# Le problème des Lecteurs-Rédacteurs

## Introduction

Ce chapitre décrit une application de notre système à la preuve d'un programme UNITY, à savoir le problème classique des lecteurs rédacteurs. Cet exemple nous permet d'illustrer la mécanisation des preuves de sûreté et en particulier la réutilisation des preuves. Nous allons décrire les étapes principales de la mécanisation de la preuve du problème des lecteurs rédacteurs tel qu'il est proposé par Chandy et Misra dans [Cha88].

Un nombre $N$ de processus se partagent un fichier pour une lecture ou une écriture. A un instant $t$, $nr$ représente le nombre de processus en lecture et $nw$ le nombre de processus en écriture. La solution proposée garantit le progrès pour les écrivains, c'est à dire que les écrivains sont prioritaires. Les variables d'état doivent satisfaire les invariants suivants:

$$\textbf{invariant} \quad 0 \leq nr \leq N \wedge 0 \leq nw \leq N \tag{7.1}$$

$$\textbf{invariant} \quad nw \leq 1 \wedge (nr = 0 \vee nw = 0) \tag{7.2}$$

L'invariant 7.1 spécifie qu'un certain nombre de lectures peuvent avoir lieu de façon concurrente mais qu'une écriture ne peut se faire avec une autre écriture ou une lecture. Nous décrivons dans ce qui suit, la solution proposée par Chandy et Misra.

Ayant 7.1, on peut écrire 7.2 comme $((nr + N * nw) \leq N)$. Soit $t$ une variable qui satisfait $t = N - (nr + N * nw)$, l'invariant 7.2 peut être écrit:

$$\textbf{invariant} \quad 0 \leq t \tag{7.3}$$

Nous commençons par décrire la spécification du problème, le programme "solution" et la preuve proposée par Chandy et Misra. Nous décrirons par la suite la représentation du programme en LP et la mécanisation de sa preuve de correction. Nous montrerons que la mécanisation de la preuve nécessite de modifier le programme-spécification de Chandy et Misra, notamment les pré-conditions de certaines actions. En particulier, nous allons montrer comment une preuve, même simple, peut être détaillée et subdivisée en de nombreuses étapes afin de la vérifier formellement. De plus, nous verrons comment appliquer le principe d'induction associé à l'opérateur *leads_to*.

## 7.1 Le programme

Le programme UNITY décrivant le comportement des processus lecteurs et rédacteurs est le suivant:

**Program** {User}
**Declare** $t : integer, b : boolean$
**Always** $t = N - (nr + N * nw)$
**initially** $b = false \,[]\, nr, nw = 0, 0$
**assign**

| | | | |
|---|---|---|---|
| $\{startread\}$ | $nr$ | $:= nr + 1$ | $if\ 1 \leq t \wedge \neg b$ |
| $[]\{endread\}$ | $nr$ | $:= nr - 1$ | $if\ nr \neq 0$ |
| $[]\{startwrite\}$ | $nw, nq, b$ | $:= nw + 1, nq - 1, false$ | $if\ N \leq t \wedge 0 < nq$ |
| $[]\{endwrite\}$ | $nw$ | $:= nw - 1$ | $if\ nw \neq 0$ |
| $[]\{set\_b\}$ | $b$ | $:= (nq > 0)$ | |

**end**{User}

La priorité aux écrivains est assurée à l'aide de la variable booléenne $b$ et d'un compteur $nq$ de processus écrivains en attente. La stratégie est la suivante: $b$ est vraie si il y a un processus écrivain en attente, si un processus est en attente d'écriture, une opération d'écriture commencera ou bien $b$ sera vraie, $b$ reste vrai jusqu'à ce qu'une opération d'écriture commence et une opération de lecture ne peut se faire tant que $b$ est vraie.

Nous avons légèrement modifié le programme proposé dans [Cha88], en introduisant une condition initiale sur les variables $nr$ et $nw$. Nous avons ajouté des conditions sur les actions *endread* et *endwrite* et enfin la condition $N \leq t$ est remplacée par la version originale de l'invariant (7.2), $nr = 0 \wedge (N = 0 \vee nw = 0)$. Nous expliquons dans la suite le pourquoi de ces modifications.

## 7.2 Correction

Les invariants (7.1) et (7.2) sont considérés comme des hypothèses de base de la spécification. Ce sont des propriétés qui posent des contraintes sur les valeurs minimales et maximales des variables d'états.

La propriété suivante est supposée valide: Les lectures se font en un temps fini. En d'autres termes, si à un instant $t$, le nombre de lecteurs est égal à un certain entier $k$ strictement positif, alors le nombre de lecteurs sera nécessairement différent de $k$ dans le futur.

$$nr = k \wedge 0 < k \ \textbf{leads\_to}\ nr \neq k \tag{7.4}$$

Pour garantir la priorité aux écrivains , le programme *User* doit vérifier, outre les trois invariants précédent, la propriété de progrès suivante:

$$0 < nq \ \textbf{leads\_to}\ nw = 1 \tag{7.5}$$

où $nq$ représente le nombre de processus écrivains en attente. Cette propriété peut donc être considérée comme *la spécification* que le programme *User* doit satisfaire.

La preuve de cette propriété consiste à prouver des propriétés plus simples puis à utiliser les règles d'inférence de la relation *leads_to* afin de dériver la propriété voulue. La preuve correspondante est la suivante ($\mapsto$ représente l'opérateur *leads_to*).

$b \Rightarrow nq > 0$ (6)

$nq > 0$ **ensures** $nw = 1 \vee b$ (7)

$b \wedge nr = 0$ **ensures** $nw = 1$ (8)

$b \wedge nr = k \wedge k > 0$ **unless** $b \wedge nr < k$ (9)

$b \wedge nr = k \wedge k > 0 \mapsto b \wedge nr < k$ (10)

PSP_rule avec (4,9)

$b \mapsto b \wedge nr = 0$ (11)

IND avec (10)

$b \mapsto nw = 1$ (12)

TRANS_rule avec (10,8)

$nq > 0 \mapsto nw = 1$ (13)

CAN_rule avec (12) et (7)

## 7.3 Représentation en LP

Dans ce qui suit, nous expliquons la formalisation du programme *User* en LP, en définissant:

- Une constante de type `Actlist` pour représenter le programme *User*:

```
declare operator
User : -> Actlist
```

Nous *affectons* à cette constante la liste des actions du programme.

```
User = cons(startread,..., cons(set_b,nil))
```

- Quatre constantes de type `Id`, pour représenter les variables d'états du programme: `nw`, `nr`, `nq` sont des identificateurs d'entiers naturels et `bo` est un identificateur d'une expression booléenne. Par contre, `k` est une variable qui apparaît dans le texte de la preuve, c'est donc une variable auxiliaire de preuve. Elle est alors déclarée comme une variable de type `id_of_var`. De même, `N` est une constante du problème déclarée comme une constante de type `id_of_var`.

- Une constante de type `Bexp` pour représenter la condition initiale:

```
assert
init = (id_to_exp(nr) \= nat_to_exp(0))
       /\ (id_to_exp(nw) \= nat_to_exp(0))
       /\ (bexp_to_exp(id_to_bexp(bo))\= bexp_to_exp(F))
```

**Remarque 7.1** `User` *et* `init` *ne sont que des abréviations. Nous pouvons utiliser directement l'expression booléenne chaque fois que nous avons besoin de la condition initiale. Pour des raisons d'homogénéité et de clarté, l'utilisation de ces constantes s'avère nécessaire surtout en cas de réutilisation de la preuve ou d'optimisations ultérieurs du programme.*

Pour les substitutions textuelles, les assertions suivantes sont nécessaires. Sémantiquement, cela nous assure que le nom des identificateurs est unique[37].

```
set name read_write
assert
nr = firstId
nw = nextid(firstId)
bo = nextid(nextid(firstId))
nq = nextid(nextid(nextid(firstId)))
```

La section **always** est formalisée par une équation:

```
assert
tn = ((id_to_exp(nn)-(id_to_exp(nr)+(id_to_exp(nn) * id_to_exp(nw)))))
```

Nous associons des constantes de la sorte `Act` à chaque action du programme:

```
declare operators
startread,endread,startwrite,endwrite,set_b : -> Act
```

puis à chaque constante, nous affectons l'action correspondante. Par exemple, les deux actions, *endwrite* et *set_ b*,

$\{endwrite\}\ nw := nw - 1 \qquad if\ nw \neq 0$

$\{set\_b\}\ b := nq > 0$

sont formalisées à l'aide des opérateurs `cond_mult_assg((id.exp), bexp)` et `assg(id.exp)` qui représentent l'affectation conditionnelle multiple et l'affectation simple:

```
cons(cond_assg((nw.(id_to_exp(nw)-nat_to_exp(s(0)))),
                nnot(id_to_exp(nw) \= nat_to_exp(0))),
 cons(assg(bo.(id_to_exp(nq) > nat_to_exp(0))),nil));
```

## 7.4 Obligations de preuves

### 7.4.1 Propriété de sûreté

Les obligations de preuve décrivent les propriétés que le programme *User* doit vérifier. Plus précisément, ces obligations sont des théorèmes équationnels qui représentent les invariants et les propriétés (6-9). En particulier, nous allons détailler dans ce qui suit les preuves mécanisées et nous illustrerons l'utilisation de l'*immunité* (cf. Chapitre 1, page 19).

#### Preuve de la propriété (6)

Les variables $b$ et $nq$ ont été introduites pour garantir la priorité aux écrivains. La stratégie consiste à utiliser un compteur $nq$, qui représente le nombre d'écrivains en attente, et empêcher la lecture si $nq > 0$. De plus, afin d'éviter d'ajouter cette condition à toute instruction de lecture (il y a autant d'instructions de lecture que de processus lecteurs), on utilise une variable $b$ qui prend la valeur *vraie* dès qu'un écrivain se met en attente. La propriété suivante doit donc être stable pendant toute l'exécution du programme.

$$b \Rightarrow nq > 0 \tag{7.6}$$

37. Nous avons défini un schéma d'induction sur la sorte `Id`, dont les constructeurs sont `firstId` et `nextId` (cf.chapitre 2,p. 45)

La preuve est complètement automatique car elle ne fait appel qu'à des propriétés sur les expressions booléennes. En effet, cette propriété est réécrite de la façon suivante:

```
stable(p,User)
→  unless(p,F,User)
→  [p \=> wp(startread,p) = T]
   /\ [p \=> wp(endread,p) = T]
   /\ [p \=> wp(startwrite,p) = T]
   /\ [p \=> wp(endwrite,p) = T]
   /\ [p \=> wp(set_b,p) = T]
```

Pour les actions `s` qui ne modifient pas le prédicat `p`, le terme `p \=> wp(s,p)` est une tautologie éliminée par LP. Seules les actions *startwrite* et *set_b* sont pertinentes pour cette preuve.

Étudions les principales étapes:

```
  [p \=> wp(startwrite,p) = T]
    /\ [p \=> wp(set_b,p) = T]
```

En utilisant la définition de `wp` et des substitutions, LP obtient la formule suivante:

```
[(b \=> nq > 0)
 \=> ((((t < N) ^ (nq > 0)) \=> (F \=> (nq-1 > 0)))
       ^ (nnot((t < N) ^ (nq > 0)) \=> (b \=> nq > 0)))=T]
/\ [(b \=> nq > 0) \=> (nq > 0 \=> nq > 0) = T]
```

puis, il simplifie la formule en utilisant les règles définies sur les expressions de prédicats et les connecteurs logiques associés (Chapitre 2, paragraphe 2.3):

```
[(b \=> nq > 0)
 \=> ((((t < N) ^ (nq > 0)) \=> T)
       ^ (nnot((t < N) ^ (nq > 0)) \=> (b \=> nq > 0)))=T]
/\ [(b \=> nq > 0) \=> T = T]
→
[(b \=> nq > 0)
 \=> ((nnot((t < N) ^ (nq > 0)) \=> (b \=> nq > 0)))=T]
/\ [T = T]
→
[ T  ^ (T  \or ((t < N) ^ (nq > 0))) = T] /\ [T = T]
→
[T = T]
```

En conclusion, le théorème prouvé est réduit à `true`, et donc il est éliminé de la base. Ceci est important car il ne subsiste aucune trace de ce théorème dans le système. Par conséquent, nous ne pouvons plus l'utiliser dans les preuves ultérieurement.

### Preuve de la propriété 7

La propriété suivante exprime le fait que si des écrivains sont en attente, alors tôt ou tard, un écrivain sera en écriture ou bien la variable $b$ aura la valeur *vrai*.

$$nq > 0 \textbf{ ensures } nw = 1 \vee b \tag{7.7}$$

En utilisant la définition de *ensures*, ce théorème est subdivisé en deux sous buts:

$$[nq > 0 \textbf{ unless } nw = 1] \;\&\; [\exists\ a\ in\ User :: \{nq > 0 \wedge \neg(nw = 1)\}\ a\ \{b \vee nw = 1\}]$$

Ce qui s'écrit dans notre système:

$$unless(nq > 0, nw = 1, User)\ \&\ exist\_Act(nq > 0, nw = 1, User)$$

La preuve consiste d'une part à prouver la partie *unless*, et d'autre part à montrer qu'il existe une action dans le programme qui satisfait la partie *exist_Act*.

**Unless:** Nous commençons par prouver le lemme suivant:

```
nnot(nw = 1) => nw = 0
```

Pour ce faire, nous utilisons les invariants 7.1 et 7.2, introduits comme axiomes:

```
set name invariant1
assert
(id_to_nat(nw) <= id_to_nat(nn));
(id_to_nat(nr) <= id_to_nat(nn));
(nat_to_exp(0) <= id_to_exp(nr)) = T;
(nat_to_exp(0) <= id_to_exp(nw)) = T;
..
set name invariant2
ass
(id_to_exp(nw) <= nat_to_exp(1))
 ^ ((id_to_exp(nr) \= nat_to_exp(0)) \or (id_to_exp(nw) \= nat_to_exp(0)))=T
```

**Remarque 7.2** *Pour l'invariant 1, nous avons utilisé deux types d'opérateurs de conversion* `id_to_nat` *pour les deux premières et* `id_to_exp` *pour les deux suivantes. Par conséquent, l'opérateur relationnel* `<=` *ne possède pas la même signature dans les deux expressions. En fait, les deux premières équations ont été introduites sous la forme de propriétés sur des expressions.* LP *les normalisent en des propriétés sur des entiers, à l'aide des propriétés des fonctions de conversion (cf. p. 2.4.1). De plus, grâce à cette correspondance, les deux derniers axiomes de l'invariant 1 sont éliminés (les entiers naturels sont positifs ou nuls). Par contre, l'invariant 2 est subdivisé en deux propriétés, où* LP *utilise la correspondance entre l'opérateur* `^` *(du langage objet) et l'opérateur* `/\` *(de la méta-théorie):*

```
invariant2.1: id_to_nat(nw) <= s(0) -> true
invariant2.2: (id_to_exp(nr) \= nat_to_exp(0))
              \or (id_to_exp(nw) \= nat_to_exp(0)) -> T
```

Revenons à présent à la preuve du lemme. En LP, cette preuve est la suivante:

```
set name lemme
prove
  nnot(nat_to_exp(id_to_nat(nw))\= nat_to_exp(s(0)))
   \=> (nat_to_exp(id_to_nat(nw))\= nat_to_exp(0)) = T
  ..
  resume by case id_to_exp(nw) = nat_to_exp(s(0)),id_to_nat(nw) = 0
    <> case justification
    [] case justification
    <> case id_to_exp(nw) = nat_to_exp(s(0))
    [] case id_to_exp(nw) = nat_to_exp(s(0))
    <> case id_to_nat(nw) = 0
    [] case id_to_nat(nw) = 0
  [] conjecture
```

Nous procédons par cas, suivant que `nw=0` ou bien `nw=1` : le premier sous-but consiste à montrer que les cas sont "justifiés", déduit par LP automatiquement, en utilisant la conjonction des deux invariants 1 et 2 qui donne `0 <= nw <= 1`.

Ce lemme nous permettra de montrer la propriété (7), dont nous allons détailler la preuve pour l'action `startwrite`.

Le théorème à prouver est le suivant:

```
unless(nq > 0, nw=1 \or b, User)
```

En utilisant la définition de *unless*, de `wp` et des substitutions, les principales étapes de la preuve sont:

```
nq > 0 ^ nnot(nw=1) ^ nnot(b)
 \=> ((((t >= N) ^ (nq > 0))
          \=> wp(startwrite,nq > 0 \or (nw=1 \or b)))
      ^ ((nnot(t >= N) \or nnot(nq > 0))
          \=> (nq > 0 \or (nw = 1 \or b))))
→
(nq > 0 ^ nnot(nw=1) ^ nnot(b) ^ (t >= N) ^ (nq > 0))
   \=> (((nq-1) > 0) \or (nw+1 = 1) \or F)
→
(nq > 0 ^ nnot(nw=1) ^ nnot(b) ^  (t >= N) ^ (nq > 0))
 \=> (((nq-1) > 0) \or (nw = 0))
```

En utilisant la définition de \=>, ce terme est réécrit en:

```
(nnot(nq > 0) \or (nw=1) \or b \or  nnot(t >= N) \or nnot(nq > 0))
 \or (((nq-1) > 0) \or (nw = 0))
```

La preuve est alors achevée en utilisant le lemme précédent:

```
nnot(nw=1) \=> nw=0
```

Étudions maintenant la preuve LP correspondante:

```
set name prop7
prove
  unless(id_to_exp(nq) > nat_to_exp(0),
         (id_to_exp(nw)\= nat_to_exp(s(0))) \or id_to_bexp(bo),
         User)
  ..
  set name lemma_prop7
  set immunity on
  prove
   nat_to_exp(s(id_to_nat(nw)))\= nat_to_exp(s(0))
      = nat_to_exp(id_to_nat(nw)) \= nat_to_exp(0)
    [] conjecture
  set immunity off
  normalize lemma_prop7 with read_write
  [] conjecture
qed
```

La preuve est automatique pour les actions autres que `startwrite`, dont les étapes n'apparaissent pas dans le script. Par analogie à la preuve manuelle, LP utilise de façon transparente le lemme `nnot(nw=1) \=> nw=0` qui a été orienté en une règle de réécriture:

```
(id_to_exp(nw) \= nat_to_exp(s(0))) \or (id_to_exp(nw) \= nat_to_exp(0)) -> T
```

Nous avons montré le lemme trivial `lemme_prop7`, `(s(nw) = s(0)) <=>(nw = 0)`, écrit sous forme d'une propriété sur les expressions. Ce lemme doit nous permettre de réécrire une formule de la forme

```
A \or (id_to_exp(nw) \= nat_to_exp(s(0)))
  \or (id_to_exp(nw) \= nat_to_exp(0))
  \or  B
```

en `A \or B`

En effet, bien que la propriété soit triviale, LP ne peut l'utiliser sous la forme `(s(i)=1) <=> (i=0)` pour réécrire la formule précédente, car cette propriété manipule des objets de type *entiers naturels* et non de type *expression*. Bien que les fonctions de conversions nous permettent de ramener une propriété sur les expressions arithmétiques en une propriété sur les entiers, elles ne peuvent s'appliquer que pour des faits établis:

```
when (nat_to_exp(n)=id_to_exp(id1)) yield (n=id_to_nat(id1));
```
*ou bien*
```
(nat_to_exp(n)=id_to_exp(id1)) -> (n=id_to_nat(id1))
```

Ainsi, lorsque le fait `(nat_to_exp(s(0))=id_to_exp(nw))` est présent dans la base, LP infère `(0=id_to_nat(nw))`. Ce qui lui permet de montrer le lemme `lemme_prop7` de façon automatique. En immunisant ce lemme avant de le montrer, cela nous permet de le récupérer sous sa forme "expression" sous laquelle il a été introduit, pour réécrire la formule vue plus haut en `A \or B`. Comme `lemme_prop7` a été prouvé en utilisant `nw`, nous devons le normaliser avec les règles `read_write`, qui définissent les variables du programmes et leur affecte un "nom", en particulier pour réécrire `nw` en `nextid(firstId)`. En effet, le but sur lequel s'arrête LP est:

```
Conjecture prop7.2:
  unless(id_to_exp(nq) > nat_to_exp(0),
         (id_to_exp(nw) \= nat_to_exp(s(0))) \or id_to_bexp(bo),
         User)
  Current subgoal:
    (nat_to_exp(0)
       < nat_to_exp(id_to_nat(nq)-s(0)))
     \or (id_to_exp(nw)
             \= nat_to_exp(s(0)))
     \or (nat_to_exp(s(id_to_nat(nw)))
             \= nat_to_exp(s(0)))
     \or id_to_bexp(bo)
     \or nnot(nat_to_exp(0)
                < id_to_exp(nq))
     \or nnot(id_to_exp(nr)
                \= nat_to_exp(0))
     \or nnot(id_to_exp(nw)
                \= nat_to_exp(0))
     = T
     /\ (nat_to_exp(0)
           <
          nat_to_exp(id_to_nat(nq)
                       -s(0)))
         \or (id_to_exp(nw)
                \= nat_to_exp(s(0)))
         \or (nat_to_exp(s(id_to_nat(nw)))
```

```
        \= nat_to_exp(s(0)))
\or id_to_bexp(bo)
\or nnot(nat_to_exp(0)
           < id_to_exp(nq))
\or nnot(id_to_exp(nr)
        \= nat_to_exp(0))
\or nnot(id_to_exp(nn) \= nat_to_exp(0))
= T
```

Pour réduire ce but à `true`, LP utilisera les deux lemmes prouvés `nnot(nw=1) \=> (nw=0) = T` et `(s(nw) = s(0) <=> nw = 0)`, sous forme d'expressions et orientés en règles de réécriture:

```
(nat_to_exp(s(id_to_nat(nw)))\= nat_to_exp(s(0)))
     -> (nat_to_exp(id_to_nat(nw)) \= nat_to_exp(0))
```

```
(id_to_exp(nw) \= nat_to_exp(s(0)))
     \or (id_to_exp(nw) \= nat_to_exp(0)) -> T
```

Ceci achève la preuve du sous-but *Unless* de la propriété 7.

**Exist_Act**: Il s'agit de montrer qu'il existe une action dans le programme qui nous permet de passer des états où $nq > 0 \wedge \neg(nw = 1) \wedge bo$ à un état où $nw = 1 \vee bo$, c'est à dire telle que

```
(nq > 0 ^ nnot(nw=1) ^ bo) \=> wp(s,nw=1 \or bo)
```

L'action qui vérifie cette propriété est `Set_b`. En effet, en utilisant la définition de `wp` et de la substitution, LP simplifie la propriété à `T`:

```
((nq > 0) ^ nnot(nw=1) ^ bo)  \=> (nw=1 \or (nq > 0)) = T
→
(nnot(nq > 0) \or (nw=1) \or nnot(bo)) \or (nw=1 \or (nq > 0)) = T
→ T \or (nw=1) \or nnot(bo) = T
→ true
```

□

### Preuve de la propriété 8

Étudions maintenons la preuve de la propriété de progrès "si un écrivain est en attente et qu'aucun lecteur n'est en lecture alors nécessairement, un des écrivains sera en écriture".

$$b \wedge nr = 0 \textbf{ ensures } nw = 1 \tag{7.8}$$

La première tentative de la preuve automatique échoue. Plus précisément, en essayant de prouver la partie *unless*, LP s'arrête sur le sous-but suivant

```
Conjecture prop8.6:
  unless(id_to_bexp(bo) ^ (id_to_exp(nr) \= nat_to_exp(0)),
         id_to_exp(nw) \= nat_to_exp(s(0)),
         cons(endread2, nil))
  Current subgoal:
    (id_to_exp(nw) \= nat_to_exp(s(0)))
      \or (nat_to_exp(id_to_nat(nr) - s(0)) \= nat_to_exp(0))
      \or nnot(id_to_exp(nr) \= nat_to_exp(0))
      \or nnot(id_to_bexp(bo))
      = T
```

Étudions la preuve manuelle correspondante: en utilisant la définition de *ensures*, la propriété est réécrite en:

$$[b \wedge nr = 0 \textbf{ unless } nw = 1]$$
$$\& \; [\exists\, a \textit{ in } User :: \{b \wedge nr = 0 \wedge \neg(nw = 1)\}\; a \;\{nw = 1\}]$$

Une première étape consiste à prouver que pour toute action $a$ dans $User$, on a:

$$(b \wedge nr = 0 \wedge \neg(nw = 1)) \Rightarrow wp(a, (b \wedge nr = 0) \vee nw = 1)$$

Soit $a$ l'action *endread*, $nr := nr - 1$. Nous appliquons la définition de $wp$:
$(b \wedge nr = 0 \wedge \neg(nw = 1)) \Rightarrow ((b \wedge (nr - 1 = 0)) \vee nw = 1)$
simplifiée en :
$(b \wedge nr = 0 \wedge \neg(nw = 1)) \Rightarrow ((b \wedge nr = 1) \vee nw = 1)$

Il est clair que cette propriété n'est pas valide.

Essayons maintenant de raisonner en termes d'*états*. L'état vérifiant $(b \wedge nr = 0 \wedge \neg(nw = 1))$ est un état valide vérifiant les invariants (1-3). Partant d'un tel état, l'action $nr := nr - 1$ peut être choisie et exécutée. L'état résultant $b \wedge nr = -1 \wedge \neg(nw = 1)$ ne satisfait pas l'invariant 1. Il faut noter que Chandy et Misra dans [Cha88] stipulent que les actions de lecture et d'écriture peuvent avoir d'autres conditions que celles qui apparaissent dans le programme $User$. Pour expliquer la démarche à suivre, c'est à dire les principales étapes de la preuve, nous n'avons pas besoin d'ajouter de conditions. Par contre, du moment que nous nous proposons d'effectuer des preuves formelles, il est nécessaire et même indispensable de modifier les pré-conditions des actions. D'autant plus que la propriété en question permet de prouver la propriété de progrès (5), qui nous garantit que s'il existe des écrivains en attente, ils finiront par avoir accès au fichier.

La solution consiste à transformer les actions *endread* et *endwrite* en affectations conditionnelles avec les deux conditions,$\neg(nr = 0)$ et $\neg(nw = 0)$. C'est la raison pour laquelle le programme donné initialement diffère de celui proposé par les auteurs. Les propriétés prouvées jusqu'ici restent valides par rapport à cette modification.

Revenons à présent à la propriété 8. Avec cette modification, la preuve est automatique. Les étapes de réécriture sont les suivantes:

```
(b ^ nr=0 ^ nnot(nw=1)) \=> wp(endread,(b ^ nr=0) \or nw=1) = T

→ (b ^ nr=0 ^ nnot(nw=1))
  \=> ((nnot(nr=0) \=> (b  ^ (nr-1=0)) \or nw=1)
      ^ ((nr=0) \=> (b  ^ nr=0) \or nw=1)) = T

→ [(b ^ nr=0 ^ nnot(nw=1) ^ nnot(nr=0))
    \=> ((b  ^ (nr-1=0)) \or nw=1)
  ^ (b ^ nr=0 ^ nnot(nw=1) ^ (nr=0))
    \=> ((b  ^ (nr=0)) \or nw=1)] = T

→ (F \=> ((b  ^ nr-1=0) \or nw=1)) ^ T ] = T
→ true
```

Pour l'action *startwrite*, la preuve est identique à celle de la propriété 7. Cette similitude nous permet la réutilisation de la preuve. Il est évident que le lemme utilisé (`lemme_prop7`) peut être prouvé auparavant et utilisé pour la preuve de la propriété 7 et de la propriété 8, mais pour des

raisons de clarté et pour pouvoir corriger des preuves en cas de modifications, cette structure, qui spécifie explicitement les lemmes, est plus adéquate. De plus, l'utilité de ces lemmes (qui généralement spécifient des propriétés triviales) n'apparaît que dans le contexte de la preuve pour la réécriture du but.

Le sous-but *exist_Act* est résolu avec l'action *startwrite*. Ceci est complètement transparent à l'utilisateur, vu que la preuve nécessite la preuve du lemme `lemme_prop7`, introduit pour la preuve de la partie *unless*. Le but est normalisé à `T` de façon transparente.

### Preuve de la propriété 9

La propriété de sûreté 9 garantit que si un certain nombre de lecteurs sont en lecture et qu'un écrivain est en attente alors le nombre de lectures ne peut que diminuer:

$$b \wedge nr = j \wedge j > 0 \textbf{ unless } b \wedge nr < j \tag{7.9}$$

La preuve de cette propriété pour l'action *startread* est triviale du fait de la présence de `nnot(b)` dans la condition de l'action. De même, pour l'action *endwrite* qui ne modifie ni `p` ni `q`.

Étudions la preuve pour l'action *endread*, où la variable *nr* est incrémentée de 1. Toujours en utilisant les définitions dans le système courant, LP suspend la preuve sur le but suivant:

```
bo ^ nr=j ^ (j > 0) ^ nnot(nr < j) ^ nnot(nr=0)
 \=> ((bo ^ (nr-1=j) ^ (j > 0)) \or (((nr-1) < j) ^ (j > 0)))
```

Pour résoudre ce but, il suffit de prouver la propriété suivante:

```
(nr=j ^ (j > 0)) \=> (nr-1 < j)
```

Pour la preuve LP, le but est formulé sous forme d'expression, c'est à dire une expression booléenne *sous forme normale*. Par conséquent, pour terminer la preuve il faut réduire à `T` la formule:

```
((nat_to_exp(0) < id_to_exp(j)) ^ (nat_to_exp(nr) \= id_to_exp(j)))
   \=> ((nat_to_exp(nr) - nat_to_exp(s(0))) < id_to_exp(j))
```

Il faut noter que `j` est une variable de preuve de type `Id_of_var` (variable auxiliaire de preuve) et non `Nat`.

Une première étape consiste à prouver un lemme trivial pour deux variables `m` et `n` de type `nat`.

```
prove (m=n /\ (n > 0)) => (m-1 < n)  by induction on n using nat.3
```

Puis nous utilisons la règle de réécriture suivante:

```
(((nat_to_exp(0) < nat_to_exp(n)) ^ (nat_to_exp(m) \= nat_to_exp(n)))
   \=> ((nat_to_exp(m) - nat_to_exp(s(0))) < nat_to_exp(n))) = T
 ->
   ((n > 0) /\ (m=n)) => (m-1 < n))
```

Cette propriété permet de "prouver" un propriété sur les *expressions* en utilisant la propriété correspondante sur les *entiers*. Par conséquent, une fois le lemme sur les entiers prouvé, cette règle de réécriture sera normalisée à

```
(((nat_to_exp(0) < nat_to_exp(n))nat_to_exp(n)))
   \=> ((nat_to_exp(m) - nat_to_exp(s(0))) < nat_to_exp(n))) = T
-> true
```

En instanciant la variable `m` par le terme `id_to_nat(nr)` et la variable `n` par le terme `id_to_nat(j)`, nous obtenons la propriété voulue.

La preuve de la propriété 9 comprend ainsi deux sous-preuves similaires pour les actions *endread* et *startwrite*, où chacune comprend un lemme de simplification des expressions prouvé à l'aide d'une propriété sur les entiers. Cette similitude nous permet d'utiliser la même structure où seuls les lemmes diffèrent:

```
set name prop9
prove
  unless(id_to_bexp(bo)^ (id_to_exp(nr)\= id_to_exp(j))
            ^ (id_to_exp(j) > nat_to_exp(0)),
         id_to_bexp(bo)^ (id_to_exp(nr) < id_to_exp(j)),User)
  ..
%----------------------endread-
  set immunity on
  set name lemme_prop9
  make passive exp_to_nat_or_id_prop.1:2
  prove
    ((nat_to_exp(0)  < nat_to_exp(id_to_nat(j)))
     ^ (nat_to_exp(id_to_nat(firstId)) \= nat_to_exp(id_to_nat(j))))
    \=> ((nat_to_exp(id_to_nat(firstId)) - nat_to_exp(s(0)))
           < nat_to_exp(id_to_nat(j)))
    = T
    ..
    assert
      (((nat_to_exp(0)  < nat_to_exp(j))
        ^ (nat_to_exp(i) \= nat_to_exp(j)))
        \=> ((nat_to_exp(i) - nat_to_exp(s(0)))  < nat_to_exp(j)) = T)
      <=> (((i=j) /\ (0  < j)) => ((i-1)  < j))
      ..
    set name lemme2
    prove ((i=j) /\ (0  < j)) => ((i-1)  < j)
      resume by induction on j using nat.3
        <> basis subgoal
        [] basis subgoal
        <> induction subgoal
        [] induction subgoal
      [] conjecture
    set immunity off
    rewrite lemme_prop9.2 with lemme2.1
    instantiate j by id_to_nat(j),i by id_to_nat(firstId) in lemme_prop9
    [] conjecture
  set immunity off
  normalize lemme_prop9 with bool*
  rewrite lemme_prop9 with exp_to_nat_prop.1:3
  normalize lemme_prop9 with exp_to_nat_or_id_prop.1:2
%----------------------startwrite
  set immunity on
  set name lemme_prop93
  make passive exp_to_nat_or_id_prop.1:2
  prove
    ((nat_to_exp(0)  < nat_to_exp(id_to_nat(j)))
     ^ (nat_to_exp(id_to_nat(firstId)) \= nat_to_exp(id_to_nat(j))))
    \=> nnot(nat_to_exp(id_to_nat(firstId)) \= nat_to_exp(0))
```

```
    = T
    ..
    assert
      (((nat_to_exp(0)  < nat_to_exp(j))
        ^ (nat_to_exp(i) \= nat_to_exp(j)))
        \=> nnot(nat_to_exp(i) \= nat_to_exp(0)) = T)
      <=> (((i=j) /\ (0  < j)) => ¬(i=0))
      ..
    set name lemme2
    prove ((i=j) /\ (0  < j)) => ¬ (i=0)
      [] conjecture
    set immunity off
    rewrite lemme_prop93.2 with lemme2.1
    instantiate j by id_to_nat(j),i by id_to_nat(firstId) in lemme_prop93
    [] conjecture
  set immunity off
  normalize lemme_prop93 with bool*
  normalize lemme_prop93 with exp_to_nat_or_id_prop.1:2
  [] conjecture
qed
```

La ligne de commande `make passive exp_to_nat_or_id_prop.1:2`, nous permet de dés-activer les règles de réécriture nommées, afin qu'elle ne soient pas utilisées pour réécrire le lemme `lemme_prop9`, puisque nous en avons besoin sous sa forme originale. Ce sont les deux règles d'abréviations qui permettent de réécrire les termes `nat_to_exp(id_to_nat(id1))` en `id_to_exp(id1)` (cf. Chapitre 2, paragraphe 2.4.1). Nous pouvons noter que la stratégie utilisée pour l'action *startwrite* est la même que celle pour l'action *endread.*

Ceci achève les preuves des propriétés (6-9), nous décrivons dans le paragraphe suivant les preuves des propriétés de vivacité qui se ramènent généralement à des preuves de sûreté et à l'utilisation des règles d'inférence de l'opérateur *leads_ to.*

### 7.4.2 Propriété de vivacité

Les propriétés de vivacité que le programme *User* doit satisfaire sont les propriétés (10) à (13) de la preuve présentée au paragraphe 7.2. Le but de ce qui suit est de montrer le niveau de détail requis pour la certification d'une preuve une fois sa structure établie. Nous présentons dans ce qui suit la structure de la preuve de la propriété (11), ayant la propriété (10). Il faut noter que dans la preuve originale, l'étape (10)-(11) est obtenue en appliquant "l'induction", alors qu'elle nécessite la vérification de 12 propriétés:

**Preuve:**

$b \wedge nr = k \wedge k > 0 \mapsto b \wedge nr < k$ (10)

PSP_rule avec (4) et (9).

$b \wedge nr = k \wedge nr > 0 \Rightarrow b \wedge nr = k \wedge k > 0$ (10.1)

$b \wedge nr = k \wedge nr > 0 \mapsto b \wedge nr = k \wedge k > 0$ (10.2)

IMPL_rule avec (10.1)

$b \wedge nr = k \wedge nr > 0 \mapsto b \wedge nr < k$ (10.3)

TRANS_rule sur (10.2) et (10)

$(b \wedge nr < k) \Rightarrow (b \wedge nr < k \wedge (nr = 0 \vee nr > 0))$ (10.4)

*propriété* sur les entiers.

$(b \wedge nr < k) \Rightarrow (b \wedge nr < k \wedge (nr = 0)) \vee (b \wedge nr < k \wedge nr > 0)$ (10.5)

*Distributivité.*

$(b \wedge nr < k) \mapsto (b \wedge nr < k \wedge (nr = 0)) \vee (b \wedge nr < k \wedge nr > 0)$ (10.6)

IMPL_rule avec (10.5)

$(b \wedge nr < k \wedge (nr = 0)) \vee (b \wedge nr < k \wedge nr > 0)$
$\Rightarrow (b \wedge nr < k \wedge nr > 0) \vee (b \wedge nr = 0)$ (10.7)

*propriété* de $\wedge$ et $\vee$.

$(b \wedge nr < k \wedge (nr = 0)) \vee (b \wedge nr < k \wedge nr > 0)$
$\mapsto (b \wedge nr < k \wedge nr > 0) \vee (b \wedge nr = 0)$ (10.8)

IMPL_rule avec (10.7)

$(b \wedge nr < k) \mapsto ((b \wedge nr < k \wedge nr > 0) \vee (b \wedge nr = 0))$ (10.9)

TRANS_rule sur (10.6) et (10.8)

$b \wedge nr = k \wedge nr > 0 \mapsto ((b \wedge nr < k \wedge nr > 0) \vee (b \wedge nr = 0))$ (10.10)

TRANS_rule sur (10.3) et (10.9)

$b \wedge nr > 0 \mapsto b \wedge nr = 0$ (10.11)

IND sur (10.10)

$b \wedge nr = 0 \mapsto b \wedge nr = 0$ (10.12)

IMPL_rule.

$b \mapsto b \wedge nr = 0$ (11)

DISJ_unless sur (10.11) et (10.12)

**finpreuve**

Nous ne décrivons pas les preuves LP des propriétés (10) à (10.10), qui se ramènent à utiliser des règles d'inférence de l'opérateur `leads_to`. Cela consiste à instancier les variables dans les règles par les termes adéquats afin d'obtenir le but recherché. De plus, ces preuves nécessitent des lemmes intermédiaires pour faciliter la manipulation des expressions booléennes.

Par exemple, la preuve LP de la propriété (10) est:

```
set name prop10
prove
leads_to(id_to_bexp(bo) ^ (id_to_exp(nr)\= id_to_exp(j))
          ^ (id_to_exp(j) > nat_to_exp(0)),
         id_to_bexp(bo) ^ (id_to_exp(nr) < id_to_exp(j)),User)
..
set name lemme_prop10
prove (p \or q) ^ (p \or nnot(q)) = p
  set immunity on
  prove  (p \or q) ^ (p \or nnot(q)) = (p \or (q ^ nnot(q)))
    [] conjecture
  cri-p prop10 with bool_str*
  [] conjecture

instantiate
p by (id_to_exp(nr) \= id_to_exp(j)) ^ (id_to_exp(j) > nat_to_exp(0)),
q by  nnot(id_to_exp(nr) \= id_to_exp(j)),
pgm by User,
r by id_to_bexp(bo) ^ (id_to_exp(nr)\= id_to_exp(j))
     ^ (id_to_exp(j) > nat_to_exp(0)),
b by id_to_bexp(bo) ^ (id_to_exp(nr) < id_to_exp(j)) in PSP_lead
..
  [] conjecture
qed
```

Nous montrons un lemme sur les expressions booléennes, nécessaire du fait que les formules sont normalisées en forme normale. En effet si nous appliquons la règle d'inférence *PSP_rule* (cf. p.4.4.4), sur les deux propriétés suivantes:

```
leads_to((nr \= j) ^ (j > 0), nnot(nr \= j),User)
```

```
unless(bo ^ (nr \= j) ^ (j > 0), bo ^ (nr < j),User)
```

Nous obtenons la formule:

```
leads_to(bo ^ (nr \= j) ^ (j > 0),
         ((bo ^ (nr < j)) \or (bo ^ (nr \= j) ^ (j >  0)))
         ^ ((bo ^ (nr < j)) \or nnot(nr \= j)),
         User)
```

puis cette formule est réécrite de façon à ce que on ait une conjonction de disjonctions:

```
leads_to(bo ^ (nr \= j) ^ (j > 0),
         ((nr < j) \or (0 < j))
          ^ ((nr < j) \or (nr \= j))
          ^ ((nr < j) \or nnot(nr \= j))
          ^ bo,
          User)
```

Pour avoir le résultat cherché, il nous faut montrer que le terme

```
((nr < j) \or (nr \= j)) ^ ((nr < j) \or nnot(nr \= j))
```

se réécrit en `(nr <j)`. Pour ce faire, une fois le lemme `lemme_prop10` prouvé, il suffit d'instancier ses variables par les termes adéquat. Ce lemme, bien qu'il semble trivial, doit être prouvé à cause de l'orientation des règles de réécriture exprimant la distributivité de ^ par rapport à \or. A l'inverse des lemmes utilisés pour les expressions arithmétiques du paragraphe précédent, le lemme `lemme_prop10` est extrait de la preuve de la propriété pour être ajouté à la base de règles, car il exprime une propriété générale de simplification.

### 7.4.3 Le principe d'induction du Leads_to

Le principe d'induction associé à la relation *leads_to* est formalisé en LP par l'axiome suivant:

```
Set name IND_leads_to
assert
leads_to((p ^ (exp_list1 == exp_list2)),
         (p ^ (exp_list1 ≪ exp_list2)) \or q,
          pgm)
=> leads_to(p,q,pgm)
```

où `exp_list1` et `exp_list2` sont des listes d'expressions. La fonction `==:exp_list,exp_list -> Bexp`, teste l'égalité de deux listes d'expressions et `<<:exp_list,exp_list -> Bexp` spécifie un ordre lexicographique sur des listes d'expressions (cf. chapitre 2,p. 64).
Ce principe, appliquer à la propriété (10.10) :

$$b \wedge nr = k \wedge nr > 0 \mapsto ((b \wedge nr < k \wedge nr > 0) \vee (b \wedge nr = 0))$$

nous permet de montrer la propriété (10.11):

$$b \wedge nr > 0 \mapsto b \wedge nr = 0$$

Le théorème LP correspondant à la propriété (10.10) est:

```
prop10_10.1(I): leads_to(
                (id_to_exp(nr) > nat_to_exp(0))
                 ^ (id_to_exp(nr) \= id_to_exp(j))
                 ^ id_to_bexp(bo),
                ((id_to_exp(nr) < id_to_exp(j))
                  ^ (id_to_exp(nr) > nat_to_exp(0))
                  ^ id_to_bexp(bo))
                 \or  (id_to_bexp(bo)
                    ^ (id_to_exp(nr) \= nat_to_exp(0))),
                User)
                -> true
```

L'application du principe d'induction revient à instancier les variables `p` par `(bo ^ (nr >0))`, `q` par `(bo ^ (nr =0))`, la liste d'expressions `exp_list1` par `cons(nr,enil)` et `exp_list2` par `cons(j,enil)`.

Du point de vue sémantique, la propriété (10.10) garantit que partant d'un état où $bo \wedge (nr > 0)$ est valide et $(nr = j)$, l'exécution du programme atteint fatalement un état où $bo \wedge (nr = 0)$ est valide, ou bien un état où $bo \wedge (nr > 0)$ est valide et $(nr < j)$. Étant donné que la valeur de $nr$ ne peut décroître indéfiniment, on atteint fatalement un état où $bo \wedge (nr = 0)$ est valide. D'ou la propriété (10.11).

La preuve LP correspondante est:

```
% b^ nr=j^ nr >0 -> (b ^ nr=0)
set name prop10_11
set immunity on
prove
  leads_to(id_to_bexp(bo) ^ (id_to_exp(nr) > nat_to_exp(0)),
           (id_to_bexp(bo) ^ (id_to_exp(nr) \= nat_to_exp(0))),
           User)
  ..
  set immunity off
  instantiate
    p by id_to_bexp(bo) ^ (id_to_exp(nr) > nat_to_exp(0)),
    q by (id_to_bexp(bo)^ (id_to_exp(nr) \= nat_to_exp(0))),
    pgm by User,
    exp_list1 by cons(id_to_exp(nr),enil),
    exp_list2 by cons(id_to_exp(j),enil)
    in IND_leads_to
    ..
  [] conjecture
qed
set immunity off
```

La commande `set immunity on` permet d'immuniser la propriété `prop10_11`, de telle sorte qu'une fois prouvée, elle est conservée dans la base sous sa forme originale. En effet, la stratégie globale utilisée pour ces différentes preuves des propriétés de vivacité consiste à immuniser à chaque étape la propriété avant de la prouver afin qu'elle puisse être utilisée dans l'étape qui suit sous sa forme originale. Cela facilite la lisibilité des théorèmes prouvés, la détection des erreurs et la mise au point des preuves.

## 7.5 La preuve mécanisée en HOL-Unity

Cet exemple "simple" a été également vérifié par Flemming Andersen, à l'aide du système HOL-UNITY, et sa preuve est décrite dans [And92]. La comparaison avec ce système sera approfondie dans les travaux connexes présentés à la fin de ce document, mais nous pouvons dès lors faire quelques remarques sur l'exemple des lecteurs-rédacteurs. Tout d'abord, le système utilisé ne permet que l'utilisation du type entier ce qui nécessite d'exprimer les variables booléennes en variables entières. A chaque variable du programme est associé un entier positif ($nr \rightarrow 0, nw \rightarrow 1, nq \rightarrow 2, b \rightarrow 3$), et ceci dans le but de prouver que toutes les variables sont distinctes. Dans notre système, un processus similaire est effectué, en associant à chaque identificateur du programme le terme $nextid^n(firstid)$, $n$ étant l'ordre chronologique dans lequel ces identificateurs sont "lus". Notre traitement simule la lecture du texte du programme et l'affectation à chaque identificateur lu de la section `declare`, la chaîne de caractères représentant son "nom". Le schéma d'induction définit sur la sorte `Id` nous garantit l'unicité du "nom":

```
Sort Id generated freely by firstId, nextId
```

où l'option `freely` nous fournit deux axiomes supplémentaires:

```
firstId ≠ nextId(id)
nextId(id1) = nextId(id2) <=> id1 = id2
```

En HOL-UNITY, la section *always* d'un programme UNITY est traduite en utilisant la facilité offerte par ML pour définir des abréviations de termes qui peuvent être utilisées dans la définition d'autres termes. Ce qui est sémantiquement équivalent à notre formalisation de la section *always*, puisque la constante, de type `Bexp` utilisée, sera réécrite en sa définition dans toute autre formule.

La preuve d'une propriété *unless* est subdivisée en cinq preuves, une par action, puis une preuve pour la totalité du programme. Dans notre système, bien que des preuves séparées pour chaque action soient possibles, la preuve d'une propriété de sûreté est prouvée pour la totalité du programme. De plus, seules les actions pertinentes requièrent une intervention, la preuve d'une propriété $\{p \wedge \neg q\}s\{p \vee q\}$, pour les actions qui ne modifient pas $p \vee q$ sont des tautologies que LP élimine de façon transparente. Intuitivement, lorsque nous essayons de montrer une propriété pour la liste des actions du programme, nous savons que LP peut s'arrêter sur les actions non triviales, ou bien pour lesquelles la propriété n'est pas satisfaite. Ce qui facilite la mise au point des preuves.

En étudiant les preuves en HOL-UNITY, nous pouvons remarquer que les tactiques utilisées pour chaque action ne sont pas les mêmes. Par analogie, nous pouvons en déduire le degré d'automatisme. L'appel aux différentes tactiques nécessite une preuve manuelle formelle pour décider des tactiques à utiliser, et des théorèmes dont on a besoin. La différence avec notre approche est que dans notre système, les théorèmes prouvés sont disponibles comme règles de réécriture et donc utilisés directement par le démonstrateur.

Enfin, la preuve des deux propriétés intéressantes (8) et (9) dont les preuves ont nécessité la modification du programme, n'ont pas été détaillées dans le document. Par conséquent, nous ne connaissons pas la solution choisie pour montrer ces propriétés, et si le système les a vraiment pris en compte. L'auteur suppose les propriétés (8) et (9) pour montrer la propriété de progrès.

## 7.6 Amélioration

Ce paragraphe décrit l'application des améliorations présentées au chapitre 4. Ces améliorations proposaient une nouvelle définition des opérateurs *ensures* et *leads_to*, qui utilise les

quantificateurs existentiels. En particulier, un programme UNITY n'est plus représenté par une *liste* d'actions mais par un *ensemble* d'actions.

### 7.6.1 Ensures

Prouver une propriété $p$ *ensures* $q$ se ramène à montrer la formule:

```
nnot(p) \or wp(a2,p \or q) \or q = T
 /\  unless(p,q,pgm)
 /\  ((nnot(p) \or wp(a2,q) \or q = T) \/ exist_Act(p,q,pgm))
```

où, le sous-but *exist_Act* consiste à chercher l'action qui vérifie la propriété:

```
(((p ^ nnot(q)) \=> wp(a,q)) = T) \/ exists_act(p,q,act_1)
```

La nouvelle définition de *ensures* est:

```
assert
ensures(p,q,pgm)
  = (unless(p,q,pgm)
     /\  (\E a1 ((a1 \in pgm) /\ (((p ^ nnot(q)) \=> wp(a1,q))=T))))
..
```

Par conséquent, montrer la propriété $p$ *ensures* $q$ se ramène à montrer la formule:

```
nnot(p) \or q \or wp(a2,p \or q) = T
 /\  unless(p,q,pgm)
 /\ \E a1 ((a1 = a2 \/ a1 \in pgm)/\ nnot(p) \or q \or wp(a1,q) = T)
```

Cette définition nous permet de montrer les propriétés *ensures* d'une façon plus intuitive. En effet, les changements que nécessite cette nouvelle formalisation réside simplement dans l'instanciation explicite de l'action qui vérifie la propriété. En effet, le rôle de l'opérateur `exist_act` était de vérifier l'existence d'une instruction dans le programme qui vérifie la transition $\{p \wedge \neg q\} s \{q\}$.

Reprenons la preuve de la propriété (7.7):

$$nq > 0 \textbf{ ensures } nw = 1 \vee b$$

En utilisant la définition de *ensures*, ce théorème est subdivisé en deux sous buts:

$$[nq > 0 \textbf{ unless } nw = 1] \;\&\; [\exists\, a \in User :: \{nq > 0 \wedge \neg(nw = 1)\}\ a\ \{nq > 0 \vee nw = 1\}]$$

Ce qui s'écrit à présent dans notre système:

$$unless(nq > 0, nw = 1, User) \;\&\; (\exists a \;\; ((a \in User) \wedge ((nq > 0) \wedge \neg(nw = 1)) \Rightarrow wp(a, nw = 1 \vee b)))$$

La preuve de cette propriété en LP est alors la suivante:

```
set name prop7
prove
  ensures(id_to_exp(nq) > nat_to_exp(0),
          (id_to_exp(nw)\= nat_to_exp(s(0))) \or id_to_bexp(bo),
           User)
  ..
  set name lemma_prop7
  set immunity on
  prove
```

```
    (nat_to_exp(s(id_to_nat(nw))) \= nat_to_exp(s(0)))
       = (nat_to_exp(id_to_nat(nw)) \= nat_to_exp(0))
    ..
    [] conjecture
  set immunity off
  normalize lemma_prop7 with read_write
  resume by specializing a1 to set_b
  [] conjecture
qed
```

La partie *unless* est identique à la preuve avec la formalisation précédente. Une fois ce sous-but prouvé, il reste à montrer qu'il existe une action dans le programme vérifiant

$$(\exists a \;\; ((a \in User) \wedge ((nq > 0) \wedge \neg(nw = 1)) \Rightarrow wp(a, nw = 1 \vee b)))$$

Ce qui correspond au sous-but:

```
\E a1
   ((a1 = endread
      \/ a1 = set_b
      \/ a1 = startread
      \/ a1 = endwrite
      \/ a1 = stratwrite
      /\ (id_to_exp(nw) \= nat_to_exp(s(0)))
          \or id_to_bexp(bo)
          \or nnot(nat_to_exp(0)
                      < id_to_exp(nq))
          \or wp(a1,
                 (id_to_exp(nw) \= nat_to_exp(s(0)))
                  \or id_to_bexp(bo))
           = T)
```

D'après la preuve manuelle, l'action `set_b` vérifie la propriété cherchée. Par conséquent il suffit de spécialiser la variable quantifiée à cette action pour avoir le résultat (cf. chapitre 1, paragraphe 1.3.2).

### 7.6.2 Leads_to

La nouvelle définition de l'opérateur *leads_to* est plus radicale, dans le sens où elle est plus proche de la définition sémantique de la relation (cf. chapitre 4).

Étudions la preuve de la propriété (10):

$$b \wedge nr = k \wedge k > 0 \mapsto b \wedge nr < k$$

La preuve consiste à appliquer la règle d'inférence

$$PSP\_rule : \frac{p \mapsto q, r \;\; unless \;\; b}{p \wedge r \;\; \mapsto \;\; (q \wedge r) \vee b}$$

formalisée en LP comme suit:

```
Set name PSP_lead
  (newlead(p,q,insert(a1,pgm),n)/\ unless(r,b,insert(a1,pgm)))
     => (\E m ((n < m) /\  newlead(p^ r,(q^ r) \or b,insert(a1,pgm),m)))
  ..
```

La preuve LP correspondante est

```
set name prop10
prove
  \E m newlead(id_to_bexp(bo)
               ^(id_to_exp(nr)\= id_to_exp(j))^(id_to_exp(j) > nat_to_exp(0)),
               id_to_bexp(bo)^(id_to_exp(nr) < id_to_exp(j)),User,m)
  ..
  instantiate
    p by (id_to_exp(nr)\= id_to_exp(j))^(id_to_exp(j) > nat_to_exp(0)),
    q by nnot(id_to_exp(nr)\= id_to_exp(j)),
    a1 by startread,pgm by User1,
    r by id_to_bexp(bo)^(id_to_exp(nr)\= id_to_exp(j))
         ^(id_to_exp(j) > nat_to_exp(0)),
    b by id_to_bexp(bo)^(id_to_exp(nr) < id_to_exp(j)) in PSP_lead
    ..
  declare operators cst2:Nat,Id_of_var-> Nat
  fix m as cst2(n,j) in psp_lead.1.1
  resume by specializing m to cst2(n, j)
qed
```

Nous commençons par "appliquer" la règle `PSP_rule`, en instanciant les variables libres de la règle par les termes adéquats. Ceci nous permet de valider les prémisses de la règle d'inférence (de l'implication =>) et par conséquent d'inférer sa conclusion . Celle-ci contient un quantificateur existentiel. Nous éliminons ce quantificateur en utilisant une fonction de skolem `cst2: Nat,Id_of_var -> Nat`, cela correspond à l'utilisation des hypothèses "quantifiées" (Chapitre 2, paragraphe 4.5).

Intuitivement, la propriété $nr = j \wedge j > 0 \mapsto nr = j$ a été vérifiée par le programme à une ètape $n$, et sachant que la propriété $b \wedge nr = j \wedge j > 0$ *unless* $b \wedge nr < j$ est valide, alors en utilisant la règle `PSP_rule`, on peut inférer $(nr = j \wedge j > 0) \wedge (b \wedge nr = j \wedge j > 0) \mapsto (nr = j \wedge b \wedge nr < j \wedge nr = j \wedge j > 0) \vee (b \wedge nr < j)$,
à une étape qui dépend de $n$.

La formule obtenue `psp_rule.1.1` est identique au but que l'on cherche à prouver, mis à part le quantificateur existentiel présent dans le but. En effet, le but est sous la forme

```
\E m
   newlead(
     (nat_to_exp(0) < id_to_exp(j))
       ^ (id_to_exp(nr) \= id_to_exp(j))
       ^ id_to_bexp(bo),
     id_to_bexp(bo)
       ^ (id_to_exp(nr) < id_to_exp(j)),
     User,
     m)
```

et le fait obtenu à l'aide de la conclusion de la règle `psp_rule.1.1` est

```
prop10.2.2: newlead(
              (nat_to_exp(0) < id_to_exp(j))
                ^ (id_to_exp(nr) \= id_to_exp(j))
                ^ id_to_bexp(bo),
              (id_to_exp(nr) < id_to_exp(j))
```

```
  ^ id_to_bexp(bo),
User,
cst2(n, j))
```

Par conséquent, d'une part, nous devons montrer qu'il existe un indice `m` tel que la propriété soit vraie, et d'autre part, nous savons qu'elle est vraie pour l'indice `cst2(n,j)`. Il suffit alors de spécialiser dans le but la variable quantifiée `m` par le terme `cst2(n, j)` pour déduire le résultat.

## 7.7 Conclusion

Nous avons décrit dans ce chapitre la preuve détaillée et mécanisée du problème classique des lecteurs rédacteurs. Malgré la relative simplicité de l'exemple, la preuve mécanisée nous a montré qu'il ne suffit pas d'établir les principales étapes d'une preuve pour la certifier. En effet, la vérification formelle et surtout sa mécanisation exige une rigueur dans la construction de la preuve et un niveau élevé de détails. Cette approche automatique dirigée par l'utilisateur a plusieurs avantages. D'une part, elle nous permet d'obtenir des preuves certifiées et d'autre part, elle est nécessaire pour toute amélioration ultérieure du programme vérifié.

Il peut sembler gênant que la preuve mécanisée nécessite plus d'efforts que la preuve manuelle. Ces efforts découlent de la définition axiomatique des types de données utilisés. Néanmoins comme nous l'avons vu dans cet exemple, ces définitions font partie de notre système et ne sont pas redéfinis pour chaque application. De plus, ce temps requis pour définir les structures de données diminue à mesure que la base s'enrichit.

Par contre, certaines preuves simples mais fastidieuses sont effectuées automatiquement par LP. L'interaction avec l'utilisateur réside dans l'instanciation des variables par les termes appropriés pour utiliser les règles de preuves. Comme nous l'avons expliqué auparavant, ce processus peut être éliminé si nous utilisons des règles de déduction. Du fait que celles ci permettent plus d'automatisme mais rendent le système plus lent, nous avons choisi la première solution.

Un effort important a été requis pour prouver des lemmes qui établissent des propriétés sur les expressions via les propriétés sur les entiers. Cette technique est réutilisable et par conséquent l'effort requis décroît avec son utilisation. Par ailleurs, ce passage vers les entiers nous permet de ne pas introduire des axiomes sur les expressions arithmétiques de façon aléatoire et de les justifier par des preuves sur les entiers naturels. Bien que certaines propriétés sur les expressions arithmétiques peuvent être montrées directement, nous privilégions le passage par les entiers afin d'éviter la duplication. Nous verrons dans l'exemple de programme de contrôle d'un ascenseur que les preuves sont plus automatiques quand il s'agit de manipuler des expressions booléennes (expressions de prédicats).

# Chapitre 8

# Un protocole de contrôle d'un ascenseur

## Introduction

Le problème étudié dans ce chapitre est emprunté de [And93]. Il décrit un protocole de contrôle d'un ascenseur spécifié en UNITY. La preuve formelle de sa correction fut établie à l'aide du système HOL-UNITY, implémentation de UNITY avec le démonstrateur d'ordre supérieur HOL [Gor93]. Cet exemple nous a intéressé pour plusieurs raisons: tout d'abord, il nous permet une comparaison fructueuse de démonstrateurs pour la mécanisation de la preuve de correction d'un programme parallèle spécifié en UNITY. De plus, c'est l'archétype d'un programme utilisant à la fois des variables d'états *booléennes*, *entières* et de type *tableau*. Par conséquent, il nous permet d'étudier la formalisation de ces données et si possible ses limites. Enfin, bien que le système soit à états finis, il est adéquat pour illustrer notre approche de preuve semi-automatisée de propriétés de sûreté.

Une propriété de progrès que doit vérifier un protocole de contrôle d'un ascenseur est que si l'ascenseur est appelé à un étage donné, alors tôt ou tard, cet étage sera desservi. Néanmoins, nous allons voir dans ce qui suit que d'une part, un tel protocole n'est pas simple à spécifier et que d'autre part, cette propriété de progrès, aussi simple qu'elle paraisse, doit être décomposée en un ensemble de propriétés *de base*, de sûreté et de progrès.

Ce chapitre est organisé comme suit. Nous commençons par introduire le problème en décrivant le protocole de contrôle d'un ascenseur tel qu'il a été proposé dans [And93]. Nous décrivons par la suite les étapes de la mécanisation de la preuve de correction, où nous nous intéressons tout particulièrement à une propriété de sûreté pour laquelle nous décrivons la preuve manuelle informelle et la preuve correspondante en LP. Enfin, nous concluons ce chapitre par une discussion sur cette expérience[38].

## 8.1 Le protocole de contrôle d'un ascenseur

Le programme *lift* décrit le comportement d'un ascenseur qui accède à un certain nombre d'étages pour répondre à des appels. "Répondre à l'appel" d'un étage donné consiste à *déplacer* l'ascenseur jusqu'a l'étage, à *arrêter* l'ascenseur à l'étage et à *ouvrir* les portes. Dans la version originale [And93], l'étage le plus bas et le dernier étage sont spécifiés à l'aide de deux constantes $min$ et $max$. Nous simplifions, sans perte de généralité à trois étages: $min = 0$ et $max = 2$.

Le comportement de l'ascenseur est décrit par sa *position* à un étage, des conditions sur le *déplacement* et l'*ouverture* des portes, et des *indicateurs* des appels. Par conséquent, l'espace

38. Pour cette expérience, nous avons utilisé la version 2.4 de LP, puis la version 3.1.

d'état de l'ascenseur est représenté par six variables, *floor* représente la position courante de l'ascenseur, *open* indique si les portes sont ouvertes à l'étage *floor*, *stop* indique si l'ascenseur est arrêté à l'étage *floor*, $req[i]$ indique s'il y a un appel à l'étage $i$, *up* indique la direction courante du mouvement de l'ascenseur, et enfin *move* indique si le déplacement de l'ascenseur est prioritaire sur l'ouverture des portes. Les quatre premières variables expriment le comportement observable de l'ascenseur alors que les deux dernières permettent de contrôler son comportement.

La propriété de progrès de l'ascenseur est que tout appel doit être tôt ou tard pris en compte et desservi. Le protocole est le suivant: si l'ascenseur est en déplacement dans une direction donnée, il continue son déplacement tant qu'il y a des appels à servir dans cette direction. Au cours de ce déplacement, il dessert tout étage où il y a un appel. Lorsqu'il n'y a plus d'appels dans une direction donnée, l'ascenseur commence à se déplacer dans la direction opposée, s'il y a des appels dans cette direction. Pour éviter que des appels répétés à un même étage ne bloque l'ascenseur à cet étage, son déplacement vers un autre étage où il y a eu un appel est prioritaire par rapport à la ré-ouverture des portes. C'est ce qui est formalisé par la variable *move*.

**Program** {Lift}
**Declare**
$floor : integer$,
$up, move, stop, open : bool$,
$req : array[0..2]\ of\ bool$,
**initially**
$floor = 0$ [] $up, move, stop, open = false, true, true, false$
$req[0], req[1], req[2] = false, false, false$
**Always**
$above = \exists i : floor < i \leq 2 \wedge req[i]$
$below = \exists i : 0 \leq i < floor \wedge req[i]$
$queueing = above \vee below$
$goingup = above \wedge (up \vee \neg below)$
$goingdown = below \wedge (\neg up \vee \neg above)$
$ready = stop \wedge \neg open \wedge move$
**assign**

| | | | |
|---|---|---|---|
| $\{req\_act\}$ | $stop, move$ | $:= true, false$ | $if\ \neg stop \wedge req[floor]$ |
| $[]\{open\_act\}$ | $open, req[floor], move$ | $:= true, false, true$ | $if\ stop \wedge \neg open \wedge req[floor] \wedge \neg(move \wedge queueing)$ |
| $[]\{close\_act\}$ | $open$ | $:= false$ | $if\ open$ |
| $[]\{req\_up\}$ | $stop, floor, up$ | $:= false, floor + 1, true$ | $if\ ready \wedge goingup$ |
| $[]\{req\_down\}$ | $stop, floor, up$ | $:= false, floor - 1, false$ | $if\ ready \wedge goingdown$ |
| $[]\{move\_up\}$ | $floor$ | $:= floor + 1$ | $if\ \neg stop \wedge up \wedge \neg req[floor]$ |
| $[]\{move\_down\}$ | $floor$ | $:= floor - 1$ | $if\ \neg stop \wedge \neg up \wedge \neg req[floor]$ |

**end**{lift}

Le protocole est conçu de telle sorte que l'ascenseur soit, dans l'état initial, arrêté à l'étage le plus bas et qu'il n'y ait aucun appel dans cet état. L'ascenseur demeure dans son état initial, à moins qu'un appel survienne, auquel cas, le programme se "met en route".

La section *Initially* exprime les conditions initiales de l'ascenseur, à savoir qu'il est arrêté à l'étage le plus bas, et qu'il n'y a aucun appel.

Le protocole est décrit par un programme comprenant sept actions. L'action *req_ act* permet l'arrêt de l'ascenseur à l'étage courant s'il y a eu un appel à cet étage. L'action *open_ act* permet l'ouverture des portes lorsque l'ascenseur est arrêté à un étage où il y a un appel et l'ouverture des portes est prioritaire sur le déplacement. L'action *close_ act* ferme les portes, les actions *req_ act* et *req_ down* permettent le déplacement dans une direction donnée lorsque les portes

sont fermées et le déplacement est prioritaire sur leur ouverture et enfin *move_up* et *move_down* maintiennent la direction de déplacement lorsqu'il n'y a pas d'appel à l'étage courant.

## 8.2 Formalisation avec LP

Nous allons décrire dans ce qui suit les principales étapes de la formalisation du programme UNITY *lift* avec LP.

### 8.2.1 Variable d'état et Variable de preuve

Nous avions défini les variables UNITY comme étant de deux types, suivant qu'elles apparaissent dans le texte du programme ou bien qu'elles permettent de spécifier les propriétés à vérifier. Les premières sont donc les variables dynamiques, ou variables d'états, représentant des quantités dont la valeur évolue au cours du temps, et sont représentées en LP comme des constantes de la sorte `Id`. Les secondes sont des quantités statiques, variables auxiliaires qui n'apparaissent que dans le contexte d'une preuve, et sont représentées par des variables LP de la sorte `Id_of_var` (cf. Chapitre 2, paragraphe 2.4.1).

Les variables *floor*, *up*, *move*, *stop*, et *open* sont déclarées en LP comme des constantes de la sorte `Id`.

```
set name lift
declare operators
floor:              -> Id
up,move,stop,open:-> Id
req :               -> Id
```

#### Les tableaux

La formalisation des tableaux doit être rigoureuse en raison des abus de notation fréquemment utilisés pour un tableau. En effet, pour un tableau $A$, qui est un objet informatique, l'expression $A[i]$ représente la i-ème composante du tableau $A$, alors que $A(i)$ est la valeur de cette composante. Par conséquent, un tableau est souvent vu comme une fonction du domaine des indices vers le domaine des valeurs. L'expression $A[i] := e$ est une affectation dont la variable modifiée est $A$ et non pas une hypothétique variable isolée $A[i]$ pour une certaine valeur de $i$. Par conséquent, l'affectation est:

$$A := (\forall i \in Dom(indice) : if \;\; i = j \;\; then \;\; e \;\; else \;\; A(i))$$

Ainsi, le tableau *req* représente une variable du programme susceptible d'être modifiée. Par conséquent, un objet de type tableau est représenté par un identificateur d'une variable de type *tableau*.

Nous définissons alors un tableau en LP comme suit:

```
set name array
dec sort Array
dec op
assign:Array,Id,Bexp ->Array
val   :Array,Id -> Bexp
__[__]:Array,Id -> Id
..
```

Nous représentons les tableaux par la sorte `array` pour laquelle nous définissons trois fonctions. Les deux premières représentent les opérations usuelles sur une structure de type *tableau*:

- `assign(A,idx,v)` affecte à la composante `idx` du tableau, la "valeur" `v`.
- `val(A,idx)` renvoie la "valeur" de la composante `idx` du tableau `A`.
- `A[idx]` renvoi la composante `idx` du tableau.

Par conséquent, nous avons défini un tableau dont chaque composante est un identificateur (puisqu'elle est susceptible d'être modifiée), et dont la valeur est une expression booléenne.

La fonction `val` est alors définie comme suit:

```
declare variables
arr:Array
idx1,idx2:Id
v:Bexp
..
assert
val(assign(arr,idx1,v),idx2)
 = (if id_to_nat(idx1)=id_to_nat(idx2) then v else val(arr,idx2));
```

Le tableau *req* est défini comme un tableau d'identificateurs `Id`, et chaque identificateur représente un variable de type expression booléenne. Par conséquent, le terme `arr[i]` ne représente pas, comme on pourrait le penser, la valeur de la i-ème composante du tableau `arr` mais l'identificateur associé à cette composante. La valeur de la i-ème composante du tableau est représentée par le terme `val(arr,i)`.

### 8.2.2 Le programme

Un programme UNITY est défini comme une liste d'actions et représenté en LP par une constante de la sorte `Actlist`, à laquelle on "affecte" la liste des actions. Comme pour les études de cas décrites précédemment, nous avons choisi de nommer chaque action pour faciliter son utilisation. Ainsi, chaque action sera représentée par une constante de la sorte `Act` à laquelle on associe un objet de type affectation, construit avec l'un des quatre constructeurs de la sorte `Act` (cf. chapitre 2, paragraphe 2.4.2).

Les conditions initiales du programme sont spécifiées dans la section **Initially**, et ne sont autres que des équations sur la valeur initiale des variables d'états. C'est donc un prédicat (d'état) spécifiant les états initiaux. Tout état vérifiant ce prédicat est un état initial potentiel du programme. Pour représenter ce prédicat, nous définissons une constante, `init`, une *expression de prédicat* de la sorte `Bexp`:

```
assert
init =
 (id_to_exp(floor) \= nat_to_exp(0))
 ^  bool_to_bexp(id_to_bexp(up) = F) ^  bool_to_bexp(id_to_bexp(move) = T)
 ^  bool_to_bexp(id_to_bexp(stop) = T) ^  bool_to_bexp(id_to_bexp(open) = F)
 ^  bool_to_bexp(val(id_to_arr(req),min) = F)
 ^  bool_to_bexp(val(id_to_arr(req),moy) = F)
 ^  bool_to_bexp(val(id_to_arr(req),max) = F)
```

Le terme `(bool_to_bexp(id_to_bexp(up)) = F)` représente la condition $up = false$. La fonction `id_to_bexp` convertit une élément de la sorte `Id` en un élément de la sorte `Bexp` et `bool_to_bexp` transforme un booléen en une expression booléenne.

Comme pour les identificateurs (`Id` et `Id_of_var`) et les entiers naturels (`Nat`) pour lesquels nous avions défini des fonctions de conversion des sortes `Id`, `Id_of_var` et `nat` vers la sorte `exp` (`id_to_exp` et `nat_to_exp`, cf. Chapitre 2, paragraphe 2.4.1), nous avons défini une fonction de conversion `Id_to_arr` pour exprimer le fait que `req` est un identificateur d'un objet de type *tableau.*

La section **always** n'est qu'un ensemble d'abréviations, traduite en un ensemble d'axiomes[39]:

```
set name always
assert
queueing = (above \/ below);
goup     = (above /\ ((id_to_bexp(up)=T) \/  ~(below)));
godown   = (below /\ ((nnot(id_to_bexp(up))=T) \/ ~ (above)));
ready    = (id_to_bexp(stop) ^ nnot(id_to_bexp(open)) ^ id_to_bexp(move)=T);
above    = (((id_to_exp(floor) < id_to_exp(ex_i))
             ^ (id_to_exp(ex_i) <= nat_to_exp(max)))
             ^ (val(id_to_arr(req),ex_i)));
below   = (((nat_to_exp(min) <= id_to_exp(ex_j))
             ^ (id_to_exp(ex_j) < id_to_exp(floor)))
             ^ (val(id_to_arr(req),ex_j)))
..
```

La section **assign** comprend un ensemble d'instructions, affectations simples, multiples et/ou conditionnelles. Elle est traduite en une liste d'actions que nous associons à la constante représentant le programme:

```
declare operators
lift                      : -> Actlist          %The programm---%
req_act,req_down,req_up   : -> Act              %The actions---%
open_act,close_act        : -> Act
move_up,move_down         : -> Act
..
assert
lift = cons(req_act,cons(open_act,cons(close_act,
           cons(req_up,cons(req_down,cons(move_up,cons(move_down,nil)))))))
```

Par exemple, l'action *open_act* est formalisée en utilisant la fonction `cond_mult_assg(plist, bexp)` dont la signature est `Set,Bexp -> Act`. Elle représente une affectation multiple conditionnelle, où `Set` est l'ensemble de paires `(Id,Exp)` et `Bexp` la condition :

```
assert
open_act
 = cond_mult_assg({open.T} \u ({id_to_arr(req)[floor].bool_to_bexp(false)}
                  \u ({move.T}\u {})),
                 (id_to_bexp(stop) ^ (nnot(id_to_bexp(open))
                  ^ (val(id_to_arr(req),floor))
                  ^ nnot(queueing)))))
```

Il est important de noter que le terme *req[floor]* de l'affectation *req[floor]:=false* et celui de la condition de l'action *open_ act* du programme *lift*, ne représentent pas des objets du même type. En effet, le premier désigne la *composante* du tableau *req* à l'indice *floor*, à laquelle on affecte la valeur *false*. Par contre, le deuxième terme *req[floor]* de la condition de l'action *open_ act* représente la *valeur* de la composante du tableau *req* à l'indice *floor*, soit en utilisant la notation précédente, cette dernière s'écrit *req(floor)* et non *req[floor]*. D'où les deux termes, `val(id_to_arr(req),floor)` et `id_to_arr(req)[floor]` dans la représentation de l'action *open_ act* en LP.

39. Les variables existentielles sont représentées par des constantes de Skolem, dû au fait que la version LP utilisée est 2.4. Une nouvelle formalisation est en cours, pour la version 3.1 qui permet l'utilisation des quantificateurs existentiels

## 8.3 Les obligations de preuves

Les obligations de preuves spécifient les propriétés que le programme $lift$ doit vérifier. Nous allons décrire la preuve de correction du protocole de l'ascenseur telle qu'elle a été proposée dans [And93], afin de montrer sa mécanisation dans notre système.

La propriété principale à vérifier consiste à montrer que s'il y a un appel à un étage $n$ alors tôt ou tard cet étage sera desservi, c'est à dire que $floor$, la valeur de l'étage courant, sera égal à $n$:

$$\forall n : min \leq n \leq max \Rightarrow (req(n) \ \textbf{leads\_to} \ open \wedge floor = n) \tag{8.1}$$

Comme en HOL-UNITY, une propriété *leads_to* doit être satisfaite par tout état "possible" du programme, c'est à dire l'espace d'états couvert par les variables d'états. Par conséquent, un prédicat *valid* est introduit, qui caractérise un sous ensemble de cet espace d'état incluant les états accessibles du programme $lift$:

$$\forall n : min \leq n \leq max \Rightarrow req(n) \wedge valid \ \textbf{leads\_to} \ open \wedge floor = n \tag{8.2}$$

Pour montrer cette propriété, il faut montrer que:

$$(moving \vee stopped \vee opened \vee closed) \wedge req(n) \wedge valid \ \textbf{leads\_to} \ opened \wedge floor = n \tag{8.3}$$

où
$moving = \neg stop \wedge \neg open$
$stopped = stop \wedge \neg open \wedge \neg move$
$opened = stop \wedge open \wedge move$
$closed = stop \wedge \neg open \wedge move$

Puis en utilisant la propriété de transitivité de l'opérateur *leads_to*, *TRANS_leads*, on montre la propriété ( 8.2) à partir de ( 8.3).

La propriété 8.3 est décomposée en des propriétés plus "simples", desquelles elle peut être déduite en utilisant les règles d'inférence de l'opérateur *leads_to*.

**Remarque 8.1** *Dans la suite, nous allons utiliser* $\mapsto$ *pour l'opérateur* leads_to, *et* $p$ **op** $q$ *pour* $\mathbf{op}(p, q, lift)$. *Nous utiliserons* $s$ *pour la variable stop,* $o$ *pour open,* $m$ *pour move,* $f$ *pour* $floor$, $q$ *pour queueing, a pour above, b pour below, goup pour goingup et godown pour goingdown.*

D'après la preuve originale, la décomposition de (2) produit les propriétés suivantes:

$$s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req(n) \wedge valid \mapsto s \wedge o \wedge m \wedge (f = n) \tag{$P_1$}$$
$$s \wedge \neg o \wedge \neg m \wedge (f \neq n) \wedge req(n) \wedge valid \mapsto s \wedge o \wedge m \wedge (f \neq n) \wedge req(n) \tag{$P_2$}$$
$$s \wedge o \wedge m \wedge (f \neq n) \wedge req(n) \wedge valid \mapsto s \wedge \neg o \wedge m \wedge \neg(f = n \wedge \neg q) \wedge req(n) \tag{$P_3$}$$
$$s \wedge o \wedge m \wedge (f = n \wedge \neg q) \wedge req(n) \wedge valid \mapsto s \wedge o \wedge m \wedge (f = n) \tag{$P_4$}$$
$$s \wedge \neg o \wedge m \wedge \neg(f = n \wedge \neg q) \wedge req(n) \wedge valid \mapsto \neg s \wedge \neg o \tag{$P_5$}$$
$$\neg s \wedge \neg o \wedge req(n) \wedge valid \mapsto s \wedge \neg o \wedge \neg m \wedge (f = n) \wedge req(n) \tag{$P_6$}$$

Pour chaque propriété $p \mapsto q$, il faut montrer $p$ *ensures* $q$, puis utiliser la règle *LEADS_to* (paragraphe 2.7). Pour illustrer le prochain paragraphe, nous allons décrire les étapes principales de la preuve de $P_2$, propriété "simple", qui exprime le fait que si l'ascenseur est arrêté à un

étage *floor* et qu'il y a un appel à cet autre étage mais où l'ouverture des portes est prioritaire sur le déplacement alors les portes s'ouvriront et le déplacement sera prioritaire sur l'ouverture des portes (pour éviter que l'ascenseur ne soit bloqué à l'étage courant).

**Preuve** de $P_2$:

$$s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid \mapsto s \wedge o \wedge m \wedge \neg(f = n) \wedge req(n) \qquad (P_2)$$

$\Leftarrow \{Définition\ de\ leads\_to\}\}$

$$s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid \textbf{ ensures } s \wedge o \wedge m \wedge \neg(f = n) \wedge req(n)$$

$\Leftarrow \{Définition\ de\ ensures\}$

$$[s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid \textbf{ unless } s \wedge o \wedge m \wedge \neg(f = n) \wedge req(n)]$$
$$\wedge [s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid \textbf{ exit\_act } s \wedge o \wedge m \wedge \neg(f = n) \wedge req(n)]$$

$\Leftarrow \{Définition\ de\ unless\}$

$$[\forall act \in lift\ ,\ (s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid) \qquad (P_{2.1})$$
$$\Rightarrow \mathbf{wp}(act, (s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid)$$
$$\vee (s \wedge o \wedge m \wedge \neg(f = n) \wedge req(n)))]$$
$$\wedge [s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid \textbf{ exit\_act } s \wedge o \wedge m \wedge \neg(f = n) \wedge req(n)] \qquad (P_{2.2})$$

**Preuve de** $(P_{2.1})$: Ce sous-but produit six sous-buts, un pour chaque action du programme. Nous allons nous intéresser à l'action *open_act*.

$$[(s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid)$$
$$\Rightarrow \mathbf{wp}(open\_act, (s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid)$$
$$\vee\ (s \wedge o \wedge m \wedge \neg(f = n) \wedge req(n)))]$$

$\{Définition\ de\ wp\ et\ des\ affectations\}$

$$(s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid)$$
$$\Rightarrow [((s \wedge \neg o \wedge req(f) \wedge \neg(m \wedge q)) \Rightarrow [(s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid)$$
$$\vee (s \wedge o \wedge m \wedge \neg(f = n) \wedge req(n))]_{open\_act}$$
$$\wedge(\neg(s \wedge \neg o \wedge req(f) \wedge \neg(m \wedge q)) \Rightarrow ((s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid)$$
$$\vee (s \wedge o \wedge m \wedge \neg(f = n) \wedge req(n))))]$$

$\{substitution\}$

$$(s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid)$$
$$\Rightarrow [((s \wedge \neg o \wedge req(f) \wedge \neg(m \wedge q)) \Rightarrow [(false)$$
$$\vee (s \wedge true \wedge true \wedge \neg(f = n) \wedge req(n))_{req[f]:=false}]$$
$$\wedge(\neg((s \wedge \neg o \wedge req(f) \wedge \neg(m \wedge q))) \Rightarrow ((s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid)$$
$$\vee (s \wedge o \wedge m \wedge \neg(f = n) \wedge req(n))))]$$

$\{simplification\}$

$$(s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid)$$
$$\Rightarrow [((s \wedge \neg o \wedge req(f) \wedge \neg(m \wedge q)) \Rightarrow [(s \wedge \neg(f = n) \wedge req(n))_{req[f]:=false}]$$

$\{substitution\ dans\ le\ tableau\}$
$(s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid)$
$\quad \Rightarrow [(s \wedge \neg o \wedge req(f) \wedge \neg(m \wedge q)) \Rightarrow [s \wedge \neg(f = n) \wedge (if\ \ n = f\ \ then\ \ false\ \ else\ \ req(n))]]$

$\{simplification\}$

$(s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid)$
$\quad \Rightarrow [(s \wedge \neg o \wedge req(f) \wedge \neg(m \wedge q)) \Rightarrow [s \wedge \neg(f = n) \wedge (((n = f) \wedge false) \vee (\neg(n = f) \wedge req(n)))]]$

$(s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid)$
$\quad \Rightarrow [(s \wedge \neg o \wedge req(f) \wedge \neg(m \wedge q)) \Rightarrow [(s \wedge \neg(f = n) \wedge (\neg(n = f) \wedge req(n)))]]$

$true$
□

**Preuve de $P_{2.2}$:**

$[s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid\ \mathbf{exit\_act}\ s \wedge o \wedge m \wedge \neg(f = n) \wedge req(n)]$

$\{Définition\ de\ exis\_act\}$

$[(s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid)$
$\quad \Rightarrow \mathbf{wp}(req\_act, (s \wedge o \wedge m \wedge \neg(f = n) \wedge req(n)))]$
$\vee[(s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid)$
$\quad \Rightarrow \mathbf{wp}(open\_act, (s \wedge o \wedge m \wedge \neg(f = n) \wedge req(n)))]$
$\vee \ldots$
$\vee[(s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid)$
$\quad \Rightarrow \mathbf{wp}(move\_down\_act, (s \wedge o \wedge m \wedge \neg(f = n) \wedge req(n)))]$

$\{seule\ \ open\_act\ \ vérifie\ \ la\ \ \ propriété\ \ wp\}$

$[false]$
$\vee[(s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid)$
$\quad \Rightarrow [((s \wedge \neg o \wedge req(f) \wedge \neg(m \wedge q)) \Rightarrow (s \wedge true \wedge true \wedge \neg(f = n) \wedge req(n)))$
$\qquad \wedge(\neg(s \wedge \neg o \wedge req(f) \wedge \neg(m \wedge q)) \Rightarrow (s \wedge o \wedge m \wedge \neg(f = n) \wedge req(n)))]]$
$\vee[false]$

$\{Simplifications\}$

$[false]$
$\vee[(s \wedge \neg o \wedge \neg m \wedge \neg(f = n) \wedge req(n) \wedge valid)$
$\quad \Rightarrow [((s \wedge \neg o \wedge req(f) \wedge \neg(m \wedge q)) \Rightarrow (s \wedge \neg(f = n) \wedge req(n)))$
$\qquad \wedge(\neg(s \wedge \neg o \wedge req(f) \wedge \neg(m \wedge q)) \Rightarrow (s \wedge o \wedge m \wedge \neg(f = n) \wedge req(n)))]]$
$\vee[false]$

$\{(\neg(f = n) \wedge req(n) \wedge \neg q) \equiv false\}$

$true$
□

Dans le paragraphe qui suit, nous illustrons la preuve mécanisée de la propriété $P_2$. En particulier, LP prouve de façon automatique les sous-buts de la preuve manuelle avec les règles de simplification des expressions booléennes et nous évite l'introduction des détails fastidieux pour manipuler les étapes "logiques" de la déduction dans la preuve d'un but particulier.

## 8.4 La preuve mécanisée

En LP, montrer que le programme *lift* vérifie *p unless q* revient à montrer que le terme `unless(p,q,lift)` se réduit au terme `true`.

Nous introduisons la conjecture comme suit:

```
prove unless(p,q,lift)
```

En utilisant la définition de `unless`, LP déplie la conjecture en:

```
unless(p,q,req_act) /\ unless(p, q,cons(open_act,cons(...,cons(move_down,nil))))
```

Chaque terme de la conjonction est réécrit en introduisant `wp`:

```
((p ^  nnot(q)) \=> wp(req_act,p \or  q))
 /\ ((p ^  nnot(q)) \=> wp(open_act,p \or  q))
  :
   /\ ((p ^  nnot(q)) \=> wp(move_down,p \or q))
```

D'après la preuve manuelle, nous devons subdiviser la conjecture en six sous-buts, un pour chaque action. Nous utilisons alors la commande `resume by /\-m`, qui permet à LP de montrer chaque sous-but indépendamment. LP génère les sous-buts de la forme

```
(p ^  nnot(q))=T \=> wp(act,p \or  q) = T
```

Pour toute action du programme *lift* qui ne modifie les variables apparaissant dans `p \or q`, le sous-but correspondant est résolu par LP car la propriété se ramène à une tautologie.

Pour les actions, qui modifient les variables de `p \or q`, telle que `open_act` pour la propriété $P_2$, nous procédons par cas, suivant que $floor = n$. Ce traitement par cas correspond à la substitution dans le tableau, c'est le "if" qui apparaît dans la preuve manuelle, sachant que la substitution se fait avant la simplification[40] (le sous-terme `nnot(open)` se réduit à `false` à cause de l'affectation `open_act`).

Le script de la preuve de la propriété ($P_{2.1}$), où nous montrons le sous_but *unless* avec l'action `open_act` (un programme avec comme seule action `open_act`), est le suivant:

```
set name propriete2
prove
  unless(stopped ^ nnot(id_to_exp(floor) \= id_to_exp(nn))
           ^ val(id_to_arr(req),nn),
         opened ^ nnot(id_to_exp(floor) \= id_to_exp(nn))
           ^ val(id_to_arr(req),nn),
         cons(open_act,nil))
  ..
```

40. Comme nous l'avons expliqué au chapitre 2 consacré à la formalisation (p. 60), cette propriété de la substitution est importante pour la correction.

```
  res by /\-m
    <> /\ subgoal
    [] /\ subgoal
    <> /\ subgoal
    [] /\ subgoal
    <> /\ subgoal
    res by /\-m
      <> /\ subgoal
      [] /\ subgoal
      <> /\ subgoal
      [] /\ subgoal
      <> /\ subgoal
      [] /\ subgoal
    [] /\ subgoal
    <> /\ subgoal
    res by case id_to_nat(floor) = id_to_nat(nn)
      <> case id_to_nat(floor) = id_to_nat(nnc)
      [] case id_to_nat(floor) = id_to_nat(nnc)
      <> case ~(id_to_nat(floor) = id_to_nat(nnc))
      [] case ~(id_to_nat(floor) = id_to_nat(nnc))
    [] /\ subgoal
    <> /\ subgoal
    res by case id_to_nat(floor) = id_to_nat(nn)
      <> case id_to_nat(floor) = id_to_nat(nnc)
      [] case id_to_nat(floor) = id_to_nat(nnc)
      <> case ~(id_to_nat(floor) = id_to_nat(nnc))
      [] case ~(id_to_nat(floor) = id_to_nat(nnc))
    [] /\ subgoal
    <> /\ subgoal
    [] /\ subgoal
    <> /\ subgoal
    [] /\ subgoal
    <> /\ subgoal
    [] /\ subgoal
  [] conjecture
qed
```

Les lignes

```
    <> /\ subgoal
    [] /\ subgoal
```

correspondent à la simplification des sous-buts qui sont des tautologies. Les deux preuves par cas correspondent à la substitution de $ref(floor)$ par $false$ dans le terme de la forme $p \vee q$. Enfin, le prédicat *valid* est prouvé comme étant un invariant dans notre système est n'apparaît pas dans les propriétés à prouver.

Les propriétés $P1$, $P3$, ... $P5$, sont prouvées suivant le même principe, puisqu'elles se ramènent à des propriétés `unless(p,q,lift)` et `exist_act(p,q,lift)`.

En résumé, une propriété `ensures(p,q,lift)` est réécrite en

```
unless(p, p, cons(req_act, lift)) /\ exist_Act(p, p, cons(req_act, lift))
nnot(p) \or wp(req_act,p \or q) \or q = T                    %unless
```

```
/\  nnot(p) \or wp(open_act,p \or q) \or q = T
...
/\  nnot(p) \or wp(move_down,p \or q) \or q = T
/\  nnot(p) \or wp(req_act,q) \or q = T                    %exist_act
      \/ nnot(p) \or wp(open_act,q) \or q = T
      ...
      \/ nnot(p) \or wp(move_down,q) \or q = T
```

où chaque sous-but est soit simplifié automatiquement à `true` par LP, soit résolu par cas. Pour faciliter la preuve de la propriété $P_5$, nous montrons une propriété auxiliaire:

```
(~ above /\ ~ below /\ (req(n)=T) /\((0 < n) /\ (n < 2))  => (floor = n)
```

qui exprime la propriété que "s'il n'y a aucun appel aux étages supérieurs à l'étage *floor* où se trouve actuellement l'ascenseur, ni aux étages inférieurs à *floor* et s'il y a un appel à l'étage $n$ alors $floor = n$".

Vu le nombre de détails à gérer pour chaque propriété et pour chaque sous-but de la forme `A \=> B`, où LP réécrit les termes `A` et `B` en utilisant la base des règles définissant les expressions de prédicats (cf. paragraphe 2.3), les preuves deviennent rapidement complexes à étudier. Nous avons alors utilisé la stratégie décrite au paragraphe 2.3.3, qui consiste à "bloquer" la réécriture de `\=>` en `\or`, en immunisant la règle correspondante et à utiliser la règle:

$R_{impl}$ : `(A \=> B = T) => (A = T => B = T) -> T`

Nous commençons par "essayer" la preuve de `(A = T) => (B = T)`, où `B` est de la forme `wp(act,R)`. Cette stratégie nous permet d'utiliser la méthode *implique* de LP. Deux cas sont alors possibles: Soit la preuve échoue, auquel cas en utilisant $R_{impl}$, nous en déduisons que `A \=> B = T` est faux, ou bien la preuve réussit, auquel cas, nous utilisons les informations obtenues pour prouver `A \=> B`. Ceci nous permet de contrôler la preuve mécanisée de chaque sous-but, en identifiant clairement les hypothèses et les sous-buts à prouver. Ainsi pour chaque sous-but, la commande `resume by =>-m` demande à LP d'utiliser la méthode d'implication. Il génère l'hypothèse `((p ^ nnot(q))=T)`, afin de montrer les sous-but `wp(act,p \or q)=T`.

**Remarque 8.2** *Nous pouvons définir une stratégie de preuve générale en utilisant la commande:*

```
set proof-methods normalization, /\, =>
```

*Cette commande fait que* LP *essaye de prouver une conjecture avec la première méthode applicable parmi celles données dans la liste. Par conséquent, pour montrer une propriété unless,* LP *utilise la méthode* `/\-method` *pour subdiviser la conjecture après l'avoir normalisée. Si un sous_ but de la forme* `A => B` *apparaît,* `Lp` *utilise la méthode* `=>-method` *sans l'aide de l'utilisateur. Comme les obligations de preuves de p unless q et invariant p sont les mêmes pour tout p et q, la preuve mécanisée peut être complètement automatisée avec une seule commande et la preuve correspondante serait de la forme:*

```
prove unless(p,q,pgm)
qed
```

*Dans ce cas, lorsque la preuve est suspendue, nous savons que* LP *a besoin d'un lemme supplémentaire ou que l'action en question ne vérifie pas la propriété.*
*Bien que cette stratégie nécessite moins d'interaction avec* LP*, nous ne l'utilisons pas afin de garder le contrôle sur les étapes de base de la preuve. En effet, comme notre objectif est d'abord de*

*vérifier les preuves et les propriétés de sûreté, la mécanisation de la preuve nous amène, comme c'est le cas généralement dans ce genre d'expérience, à détailler la spécification dans ses moindres détails. Dans la plupart des cas, les étapes de bas niveau contiennent les aspects les plus subtiles du problème. Néanmoins, ces étapes sont généralement acceptées sans preuves, sur la base d'arguments intuitifs. Comme ces arguments englobent des aspects critiques du problème, ils doivent être formalisés et vérifiés afin de mettre à jour les éventuelles erreurs ou incomplétude.*

## 8.5 Discussion

Nous avons décrit dans ce chapitre, la correction d'un protocole de contrôle d'un ascenseur. Nous avons utilisé cet exemple comme illustration de note approche pour la mécanisation de la correction à l'aide d'un démonstrateur tel que LP. Nous n'avons pas décrit la preuve mécanisée de toutes les propriétés pour les raisons suivantes: tout d'abord, les preuves de sûreté ont la même structure et utilisent le même principe de vérification. De plus, le nombre de détails à gérer dans les preuves manuelles fait que les preuves informelles sont longues et complexes. Enfin, nous avons utilisé cette exemple pour tester la faisabilité de notre approche et par conséquent les preuves mécanisées de certaines propriétés ne sont pas encore au point, notamment celle de la propriété $P6$, qui utilise une règle conditionnelle et la règle d'induction de la relation *leads_to*.

La formalisation des tableaux est assez lourde à gérer et peut être améliorée. En effet, cette représentation repose sur la simplification à un tableau à trois éléments, où ces composantes sont gérées indépendamment par des propriétés supplémentaires afin de s'assurer qu'il s'agit bien des éléments d'un même tableau. Par conséquent, un tableau d'une taille importante serait plus difficile à gérer.

Contrairement à d'autres méthodes, notre approche repose sur un raisonnement basé sur le texte du programme, qui nous semble plus intuitif et plus proche des problèmes que nous voulons résoudre. De plus, notre preuve mécanisée possède une structure aussi proche que possible de la preuve manuelle informelle que l'utilisateur peut faire, en ajoutant à la preuve la rigueur du démonstrateur qui devrait éliminer les erreurs de la spécification originale.

Le but de cette expérience était de montrer, à travers un exemple, la mécanisation des propriétés de sûreté avec LP et un traitement qui comprend des variables booléennes, entières et des types abstraits tels que les tableaux. L'exemple ayant été aussi traité à l'aide d'un démonstrateur d'ordre supérieur, à savoir HOL, notre but fut de montrer la possibilité de la mécanisation d'une telle preuve à l'aide d'un démonstrateur du premier ordre tel que LP.

Pour une première comparaison avec les preuves HOL, nous pouvons remarquer que nous n'avons pas fait allusion aux états ni à l'exécution. Chez Andersen, Petersen, et Pettersson [And93], les noms des variables, les constructions du programme et les connecteurs logiques opèrent sur des états et l'organisation de la preuve est différente. La preuve d'une propriété *doit être* montrée séparément pour chaque action, puis déduite des preuves plus simples, alors que notre preuve peut se faire pour tout le programme. Ils obtiennent des preuves élémentaires de petite taille mais leur nombre croit linéairement en fonction du nombre d'actions du programme. Nos preuves ont le comportement inverse, le nombre de commandes utilisées dans une preuve est linéaire par rapport au nombre d'actions du programme. Cette approche permet de traiter des exemples de tailles importantes, puisque que les propriétés de sûreté ont la même architecture, c'est à dire que le nombre de commandes utilisées dépend du nombre d'actions du programme considéré.

La tactique générale pour les propriétés de progrès *leads_to* dans les deux approches est similaire. Dans les deux formalisations, la preuve d'un *leads_to* nécessite la décomposition de la

propriété en des propriétés plus simples dans le but d'utiliser des règles d'inférence dérivées. Dans notre approche, cette décomposition est basée la plupart du temps sur des arguments intuitifs et utilise la règle d'élimination (*can_rule*) de la relation *leads_to* (Chapitre 6).

# Conclusion

Le travail décrit dans ce document s'inscrit dans le cadre général de la vérification formelle des programmes parallèles, et plus particulièrement, nous nous sommes intéressés à la vérification mécanique des programmes UNITY [Cha88]. L'objectif principal de notre travail fut d'une part, la formalisation de l'environnement de preuve proposé et d'autre part, la mécanisation des preuves de correction des programmes UNITY avec le démonstrateur de théorème du LARCH, LP.

## Bilan

Dans la première partie de cette thèse, nous avons décrit notre approche pour formaliser l'environnement de spécification des programmes UNITY. Une première phase fut la définition des objets de base utilisés par les programmes UNITY et plus généralement les données sur lesquelles porte le raisonnement. Cette phase est souvent sous estimée car elle est considérée comme une étape moins importante que la vérification. Or toute technique de vérification, aussi puissante soit elle, est cruellement dépendante de la spécification. En effet, l'expérience a montré qu'une grande majorité des erreurs provient d'une spécification mal écrite, vague ou contradictoire.

Certes l'utilisation des démonstrateurs automatiques, où l'introduction des données est sous le contrôle de l'outil, diminue la probabilité d'introduire des faits contradictoires. Néanmoins, ces outils sont généralement très spécifiques à un formalisme précis utilisé pour les applications d'un type particulier. Par conséquent, pour définir les objets nécessaires à une application, il faut passer par une étape intermédiaire de traduction. Un effort important est alors investi pour formaliser *toutes* les données dans le moule proposé, sans plus se préoccuper de savoir si ce qu'on formalise correspond bien aux besoins formulés au départ. Par contre, LP permet une écriture naturelle des définitions proche des spécifications algébriques. Cette liberté d'"expression" doit être utilisée avec rigueur afin de ne pas définir n'importe quoi. Nous avons illustré au chapitre 3, la spécification et la vérification d'une base de faits, fondée sur les spécifications LSL et nous avons proposé au chapitre 5, une méthodologie de définition et de vérification, fondée sur une méthode incrémentale pour créer et valider les bases qui seront utilisées pour les programmes UNITY, ou pour toute application que l'on désire vérifier. Enfin, la spécification et la vérification de la base de faits est réutilisable pour tout autre application car elle définit des objets usuels, tels que les entiers, les listes, les piles, les files,etc, et adapter ces définitions pour prendre en compte une application spécifique nécessite peu de modifications, car les stratégies de preuve sont identiques.

Nous avons choisi comme outil formel pour la preuve des programmes UNITY, le calcul de *la plus faible pré-condition* dû à Dijkstra [Dij89]. Cette modélisation (et sa mécanisation) est indépendante du modèle d'UNITY, et pourrait être appliquée (et bien sur, étendue) à un autre langage de programmation, avec une extension qui prendrait en compte d'autres constructions. Le wp_calcul nous a permis de raisonner sur la structure syntaxique du programme à prouver,

plutôt que sur les traces d'exécution, tel que le système NQTHM de Goldschlag dans [Gol90]. Notre approche se rapproche plutôt de celle de Brown et Mery décrite dans [Bro93].

Pour définir le calcul des prédicats sur lequel repose le modèle que nous avons choisi, nous avons appliqué la méthode incrémentale qui consiste à introduire une base restreinte d'axiomes acceptés par le sens commun, puis à enrichir cette base par des théorèmes qui sont prouvés à mesure qu'on les introduit. Nous nous sommes aussi basé sur des définitions usuelles qui ont fait leur preuves comme celles proposées par Dijkstra dans le cadre de la sémantique des langages de programmation.

Notre objectif était de montrer à travers UNITY et LP, l'un comme environnement de spécification et l'autre comme outil de vérification, certains aspects importants liés aux méthodes formelles, à savoir la nécessité de spécifier rigoureusement les objets sur lesquels porte le raisonnement et la fragilité des preuves informelles généralement basées sur des intuitions dont un outil de validation ne peut se satisfaire. Les études de cas présentées nous ont permis d'atteindre ces objectifs, et en particulier, nous avons mis en évidence le besoin des preuves mécanisées pour la vérification d'applications plus ou moins complexes.

## Les études de cas

La deuxième partie de cette thèse a été consacrée aux applications, dont la plus importante est la vérification d'un protocole de communication à travers des canaux défectueux. En effet, le raisonnement formel sur la correction des protocoles est nécessaire pour garantir aux protocoles un comportement correct par rapport à leurs spécifications, puisque le raisonnement informel n'est pas fiable et le test n'est pas complet. Néanmoins, ce raisonnement formel est une tâche complexe et sujette à des erreurs. Cette complexité est due au fait que les protocoles eux mêmes sont complexes et exhibent des comportements compliqués. L'utilisation d'un outil de preuve pour vérifier la correction des preuves est donc nécessaire pour le succès d'une telle tâche.

L'utilisation d'un démonstrateur nécessite la vérification des hypothèses ou suppositions basées sur des arguments intuitifs, et fournit donc un degré élevé de certification de la vérification. Par ailleurs, la semi-automatisation des preuves nous a permis d'éviter des détails fastidieux des preuves manuelles afin de se concentrer sur les aspects profonds du problème. Cette expérience nous a montré que même un protocole simple comprend une complexité inhérente à la plupart des protocoles de communication, due au nombre d'obligations de preuves et aux propriétés qui en découlent. En effet, nous avons vu que la complexité et le nombre de propriétés à vérifier font que les preuves manuelles, même détaillées, ne sont pas suffisantes pour s'assurer de la correction. Les preuves manuelles comprennent des hypothèses et des suppositions non vérifiées, basées sur une compréhension hâtive du problème ou à des simplifications hasardeuses. Par contre, l'utilisation d'un outil d'aide à la preuve nécessite la formalisation et la vérification de ces suppositions. Par conséquent, il est facile d'imaginer le nombre d'erreurs qui peuvent s'introduire dans la vérification de protocoles plus complexes, où la quantité d'information à manipuler incite à utiliser des hypothèses arbitraires, posées pour un gain de temps. L'impact de telles erreurs est alors difficiles à quantifier, notamment pour des protocoles mettant en jeu des vies humaines.

La vérification du problème classique des *lecteurs-rédacteurs* est une illustration de ce que peut être une preuve mécanique par rapport à une preuve informelle structurée. En effet, nous avons montré qu'il ne suffit pas d'écrire la structure générale d'une preuve pour la vérifier mécaniquement. Nous avons mis en évidence la nécessité de développer toutes les étapes d'une preuve informelle afin que la vérification puisse être menée de façon correcte. Ce développement nous a permis de mettre en évidence des propriétés nécessitant la modification des actions du programme. Par ailleurs, cet exemple illustre la réutilisation des preuves. En effet, bien que LP

ne fournisse pas de méta-langage pour définir des tactiques, nous pouvons utiliser ses techniques pour réutiliser une même structure de preuve. Ceci est possible grâce au fait que LP permet la construction d'une preuve formelle ayant une structure identique à celle de la preuve informelle. Ainsi, si deux preuves informelles admettent les mêmes techniques de preuves, nous pouvons alors utiliser un même *script* LP.

Dans le but d'une comparaison avec d'autres démonstrateurs traitant des preuves UNITY, l'exemple du programme spécifiant le contrôle d'un ascenseur a été une expérience concluante. La preuve originale ayant été effectuée avec le démonstrateur d'ordre supérieur HOL, il fut intéressant de vérifier cette même preuve avec un démonstrateur du premier ordre. Nous avons ainsi pu montrer que les structures des preuves étaient similaires. En particulier, les preuves de vivacité dans les deux formalisations se ramènent à des preuves de sûreté. Nous avons mis aussi en évidence le caractère semi-automatique des preuves de sûreté.

## Les avantages et les inconvénients de Lp

Cette expérience de formalisation d'un environnement de programmes parallèles nous a montré que **Lp** semble bien adapté à ce type de preuve. LP interagit beaucoup avec l'utilisateur et une preuve doit être conçue rigoureusement afin d'être automatisée. Mais cette exigence diminue la probabilité qu'elle soit erronée, et en cas d'erreur, comprendre la cause de cette erreur est crucial pour la conception. En effet, lorsqu'on vérifie un programme en cours de conception ou déjà existant, il nous semble plus important de comprendre le *pourquoi* de l'erreur et de modifier la conception en conséquence que d'essayer plusieurs méthodes pour forcer la preuve à réussir. Dans cet ordre d'idée, LP doit être vu comme un *assistant* de preuve et non pas seulement comme un *correcteur* de preuve car une utilisation méthodique permet non seulement la détection des erreurs mais surtout la mise au point des preuves. Et cet aspect a d'autant plus d'importance que la preuve est complexe.

L'une des causes d'erreur dans les preuves manuelles est que beaucoup de propriétés mineures sont acceptées implicitement sans être vérifiées rigoureusement. Or, lorsqu'on traduit la spécification pour être traitée par un outil de preuve mécanique, ces propriétés doivent être explicitées et il n'est pas rare que cette explicitation mette en lumière des erreurs ou des failles de raisonnement.

En comparaison avec d'autres systèmes plus automatisés, LP ne contient (pratiquement) aucune fonction prédéfinie et les définitions ne sont pas contraintes par une sémantique particulière. Ceci peut être un avantage car une liberté totale est laissée au concepteur de la preuve pour le choix de ses fonctions, la représentation des types des données et enfin la méthode à utiliser. En effet, il ne semble pas réaliste d'envisager un système de preuve qui requière de la part de l'utilisateur des connaissances approfondies sur les démonstrateurs utilisés, d'autant plus que ces démonstrateurs sont destinés à être utilisés dans des domaines assez variés. En contre partie, cette liberté est payée par le risque de définir des systèmes inconsistants. Néanmoins, à l'aide d'une méthodologie de spécification rigoureuse basée sur l'utilisation d'objets simples, ce danger peut être minimisé. De plus, nous attachons beaucoup plus d'importance à l'efficacité de LP et à la possibilité qu'il offre pour un raisonnement interactif pour la construction des preuves et leur mise au point.

Un des reproches émis contre LP est que l'interface est très simple et ne permet pas de définir des théories, c'est à dire qu'il n'y a pas de méta-langage. Or la philosophie à la base de la conception de LP est tout d'abord la simplicité. Le méta-langage risque de le compliquer. Par ailleurs, notre but est justement de montrer que malgré cette apparente simplicité, les méthodes proposées sont puissantes. De plus, cela rejoint notre choix de UNITY comme environnement de spécification qui, malgré une logique assez simple, est assez puissant pour traiter de nombreux exemples. De

plus, en ce qui concerne LP, l'avantage des spécifications axiomatiques réside dans sa portabilité et son utilisation par une communauté assez large. De plus, rien ne nous empêche d'appliquer ces méthodologies de preuves à un autre outil de vérification mécanique. En conclusion, ce document doit être vu à la fois comme un ensemble de preuves effectuées à l'aide de cet assistant de preuve et comme une illustration de comment utiliser les techniques proposées pour effectuer ces preuves ainsi que les difficultés et complexités que l'on peut rencontrer.

La contribution essentielle de ce travail a été de tirer avantage de la simplicité et de la puissance du modèle UNITY comme environnement de spécification et de celles de LP pour la vérification. Les systèmes parallèles sont par essence complexes, et on ne peut assurer leur fiabilité sans une spécification formelle et des preuves rigoureuses. Nous avons essayer d'atteindre ces deux buts avec UNITY pour le premier aspect et avec LP pour le second.

## Travaux connexes

Nous décrivons dans ce paragraphe, d'une manière non exhaustive, certains travaux comparables. Nous nous intéressons en particulier d'une part à la vérification mécanique des programmes parallèles avec LP et d'autre part, à la vérification mécanique des programmes UNITY à l'aide d'autres outils de validation, tels que le système NQTHM de Boyer & Moore, le démonstrateur d'ordre supérieur HOL, ainsi qu'un système de vérification utilisant le *modèle checking.*

### Vérification mécanique avec Lp

Le démonstrateur du LARCH, initialement conçu pour la vérification des spécification LSL [Gut93], a été utilisé dans différents domaines, preuve de circuits matériels [Gar88b, Sta89, Sco92, Sax92, All95], preuve de propriétés en utilisant l'approche basée sur les automates [SA93], preuve de programme ADA [Gua92],etc.

L'approche qui nous intéresse est l'utilisation de LP pour la vérification mécanique de circuit, décrit dans le langage des *transitions synchronisées* [Mel92]. Ce langage est proche de la philosophie des programmes UNITY, car il décrit un calcul comme une collection d'affectations conditionnelles atomiques sans contrôle de flux de données explicite, où la communication se fait à l'aide de variables partagées. Il permet la description d'un circuit par un système de transition, où chaque transition correspond à un sous circuit et est exécutée en parallèle, de façon répétitive et indépendamment des autres. La différence entre cette approche et UNITY réside dans les concepts structurels du langage ST, qui permettent une description hiérarchique du circuit, en regroupant les composants d'une partie d'un circuit (déclaration générique paramétrée d'un ensemble de variables d'états et de transitions), pour former des *cellules* aux quelles sont associées des propriétés, *invariants* et des *protocoles* (conditions sur les transitions).

La formalisation en LP est similaire à notre approche. Le programme écrit dans le langage des transitions synchronisées est traduit en deux parties, une partie indépendante qui comprend les définitions générales telles que les entiers, prédicats, etc et une partie qui dépend du programme à vérifier. La première partie correspond à notre formalisation des entiers, des listes, et des expressions booléennes (chapitre 2 et 3). Ce sont des spécifications utilisées pour tous les programmes UNITY, et plus généralement pour toute application utilisant les entiers, les listes, etc.

Les variables d'états sont représentées par trois niveaux: le niveau *identificateur*, le niveau *variable*, et enfin le niveau *valeur*. Pour chaque type T de variables d'état, trois sortes sont ainsi

définies, `TId`, `Tvar` et `Tval`. Le premier niveau est représenté par une constante de la sorte `Id`, puis les deux autres niveaux sont obtenus par l'application des opérateurs ".":

```
. :  instance, Tid -> Tvar
. :  Tvar,   state -> Tval
```

En comparaison avec notre formalisation, le premier niveau est similaire. Nous avons utilisé la sorte `Id` pour représenter les "noms" des variables de programmes. Le deuxième niveau représente le fait qu'un identificateur est une variable différente suivant l'instance de la cellule. Ce niveau n'existe pas dans notre formalisation, puisqu'un programme UNITY ne peut contenir de procédure. En effet, une cellule peut être considérée comme une procédure avec des variables locales et globales. Le troisième niveau est représenté dans notre approche par les fonctions de conversion `id_to_S`, qui "associent" à un identificateur, un élément de la sorte `S` (Chapitre 2, paragraphe 2.4.1). Par exemple, `id_to_nat(x)` exprime le fait que `x` est un identificateur d'une variable entière. Les valeurs des variables sont représentées dans notre approche par les égalités syntaxiques de la forme `id_to_nat(x) = s(0)`, puisque nous n'avons pas formalisé la notion d'état. Ceci nous permet d'utiliser une idée intuitive de la valeur d'une variable, comme étant égale à une certaine quantité.

L'utilisation de la notion d'état implique que la notion d'invariant est représentée par une fonction dont l'un des paramètres est l'état où on désire vérifier l'invariant. Ceci implique qu'un invariant possède une expression différente suivant l'état où il doit être vérifié. Dans notre approche, la notion d'invariant est une fonction dont les arguments sont le prédicat "invariant", le prédicat représentant les conditions initiales et le programme considéré, et par conséquent l'invariant est vérifié pour tous les états du programme.

Pour la vérification formelle, les transitions sont représentées par des prédicats. Ces transitions, similaires aux affectations UNITY, sont formalisées par rapport à deux états particuliers, `pre` et `post`, représentant l'état précédant l'exécution de la transition et l'état résultant. Par exemple, la transition $Expl : \langle t < N \rightarrow t := 2 * t \rangle$ est représentée par le prédicat $(t.pre < N)\&(t.post = 2 * t.pre)$. Ceci nécessite la définition de prédicats sur les états `pre` et `post` de la forme, "si un invariant est vrai dans l'état `pre` et une transition a été exécutée (relation entre `pre` et `post`) alors l'invariant est valide dans l'état `post`". Pour chaque transition, deux prédicats et une règle sont définis:

```
assert Cond(...) = ...

assert affect(...) = ...

when transition(...)
yield Cond(...)          %preconditions
      Affect(...)        %action
```

Les transitions ont donc plusieurs instances suivant la cellule et l'état considérés. Ceci risque de compliquer les obligations de preuves mais l'avantage est la vérification localisée [Sta89], qui consiste à montrer l'invariance du circuit en montrant que chaque cellule vérifie son invariant local et pour chaque instanciation des cellules, la non-interférence est prouvée.

Une deuxième approche intéressante est la vérification mécanique des programmes spécifiés en TLA [Lam94], à l'aide du système TLP, développé par Urban Engberg [Eng95]. TLA est un sous ensemble de logique temporelle qui permet la spécification d'invariance et de progrès. Les programmes et leurs propriétés sont spécifiés dans une même logique et sont représentés par des formules. La formule TLA, $\Pi \Rightarrow \Phi$ exprime le fait que le système représenté par la formule $\Pi$ satisfait la spécification $\Phi$. Les instructions sont des expressions reliant les nouvelles valeurs des

variables aux anciennes valeurs $x\prime = x + 1$. Comme les instructions sont des formules logiques, il n'a pas de traduction d'une notation de programmation vers une notation logique.
L'approche intéressante de Urban Engberg consiste à utiliser deux traductions différentes d'une même spécification TLA, afin de simplifier le raisonnement. Il sépare le raisonnement sur les *actions* du raisonnement *temporel*. Le premier comprend seulement des constantes, des variables, des prédicats et des actions, et met en oeuvre les techniques de réécriture de LP et les règles prédéfinies pour la manipulation des booléens. Le raisonnement *temporel* comprend les opérateurs "logiques" de TLA ainsi que les règles de manipulations appropriées.

Dans les deux codages, les variables *flexibles* (variables de programme) sont déclarées comme des constantes LP, et les variables *rigides* sont déclarées comme des variables LP. Deux sortes sont utilisées pour le raisonnement prédicatif, la sorte prédéfinie de LP, `Bool` et une sorte `Val`. Pour le raisonnement temporel, la seule sorte utilisée est `any`. Les opérateurs logiques de TLA sont des opérateurs LP de signature `any,any -> any`, "prime" et □ sont définis comme des opérateurs LP de signature `any -> any`.

En comparaison avec notre approche, la même philosophie est utilisée pour représenter les variables de programme et les variables de preuve. La différence réside dans l'utilisation de deux sortes uniques pour les valeurs des variables où la sorte `val` représente tout autre type que les booléens. De plus, les valeurs possibles des variables pour un "programme" doivent être spécifiées explicitement. Enfin, il semble que les propriétés sur les entiers soient introduites et non vérifiées.

Cependant, les exemples traités par Urban Engberg à l'aide de TLP, tels qu'un *multiplieur 64 bits*, montre la faisabilité de son approche mais surtout la clarté des preuves effectuées. La traduction d'une spécification TLA en deux scripts LP est prise en charge par un traducteur ce qui décharge l'utilisateur de cette tâche fastidieuse. Bien que les exemples traités n'aient pas mis en évidence des erreurs de conception et que l'auteur n'aborde pas cet aspect de la vérification, cette approche nous semble bien adaptée à la détection des erreurs parce que les preuves sont bien structurées et lisibles, ce qui permet à l'aide de LP une mise au point importante.

L'une des applications les plus significatives de TLP est la spécification et la vérification d'un algorithme de récupération de mémoire (*Garbage Collector*) concurrent par Doligez [Dol95].

## Vérification mécanique des programmes UNITY

Nous nous intéressons dans ce paragraphe à trois exemples de vérification mécanique des programmes UNITY. Les deux premiers utilisent un démonstrateur de théorèmes, mais ont des approches différentes, l'une axiomatique comme celle que nous avons utilisée, l'autre opérationnelle, basée sur les séquences d'exécution. Le dernier exemple utilise comme méthode de preuve celle du *modèle checking*.

### Hol-Unity

Le but de la mécanisation d'UNITY en HOL [Gor93], développée par Flemming Andersen [And92], est de fournir un modèle théorique à UNITY [And92], afin de montrer sa consistance et sa complétude. Il définit une théorie générale des programmes parallèles comme extension conservative de la logique sous jacente à HOL. UNITY est alors un cas particulier de cette théorie générale. L'aspect intéressant de ce travail réside dans le codage de l'opérateur *leads_to*. En effet, cette relation est directement définie comme le plus fort prédicat vérifiant les trois règles d'inférence. En utilisant une théorie de point fixe, cet opérateur est défini comme le point fixe (ou la clôture) de la relation transitive et disjonctive induite par *ensures*.

HOL-UNITY est définie comme une collection de théories construites en amont de la théorie PPLAMB (Polymorphic Predicate Lambda Calculus) . Celle ci constitue la logique de HOL, définie dans le langage ML par un ensemble initial de constantes logiques et d'axiomes (termes prédéfinis HOL) et de règles d'inférence (fonction pré-définies ML). La première théorie, définie par Andersen, est celle des prédicats où les opérateurs logiques sont *promus* à des opérateurs sur des prédicats. Par exemple, la conjonction est définie comme suit:

$$p \wedge *q = \lambda s : *.ps \wedge qs$$

où $*$ représente le type polymorphe des états. En comparaison avec notre système, cela revient à la définition des opérateurs ^, \or, etc. et des règles qui nous permettent de passer des opérateurs définis à ceux prédéfinis de LP.

Vu que HOL-UNITY est implémenté comme une théorie HOL, raisonner avec HOL-UNITY revient à raisonner en HOL avec les définitions et théorèmes d'UNITY comme étant des théorèmes dans la logique de HOL. Nous pouvons faire la même remarque à propos de notre système vu que toutes les définitions et théorèmes UNITY forment un système de réécriture en LP.

Le modèle sémantique utilisé est celui d'un système de transitions, où un programme est représenté comme une liste de transitions. Une transition représente une affectation UNITY formalisée par une lambda expression typée. Un état est représenté par un type polymorphe *state*. Par exemple, l'affectation $v := exp$ est représentée par l'expression $\lambda\sigma : State.(\lambda w.(v = w) \rightarrow [exp]_\sigma | [w]_\sigma$, où $[exp]_\sigma$ est la valeur de l'expression dans l'état $\sigma$, et $[w]_\sigma = (\sigma w)$. Une transition est donc une fonction qui à un état $s$ associe un état $s\prime$. Un état est une fonction qui à une variable $x$ de type $Vtype$ associe une valeur de type $Valtype$.

Les propriétés dans HOL-UNITY sont définies comme des relations (modales) entre des ensembles d'états exprimés par des prédicats (d'états). Les prédicats sont des fonctions qui associent à un état une valeur booléenne, et l'espace d'état est défini par l'ensemble des variables déclarées dans le programme. Par conséquent, les propriétés de sûreté sont prouvées par rapport à tout état possible du programme.

Un des avantages de cette formalisation est la possibilité d'utiliser un type polymorphe pour représenter le type *état*, puisque les types des variables utilisées dans les programme ne sont pas connus à l'avance. Dans notre approche, nous avons définis les types les plus utilisés, tels que les entiers, les booléens, ainsi que les types abstrait de données, listes, files, etc. Enfin, à chaque spécification UNITY est associée une nouvelle théorie dans HOL, dont le nom doit être unique.

Les preuves mécaniques de sûreté sont prouvées pour toute action du programme, et par conséquent, ces preuves possèdent une structure similaire à celle que nous avons utilisée. Les différences, mis à part l'ordre supérieur, résident dans la possibilité fournie par HOL de définir des tactiques. Comme les preuves de sûreté utilisent un même principe, on peut définir des tactiques utilisables pour toute preuve de sûreté. Par contre, à l'inverse de LP où la réécriture constitue le moteur d'inférence du démonstrateur, la technique de réécriture doit être explicitement invoquée par l'utilisateur.

L'exemple du problème des lecteurs-rédacteurs (cf. chapitre 7) nous permet de donner quelques éléments de comparaison. Les noms des variables du programmes sont définis dans HOL-UNITY en utilisant un type énuméré ayant quatre valeurs, `nr,nw,q,b`, puis il montre l'existence d'une fonction qui associe à chaque valeur du type défini, un entier distinct, afin d'assurer que les variables sont distinctes. Dans notre approche, ce traitement n'est pas effectué pour chaque programme UNITY considéré, mais pour tous les programmes. En particulier, les variables des programmes sont définies comme des identificateurs de la sorte `Id`, sur laquelle nous avons défini un schéma d'induction qui nous assure que les deux identificateurs sont distincts (Chapitre 2, pa-

ragraphe 2.4.2). Pour traiter un programme particulier, il nous suffit d'associer un nom à chaque variable, sachant que les noms sont de la forme `nextid`$^n$`(firstId)`, avec $n \geq 0$.

Pour vérifier les preuves informelles des propriétés de sûreté données par Chandy et Misra, les lemmes utilisés sur les entiers naturels sont identiques aux lemmes dont nous avons eu besoin dans nos preuves de sûreté, ceci confirme la similarité de nos preuves avec celles effectués avec HOL. De même, les propriétés de vivacité sont décomposées dans les deux approches en propriétés de sûreté. Par contre, la preuve de la propriété (13) semble plus compliquée en HOL. En effet, l'interaction que nous avons utilisée dans la preuve de cette propriété consiste à ralentir la normalisation automatique des expressions booléennes effectuée par LP. En HOL-UNITY, les tactiques de réécriture des termes comprenant les opérateurs `/\*`, `\/ *`, etc., doivent être explicitement invoquées ainsi que la simplification des termes de la forme `p /\* q /\* p` en `p /\* q`. Dans notre système, ceci est pris en charge par le processus de réécriture à l'aide des règles définissant les opérateurs `^`, `\or`, etc.
En ce qui concerne la détection des erreurs, nous n'avons pas des éléments de comparaison. En effet, les propriétés problématiques ont été supposées comme étant valides et par conséquent, leurs preuves, les solutions choisies ainsi que la réaction du système face à ce type de blocage, n'ont pas été explicités dans le document.

En conclusion, la priorité d'Andersen n'était pas la mécanisation des preuves de programmes UNITY, mais plutôt la construction d'un modèle théorique pour UNITY afin de montrer la consistance et la complétude de la théorie, but qu'il a par ailleurs atteint.

### Le démonstrateur de Boyer & Moore et UNITY

David Goldschlag a utilisé le système de Boyer & Moore et un modèle opérationnel pour les programmes parallèles afin d'implémenter un système de vérification automatique pour UNITY [Gol90, Gol92]. Le but de ce travail est l'approche inverse de celle de Flemming Andersen, à savoir obtenir un système automatique de vérification, sans pour cela prouver tous les théorèmes et les corollaires de la logique d'UNITY.

Le modèle sémantique d'état-transition choisi est opérationnel, basé sur les traces d'exécution. Par conséquent, les propriétés à montrer sont des relations entre traces. Cette approche rend difficile la preuve des propriétés de la logique. À la différence de Unity en HOL, où les états et les programmes sont définis dans la logique de HOL, c'est à dire les types et les variables sont définis comme des objets de HOL, les prédicats et les programmes sont codés dans une syntaxe donnée. La différence fondamentale, avec notre approche (et celle d'Andersen) est la restriction aux états accessibles, puisque les états d'une séquence de traces sont obtenus par l'exécution d'une instruction du programme à partir de l'état initial du programme.

Un programme est une liste d'instructions et chaque instruction est une relation qui, à un état (précédent) associe un état (successeur). `(N old new E)` est vrai si `new` est un successeur possible de `old` avec la transition spécifiée par l'instruction `E` et où `N` dépend du programme traité. Un état peut être une structure de donnée. Une instruction est une liste dont le premier argument est un nom de fonction représentant une instruction générique. Une exécution d'un programme est un enchevêtrement des instructions du programme.

L'originalité de ce travail réside dans la définition d'une fonction `choose` (*scheduler*) qui choisit une instruction parmi la liste représentant le programme, à l'aide d'une fonction `next` qui retourne la prochaine instruction à exécuter (Si i est l'index de l'instruction courante, alors $(nexti) \geq i$). `Choose` et `next` implémentent un algorithme de type *Round Robin*. En plus de l'hypothèse d'équité

inconditionnelle induite par UNITY, des contraintes d'équité *faible*[41] et *forte*[42] sont définies.

Les prédicats temporels sont définis à l'aide d'un fonction `eval` qui permet d'évaluer une formule (un prédicat) dans un état. La fonction `eval` est définie en fonction de l'interpréteur `eval$` de NQTHM. L'avantage de cette approche est la possibilité de définir l'opérateur *leads_to* à l'aide de sa définition opérationnelle, à savoir si le prédicat `p` est vrai dans un état $i$, alors le prédicat $q$ doit être vrai dans un état $j$ tel que $j \geq i$. Les propriétés *unless*, *ensures* et *leads_to* sont donc prouvées par rapport aux états accessibles du programme. Les règles dérivées d'UNITY qui forment le système de preuve sont définies comme des théorèmes qui apparemment ne sont pas prouvés en tant que tels.

Enfin, le système développé possède un degré d'automatisme élevé du à la nature du démonstrateur NQTHM, qui utilise un certain nombres d'heuristiques. Néanmoins, nous ne pouvons juger comment réagit le système développé en cas de preuves incorrectes, vu que les exemples présentés sont utilisés pour illustrer la vérification automatique et non la détection des erreurs. Le but de l'auteur étant la vérification automatique de programmes plus ou moins complexes et non la mise au point des preuves dans le cas où une erreur est détectée, le problème n'est pas discuté dans le document. Cependant, nos conversations avec des spécialistes de l'utilisation de NQTHM montrent que la détection de l'origine de l'erreur en cas d'échec d'une preuve n'est pas facile.

### Model Cheking pour Unity, le système UV

Nous terminons cette brève description des travaux connexes par la vérification mécanique des programmes UNITY, utilisant la technique d'évaluation de modèle.

Le système UV [Kal94] implémente des algorithmes d'*évaluation de modèles* pour la vérification des programmes UNITY à états finis pour lesquels des propriétés UNITY propositionnelles sont vérifiées. Cette approche, bien qu'utilisant une méthode de vérification plus automatique que la technique déductive, s'intéresse à la détection des erreurs des preuves informelles (manuelles).

L'espace d'état du programme est défini comme étant le produit cartésien de toutes les variables déclarées, dont les types possibles sont les booléens, les entiers à domaines finis et les types énumérés. Les affectations gardées définissent la relation de transition du programme. La satisfaction d'une propriété est définie par rapport à l'ensemble des états accessibles par le programme.

L'aspect intéressant de cette approche réside dans le calcul du plus fort invariant du programme (qui caractérise l'ensemble des états accessibles à partir des états initiaux, par une exécution du programme) et l'utilisateur a le choix entre trois invariants par rapport auxquels vérifier sa propriété: le plus petit (construit avec les variables du programme qui doivent prendre leurs valeurs parmi leurs types respectifs), le plus grand ( à condition que le système réussisse à le calculer), et l'invariant courant (la conjonction de tous les invariants que le système a pu calculer). Un autre aspect intéressant réside dans les informations données à l'utilisateur. Pour chaque propriété, le système donne à tout moment son état: prouvée avec succès, non encore vérifiée, échec avec la possibilité de succès par rapport au plus fort invariant, prouvée non valide. Pour aider l'utilisateur dans ses preuves qui échouent, le système lui fournit des contres exemples. Pour les propriétés de sûreté qui sont des implications, il donne les états précédents et suivants, et les transitions qui ont fait échoué la condition de l'implication.

41. L'équité faible exclu l'existence d'exécution où une instruction est toujours activable mais jamais choisie. Pour garantir qu'une instruction sera effective sous une telle condition d'équité, il faut s'assurer que dès qu'elle est activable, elle le reste au moins jusqu'a ce qu'elle soit choisie.

42. Si une instruction est infiniment souvent activable, alors elle est infiniment souvent choisie.

Pour les propriétés du *leads_to*, les informations fournies sont le nombre d'itérations de point fixe effectués et un état duquel une exécution équitable mais non réussie a été calculée. Les variables de preuves sont simplement ajoutées dans le texte du programme Unity à vérifier. L'utilisateur donne le programme et les propriétés, et par conséquent les variables apparaissant dans les propriétés sont déclarées dans le texte du programme. L'inconvénient est que la modification des propriétés à vérifier entraine la modification du programme et par conséquent le calcul des invariants doit être modifié.

# Conclusion

Les méthodes formelles de spécification et de vérification ont été proposées pour réaliser l'impossible, à savoir obtenir des programmes et des systèmes fiables prouvés corrects. Néanmoins, jusqu'ici ces méthodes ont été considérées difficiles et coûteuses en pratique. De plus, les exemples traités sont généralement des exemples "d'école". Lorsque des exemples plus complexes ont été traités, la complexité était induite par leur structure logique et non par leur taille. Par conséquent, certains doutes ont été formulés quant à l'utilisation des méthodes formelles en pratique et sur leur généralisation. Néanmoins, depuis un certains nombres d'années, quelques expériences réussies ont été menées dans la vérification formelle des circuits matériels, dont on peut citer la plus connue, la vérification d'un microprocesseur par Hunt [Hun86]. Cette réussite dans le matériel a encouragé les communautés concernées à explorer la possibilité d'appliquer les méthodes formelles aux logiciels, dont un exemple probant est la vérification d'un algorithme de récupération de mémoire (*Garbage Collector*) concurrent par Doligez et Gonthier [Dol94].

**Importance de la spécification formelle** Due à la structure complexe des logiciels, même les méthodes de vérification les plus avancées ne peuvent résoudre le problème de la fiabilité de logiciel aussi facilement. La fiabilité d'un programme *correct* (vérifié formellement) dépend de façon cruciale de la correction de sa spécification formelle. En effet, la vérification n'a de sens que si les besoins sont formulés correctement. Dans une application pratique, cerner les besoins afin de fournir une spécification formelle, à partir des spécifications informelles souvent vagues et contradictoires, est une phase cruciale du développement. De plus, il s'avère souvent nécessaire de modifier et de raffiner la spécification initiale pendant le processus de développement, pour prendre en compte de nouveaux faits et pour avoir des solutions plus efficaces et élégantes. L'ingénierie de la spécification est en fait le domaine d'application le plus important des méthodes formelles.

**Importance de la vérification mécanisée** Plusieurs techniques de vérification, aussi nombreuses que diverses et plus ou moins automatiques ont été proposées et utilisées. Le point commun entre toutes ces méthodes est la conviction que les preuves manuelles et informelles ne sont pas fiables. Ceci est dû d'une part, à la nature complexe de la vérification et d'autre part, au fait qu'une preuve "humaine" comporte un grand nombre de "raccourcis", le plus souvent inconscients, dont un automate aveugle ne pourrait s'accommoder.

La différence entre ces différentes méthodes, outre le degré d'automatisme, est l'intérêt porté à la correction des erreurs détectées. En effet, certains systèmes sont conçus pour répondre par un "oui" ou par un "non", d'autres, généralement moins automatiques, proposent des techniques de mise au point des preuves.

Il nous semble évident qu'aucune méthode de vérification ne peut être jugée intrinsèquement. Nous pensons que l'avenir des méthodes formelles réside dans les systèmes généralistes qui com-

En conclusion, nos perspectives résident dans le transfert des méthodes formelles vers des applications pratiques et réelles, en particulier pour la sûreté et la sécurité de fonctionnement. En effet, l'apport essentiel de ce travail fut de comprendre l'importance d'une vérification assistée par un outil de vérification ou de validation et de montrer sa faisablité, tout en estimant la difficulté d'utilisation des méthodes formelles. Par ailleurs, il ne faut pas sous-estimer les difficultés du transfert de compétence. En effet, les industriels sont difficilement convaincus par les méthodes formelles, jugées difficiles à utiliser et à appliquer. La difficulté réside-t-elle dans les méthodes utilisées ou bien dans la difficulté des chercheurs à vulgariser leurs méthodes? Il est certain qu'un effort supplémentaire doit être accompli afin que les méthodes formelles (en particulier les spécifications) soient considérées comme un moyen nécessaire de communication entre le milieu académique et le milieu industriel et que le savoir-faire des chercheurs diffusent dans les entreprises.

Dans le futur, nous espérons traiter des exemples complets à la fois comme archétypes, mais aussi pour leur intérêt propre pour le milieu industriel. Actuellement, le concepteur de systèmes est très peu aidé dans sa tâche par des outils de validation et de vérification. Ce manque d'outils est dû à notre avis, plutôt à un retard de transfert de résultats fondamentaux existants qu'à l'absence de ces derniers.

# Bibliographie


[All95] M. Allemand. *Modélisation fonctionnelle et preuves de circuits avec LP.* Thèse de Doctorat, Université de Provence, 1995.

[And92] F. Andersen. *A theorem Prover for UNITY in Higher Order Logic.* Thèse de Doctorat, Technical University of Denmark, 1992.

[And93] F. Andersen, K.D. Petersen, et J.S. Pettersson. Program verification using HOL-Unity. J.J. Joyce et C.H. Seger, éditeurs, *Proceedings sixth International Workshop on Higher Order Logic theorem proving and its applications*, volume 780, série *Lecture Notes in Computer Science*, pages 1–15, Vancouver, Canada, août 1993. Springer-Verlag.

[Arn89] A. Arnold. Mec : A system for constructing and analising transition systems. J. Sifakis, éditeur, *Proceedings of a Workshop on Automatic Verification Methods for Finite State Systems, Grenoble (France)*, volume 407, série *Lecture Notes in Computer Science*, pages 117–150. Springer-Verlag, juin 1989.

[BC86] A. Ben Cherifa et P. Lescanne. An actual implementation of a procedure that mechanically proves termination of rewriting systems based on inequalities between polynomial interpretations. J. Siekmann, éditeur, *Proceedings 8th International Conference on Automated Deduction, Oxford (UK)*, pages 42–51. Springer-Verlag, 1986.

[BC87] A. Ben Cherifa et P. Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, 9(2):137–160, octobre 1987.

[Bor93] A. Borjesson, K. M. Larsen, et A. Skou. Generality in design and compositional verification using TAV. *IFIP Transactions C-10: Formal Descriptions Techniques*, pages 449–464, 1993.

[Bou92] R. H. Bourdeau et B. H. C. Cheng. An object-oriented toolkit for constructing specification editors. *COMPSAC'92: Computer Software and Applications Conference*, September 1992.

[Boy79] R. S. Boyer et J. S. Moore. *"A Computational Logic"*. ACM monograph series. Academic Press, Inc, 1979.

[Boy88] R. S. Boyer et J. S. Moore. *A Computational Logic Handbook.* Academic Press inc., Boston, 1988.

[Bro93] N. Brown et D. Mery. A proof environment for concurrent programs. *In Proceedings FME'93 Symposium*, volume 670, série *Lecture Notes in Computer Science.* Springer-Verlag, 1993.

[Cha88] K. M. Chandy et J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988. ISBN 0-201-05866-9.

[Che92a] B. Chetali. Preuve automatisée de circuits digitaux: Utilisation de LP pour prouver l'algorithme de division sans restauration et son implémentation avec l'UAL CATHEDRAL-II. Rapport de DEA, Université Henri Poincaré – Nancy 1, 1992.

[Che92b] B. Chetali et P. Lescanne. An exercise in LP: The proof of the non restoring division circuit. U. Martin et J.M. Wing, éditeurs, *Proceedings First International Workshop on Larch*, volume 780, série *Workshops in Computing*, pages 55–68, Dedham, Boston, août 1992. Springer-Verlag.

[Che95a] B. Chetali. Formal verification of concurrent programs using the Larch prover. Larsen K.G Engberg, U.H. et A. Skou, éditeurs, *Proceedings of the Workshop on Tools and Algorithms for the construction and analysis of Systems*, BRICS Notes, pages 174–186, Aarhus, Denmark, mai 1995.

[Che95b] B. Chetali et P. Lescanne. Formal verification of a protocol for communications over faulty channels. *Proc. of the 8th International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols (FORTE'95)*, Montréal, Quebec, Canada, octobre 1995. IFIP.

[Cla86] E. M. Clarke, E. A. Emerson, et A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specification. *ACM Transactions on Programming Languages and Systems*, 8(2), avril 1986.

[Coo78] A. S. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1), février 1978.

[Coq95] Projet Coq. The Coq proof assistant (version 5.10). Reference Manual, 1995.

[Cor92] D. Cormer. *TCP/IP: architecture, protocoles, applications.* Informatique intelligence artificielle. InterEditions, 1992.

[Cré95] P. Crégut et Heyd B. Coq-Unity. *Actes du GDR Programmation*, 1995.

[Cyr95] D. A. Cyrluk et M. K. Srivas. Theorem proving: Not an esoteric diversion, but the unifying framework for industrial verification. *Proceedings of the IEEE International Conference on Computer Design (ICCD)'95*, pages 538–544, Austin, Texas, octobre 1995. To appear.

[Der92] N. Dershowitz. Rpo for ac-termination. M. Rusinowitch et Jean-Luc Rémi, éditeurs, *Proceedings 3rd International Workshop on Conditional Term Rewriting Systems, Pont-à-Mousson (France)*, pages 96–100, juillet 1992.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Inc., 1976.

[Dij89] E. W. Dijkstra et C. S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1989.

[Dol94] D. Doligez et Gonthier G. Portable, unobtrusive garbage collection for multiprocessor systems. *POPL'94.* ACM, 1994.

[Dol95] D. Doligez. *Conception, realisation et certification d'un glaneur de cellules concurrent.* Thèse de Doctorat, Université Paris 7, 1995.

[Eng95] U. H. Engberg. *Reasoning in the Temporal Logic of Actions.* Thèse de Doctorat, Aarhus University, Denmark, 1995.

[Flo67] R. W. Floyd. Assigning meanings to programs. *Proceedings of the American Mathematical Society's Symposia in Applied Mathematics*, volume 19, pages 19–31, 1967.

[Gar88a] S. J. Garland et J. V. Guttag. Inductive methods for reasoning about abstract data types. *Proceedings of the 15th Symposium on Principles of Programming Languages*, pages 219–228, San Diego (USA), 1988. ACM.

[Gar88b] S. J. Garland, John V. Guttag, et J. Staunstrup. Verification of VLSI circuits using LP. *Proceedings of the IFIP WG 10.2 "The Fusion of Hardware Design and Verification".* Elsevier Science Publishers B. V. (North-Holland), 1988.

[Gar89] S. J. Garland et J. V. Guttag. An overview of LP, the Larch Prover. N. Dershowitz, éditeur, *Proceedings 3rd Conference on Rewriting Techniques and Applications, Chapel Hill (N.C., USA)*, volume 355, série *Lecture Notes in Computer Science*, pages 137–151. Springer-Verlag, avril 1989.

[Gar90] S. J. Garland et J. V. Guttag. Using LP to debug specifications. *Proceedings of the IFIP TC2/WG2.2/WG2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee (Israel).* Elsevier Science Publishers B. V. (North-Holland), avril 1990.

[Gar91] S. V. Garland et J. V. Guttag. A guide to LP, the Larch prover. Rapport no. 82, Digital Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, USA., 1991.

[Ger89] R. Gerth et A. Pnueli. Rooting UNITY. *Proceedings of the Fifth International Workshop on Software specification and Design*, 1989.

[Gol90] D. M. Goldschlag. Mechanically verifying concurrent programs with the Boyer-Moore Prover. *IEEE Transactions on Software Engineering*, 16(9):1005–1022, septembre 1990.

[Gol92] D. M. Goldschlag. *Mechanically Verifying Concurrent Programs.* Thèse de Doctorat, University of Texas at Austin, 1992. research Report 71.

[Gor93] M.-J.-C. Gordon et T.-F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993. ISBN 0-521-44189-7.

[Gua92] D. Guaspari, C. Marceau, et W. Polak. Formal verification of ADA programs. Ursula Martin et Jeannette M. Wing, éditeurs, *First International Workshop on Larch*, pages 104–141. Springer-Verlag, July 1992.

[Gut74] J. V. Guttag. Dyadic specification and its impact on reliability. J. E. Donahue, J. D. Gannon, Guttag J. V., et J. J. Horning, éditeurs, *Three Approaches to Reliable Software: Language Design Dyadic Specification, Complementary Semantics.* University of Toronto, December 1974. TR CSRG–45.

[Gut78] J. V. Guttag et J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1), 1978.

[Gut83] J. V. Guttag et J. J. Horning. An introduction to the Larch shared specification. IFIP, éditeur, *Proceedings of the IFIP 9th World Computer Congress*, 1983.

[Gut86] J. V. Guttag et Horning J. J. Report on the Larch shared langage. *Science of Computer Programming*, 6(2):103–134, 1986.

[Gut93] J. V. Guttag, J.J. Horning, S.J Garland, K.D. Jones, A. Modet, et J. M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes.* Prentice Hall, Inc., 1985.

[Hun86] W. A. Hunt. A verified microprocessor. TR no. 47, Institute of Computer Science,University of Texas at Austin, Feb 1986.

[Jou87] J.-P. Jouannaud et P. Lescanne. La réécriture. *Techniques et Sciences Informatiques*, 5(6):433–452, 1987.

[Kal94] M. Kaltenbach. Model checking for UNITY: The uv system. Technical Report no. TR94-31, Department of Computer Science, The University of Texas at Austin, may 1994.

[Kna90] E. Knapp. An exercise in the formal derivation of parallel programs: Maximum flows in graphs. *ACM Transactions on Programming Languages and Systems*, 12(2):203–223, 1990.

[Kna94] E. Knapp. Soundness and completeness of UNITY logic. *Proc. of the 14th Conference FST and FCS'94*, volume 880, série *Lecture Notes in Computer Science*, pages 378–389. Springer-Verlag, 1994.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[Lam82] L. Lamport et S. Owicki. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, jul 1982.

[Lam91] L. Lamport. The temporal logic of actions. Rapport no. 79, Digital Systems Research Center, Palo Alto, California, USA, décembre 1991.

[Lam94] L. Lamport et S. Owicki. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, may 1994.

[Lau92] M. R. Laux, R. H. Bourdeau, et B. H. C. Cheng. An integrated environment supporting the reuse of formal specifications. TR no. MSU–CPS–ACS–70, Michigan State University, Department of Computer Science, September 1992.

[Les83] P. Lescanne. Computer experiments with the REVE term rewriting systems generator. *Proceedings of 10th ACM Symposium on Principles of Programming Languages*, pages 99–108. ACM, 1983.

[Man82] Z. Manna et A. Pnueli. Verification of concurrent programs: A temporal proof system. *Proceedings of the 4th School of Advanced Programming, Amsterdam (Holland)*, juin 1982.

[Man91] Z. Manna et A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems,Specification.* Springer-Verlag, 1991.

[Mel92] N. Mellergaard et J. Staunstrup. Generating proof obligations for circuits. Ursula Martin et Jeannette M. Wing, éditeurs, *First International Workshop on Larch*, pages 185–199. Springer-Verlag, July 1992.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92, série *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[Mis90] J. Misra. Soundness of the substitution axiom. Notes on Unity:14-90,unpublished manuscript, 1990.

[Owi76] S. Owicki et D. Gries. Verifying parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.

[Pet91] J. S. Pettersson. Comments on "always-true is not an invariant". *Information Processing Letters*, 40:231–233, 1991.

[Pet92] J. S. Pettersson. Assertional reasoning about invariance. Research Report no. TFL-RR-1992-3, Tele Denmark Research, juin 1992.

[Pet93] K. D. Petersen et J. S. Pettersson. Proving protocols correct. Research Report no. TFL-RR-1993-3, Tele Denmark Research, juin 1993.

[Pra94] I. S. W. B. Prasetya. Error in the UNITY substitution rule for subscripted operators. *Formal Aspects of Computing*, 6:466–470, 1994.

[Que81] J.-P. Queille et J. Sifakis. Specification and verification of concurrent systems in cesar. *Proceedings of the Fifth International Symposium in Programming*, 1981.

[SA93] J. F. Søgaard-Anderson, S. J. Garland, J. V. Guttag, N. A. Lynch, et A. Pogosyants. Computed-assisted simulation proofs. Costas Courcoubetis, éditeur, *Fifth Conference on Computer-Aided Verification*, pages 305–319, Heraklion, Crete, June 1993. Springer-Verlag Lecture Notes in Computer Science 697.

[San91] B. A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3:189–205, 1991.

[Sax92] J. B. Saxe, S. J. Garland, J. V. Guttag, et J. J. Horning. Using transformations and verification in circuit design. J. Staunstrup et R. Sharp, éditeurs, *International Workshop on Designing Correct Circuits.* North-Holland, IFIP Transactions A-5, 1992.

[Sco92] E. A. Scott et K. J. Norrie. Using LP to study the language PL. Ursula Martin et Jeannette M. Wing, éditeurs, *First International Workshop on Larch*, pages 227–245. Springer-Verlag, July 1992.

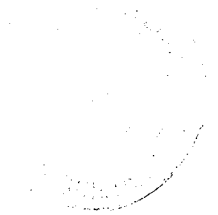
[Sta88] M. G. Staskauskas. The formal specification and design of a distributed electronic funds-transfer system. *IEEE Transactions on computers*, 37(12):1515–1528, 1988.

[Sta89] J. Staunstrup, S. J. Garland, et John V. Guttag. Localized verification of circuit descriptions. J. Sifakis, éditeur, *Proceedings of a Workshop on Automatic Verification Methods for Finite State Systems, Grenoble (France)*, volume 407, série *Lecture Notes in Computer Science*, pages 349–364. Springer-Verlag, juin 1989.

[Sta92] M. G. Staskauskas. *Specification and verification of large-scale Reactive Programs.* Thèse de Doctorat, University of Texas at Austin, 1992.

[Sta93] M. G. Staskauskas. Formal derivation of concurrent programs: An example from industry. *IEEE Transactions on Software Engineering*, 19(5):503–528, 1993.

[Win83] M. J. Wing. *A two-tiered Approach to specifying programs.* Thèse de Doctorat, Laboratory for Computer Science, Massachusetts Institute of Technology, mai 1983. Research report MIT/LCS/TR-299.



# Index

Nom : **CHETALI**

Prénom : **Boutheina**

**DOCTORAT de l'UNIVERSITE HENRI POINCARE, NANCY-I**

en INFORMATIQUE

VU, APPROUVÉ ET PERMIS D'IMPRIMER

Nancy, le - 5 JUIN 1996 UHP 32

Le Président de l'Université

UNIVERSITÉ HENRI POINCARÉ, NANCY 1 ★ LE PRÉSIDENT NANCY

3C-F9

UNIVERSITÉ HENRI POINCARÉ · NANCY 1
24-30, rue Lionnois B.P.3069 54013 Nancy cedex
Tél : 83 32 81 81 Fax : 83 32 95 90
Adresse électronique : @presn1.u-nancy.fr

## Résumé

Cette thèse est consacrée à l'utilisation des méthodes formelles de spécification et de vérification dans le cadre des techniques déductives basées sur la preuve de théorèmes. En particulier, nous nous intéressons à la spécification et à la vérification mécanique de programmes parallèles décrits en UNITY à l'aide du démonstrateur du LARCH, LP. Nous décrivons la formalisation et la mécanisation de la logique et de la méthodologie d'UNITY à l'aide d'un outil de démonstration automatique du premier ordre et à large spectre tel que LP et à leur mise en oeuvre dans des exemples utiles et conséquents. Nous formalisons dans un premier temps la syntaxe et la sémantique d'UNITY dans l'environnement de LP en choisissant comme outil formel la plus faible pré-condition introduite par Dijkstra. Cette modélisation comprend la représentation syntaxique concrète des objets prédicats, de la notation de programmation et des prédicats temporels de UNITY dans une logique du premier ordre. Nous décrivons la construction et la validation d'une base de faits basée sur l'approche des spécifications LSL. Nous proposons une méthodologie de preuve incrémentale basée sur l'utilisation d'un démonstrateur pour la vérification mécanisée dans le but à la fois d'aider à la mise au point des preuves et à la réutilisation des preuves.

Nous illustrons l'approche proposée à l'aide de trois études de cas. La vérification formelle mécanique d'un *protocole de communication* à travers des canaux défectueux met en évidence la méthodologie utilisée pour montrer des propriétés de sûreté et de vivacité et comment un démonstrateur peut être effectivement utilisé pour détecter des failles dans la spécification. La vérification du problème des *lecteurs rédacteurs* illustre un aspect important dans l'utilisation des démonstrateurs, à savoir la réutilisation et la mécanisation des preuves. Enfin, la vérification d'un protocole de *contrôle d'un ascenseur* permet de comparer notre approche à celle utilisée avec le démonstrateur d'ordre supérieur HOL.



**Mots-clés:** Vérification formelle, Vérification mécanique, Méthodologie de vérification, Preuves de Théorèmes, Protocole de communication, Unity



## Abstract

This thesis is an approach to the formal specification and verification of distributed systems and in particular to the computer assisted verification. In this work, we use the LARCH Prover to verify concurrent programs and the chosen specification mechanism is "UNITY logic". We describe the mechanization of the syntax and the semantic of UNITY in LP, and how we can use the theorem proving methodology to prove safety and liveness properties. We choose the calculus of the weakest pre-condition of Dijkstra as the formal tool to reason about UNITY programs. We choose a concrete syntactical representation of state dependent predicates and the programming notation in first order logic. To ensure the soundness of the encoding, we use an incremental method for the specifications and standard techniques for the verification.

To illustrate the feasibility of our approach, we present three case studies of different complexities. The first one describes the formal verification of a *communication protocol over faulty channels*, where we show how we can use the theorem proving methodology to prove safety and liveness of a communication protocol and how a theorem prover can be actually used to detect flaws in a system specification. The second study presents the mechanical proof of the correctness of the *reader-writer* problem. We use this small example to illustrate the level of details needed to mechanize a structured hand proof. The last study describes the proof of a *lift-control* protocol. As the original proof was developed with the HOL theorem prover, we use this example for a first comparison with the HOL proofs.



**Keywords:** Formal Verification, Theorem proving, Theorem Prover methodology, Computer checked proof, Communication Protocol, Unity