THÈSE

présentée à l'École Normale Supérieure de Cachan

pour obtenir le grade de

DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

par : Muriel Roger

Spécialité : Informatique

Raffinements de la résolution et vérification de protocoles cryptographiques

Soutenue le 24 octobre 2003

Composition du Jury :

– Guy Cousineau	rapporteur
– Didier Galmiche	rapporteur
– Jean Goubault-Larrecq	directeur de thèse
– Francis Klay	examinateur
– Yassine Lakhnech	président du jury
– Denis Lugiez	examinateur
– Michaël Rusinowitch	examinateur

Table des matières

Introduction

Le domaine de la démonstration automatique est certainement l'un des plus vieux domaines de l'informatique, qui prend ses sources avant l'apparition des premiers ordinateurs. Au départ, l'intérêt de la démonstration automatique repose sur deux aspects pour les mathématiciens, qui d'ailleurs donneront lieu à un certain nombre de polémiques : le premier est l'espoir d'obtenir ainsi une aide à la découverte de théorèmes, et le deuxième, plus réaliste, repose sur l'aspiration à une aide pour la vérification de théorèmes.

Les premiers programmes de démonstration automatique font leur apparition dans les années 1960. En 1965, J. A. Robinson invente la méthode de *résolution* sur les clauses de la logique du premier ordre [Rob65], qui révolutionnera le domaine. Cette méthode, fondée sur une seule règle d'inférence, facilement représentable sur ordinateur, est complète pour la logique du premier ordre. Elle peut cependant produire des milliers de clauses sans atteindre de preuve. Ainsi, trouver une preuve utilisant la méthode de résolution est vite devenu un problème de fournir des critères qui permettent de guider la procédure en interdisant d'effectuer la règle d'inférence sur certaines clauses, tout en préservant la complétude. Cette recherche a donné lieu à un certain nombre de raffinements de la résolution : l'hyperrésolution (positive et négative), la résolution ordonnée, la résolution linéaire, la résolution unitaire sur les clauses de Horn, la résolution ordonnée avec sélection sont des raffinements complets de la résolution. De plus, certains critères permettent d'identifier des clauses qui sont inutiles à la recherche de preuves, et ainsi de pouvoir les éliminer de l'ensemble courant des clauses sur lequel est effectué la résolution. Tout ceci fait que la méthode de résolution est devenue une procédure efficace et aujourd'hui, un certain nombre de démonstrateurs automatiques utilise la méthode de résolution, parmi lesquels nous pouvons citer OTTER [Kal01], Vampire [RV01], SPASS [Wei99a], qui sont dédiés à la logique du premier ordre.

De nos jours, les applications de méthodes de preuves automatiques servent en grande partie à l'informatique. Parmi celles-ci, nous pouvons citer l'analyse et la preuve de programmes, la validation de spécifications formelles, la vérification de protocoles logiciels ou matériels. Ces applications sont d'autant plus importantes lorsqu'elles concernent des logiciels critiques, dont les défaillances peuvent avoir des conséquences catastrophiques sur le plan humain. Ceci concerne plus spécifiquement les logiciels intervenant par exemple dans les domaines du transport, de la santé, de l'énergie. Cependant, les logiciels interviennent dans beaucoup d'autres secteurs d'activité, les télécommunications, le commerce électronique, les banques, et bien d'autres encore, dont les déficiences peuvent avoir des répercussions importantes sur un plan économique. C'est pourquoi le domaine de la sécurité informatique est en pleine expansion. Celle-ci présente deux aspects : l'un concerne ce que nous appellerons plutôt la *sûreté*, à savoir l'assurance que le logiciel fait bien ce pourquoi il a été conçu, et qu'il ne présente pas de défaillances, et l'autre, que nous appellerons plutôt la *sécurité*, vise à garantir qu'une (ou plusieurs) personne malveillante ne puisse pas altérer ou "pirater" des programmes ou des systèmes. C'est dans ce dernier cadre que se situe la vérification des protocoles cryptographiques, un domaine en pleine expansion depuis quelques années.

La cryptographie est une science très ancienne, qui est restée longtemps confinée à des applications militaires et diplomatiques, jusque dans les années 70 où une norme commerciale de déchiffrement a vu le jour, le Data Encryption Standard, sous la pression des milieux financiers et bancaires, étendant ainsi la cryptographie au domaine civil. Peu après, la notion de chiffrement à *clef publique* fait son apparition, qui sera abondamment utilisée par la suite et qui l'est toujours maintenant. Dès lors, la cryptographie est devenue un domaine de recherche ouvert. Aujourd'hui, un des enjeux de la vérification des protocoles cryptographiques est de se doter de méthodes systématiques et automatisables. On peut avancer le fait que la découverte d'une faille en 1995 dans le protocole de Needham et Schroeder, qui était connu depuis 17 ans, a fait prendre conscience à la communauté de l'importance qu'il y avait à rechercher du côté de l'automatisation de la vérification des protocoles cryptographiques.

Dans ce cadre, les modélisations de protocoles cryptographiques à l'aide de logiques clausales présentent l'intérêt qu'il existe déjà des méthodes de preuve efficaces, notamment la méthode de résolution. En particulier, le cadre de la logique du premier ordre permet à la fois une formalisation adéquate des protocoles cryptographiques et de certaines de leurs propriétés les plus courantes, et se prêtent bien à des mécanismes d'abstraction automatique qui en permettent une vérification automatique aisée. Ceci étant dit, certains protocoles modélisés en logique du premier ordre le sont modulo une théorie équationnelle qui peut varier d'un protocole à l'autre. Et de manière plus générale, au-delà même de la vérification des protocoles cryptographiques, selon le problème que l'on veut modéliser, on peut avoir besoin de différentes logiques clausales munies elle-même de différentes théories équationnelles, ce qui nous contraint à développer des outils de démonstrations spécifiques à chaque cas.

L'objectif de cette thèse est de définir un cadre général permettant de répondre le mieux possible à ce problème, tant d'un point de vue théorique que d'un point de vue pratique puisque nous présenterons un démonstrateur automatique modulaire que nous avons réalisé, basé sur un raffinement de la résolution. Un aspect important sera l'application à la vérification d'un protocole cryptographique réel faisant intervenir dans sa modélisation un opérateur associatif et commutatif : le protocole d'accord de clé en groupe IKA.1.

Cette thèse est structurée en deux parties. La première présente un état de l'art de certaines techniques de démonstration automatique, l'accent étant porté sur un de ses raffinements, la résolution ordonnée avec fonction de sélection, et un état de l'art de la vérification de protocoles cryptographiques, portant plus précisément sur les modèles à la Dolev-Yao. La deuxième partie présente une généralisation de la résolution ordonnée avec sélection telle qu'elle est définie au premier ordre, qui servira de base au démonstrateur automatique modulaire dont nous décrirons le fonctionnement. Enfin, nous développerons une modélisation d'un protocole cryptographique sous forme clausale, et examinerons les résultats fournis par le démonstrateur.

La première partie se fixe pour objectif de présenter deux états de l'art, l'un concernant des techniques de démonstration automatique, et l'autre la vérification des protocoles cryptographiques. Nous commencerons au chapitre 1, qui concernera uniquement la logique du premier ordre, par présenter la résolution ordonnée avec sélection ainsi que des résultats fondamentaux s'y rapportant, à savoir la correction et la complétude de ce raffinement de la résolution, diverses stratégies d'élimination de clauses redondantes, ainsi que deux techniques de splitting. Ces raffinements permettent notamment de démontrer la décidabilité de certaines sous-classes de la logique du premier ordre. Nous en profiterons pour exposer les liens particuliers qui existent entre les clauses de Horn et les automates d'arbres, en particulier le fait que ceux-ci peuvent être définis exactement comme un ensemble de clauses de Horn. Toutes ces notions introduites pour la logique du premier ordre nous serviront par la suite. Le chapitre 2 présente un aperçu de la vérification de protocoles cryptographiques, l'accent étant mis sur les modèles à la Dolev-Yao qui nous intéressent plus particulièrement dans le cadre de cette thèse. Après un bref historique, nous introduirons différentes notions, puis nous présenterons plus particulièrement les principes fondamentaux des modèles à la Dolev-Yao, ainsi que les méthodes de vérification mises en œuvre. Enfin, nous nous intéresserons plus spécifiquement à trois modèles existants qui se fondent sur les clauses de Horn.

Pour la deuxième partie, nous partons de l'idée énoncée plus haut de disposer d'un cadre général permettant de traiter différentes logiques clausales munies ou non d'une théorie équationnelle, et ainsi de pouvoir développer un outil de démonstration automatique. Le chapitre 3 a pour objectif de définir ce cadre général. Nous définirons ainsi une notion très générale de familles de *termes symboliques*, englobant notamment les termes finis de la logique du premier ordre, les termes rationnels, les termes modulo une théorie équationnelle, les λ-termes modulo β ou $\beta\eta$, ce que nous montrerons. Ceci nous permettra d'établir un langage de clauses dites *symboliques* construites sur les termes symboliques. Nous verrons que nous pouvons, sous certaines conditions assez naturelles, généraliser les résultats présentés dans le cas du premier ordre. La résolution abstraite ordonnée avec sélection que nous définirons nous fournira une procédure de résolution pour des ensembles de clauses symboliques dont nous démontrerons la correction et la complétude. Nous montrerons que cette procédure reste correcte et complète avec différentes stratégies d'élimination de clauses symboliques redondantes (les tautologies, les clauses symboliques subsumées linéairement et les clauses pures), ainsi qu'avec l'une ou l'autre des stratégies de splitting (le splitting par tableaux, et le splitting sans splitting).

Disposant ainsi d'une procédure de résolution correcte et complète sur des ensembles de clauses symboliques, nous présenterons au chapitre 4 l'architecture d'un démonstrateur automatique *modulaire* que nous avons implémenté, dont le noyau est basé sur la résolution abstraite ordonnée avec fonction de sélection, élimination des tautologies et des clauses subsumées linéairement, et splitting sans splitting. Ce noyau sera paramétré par un module de termes dits *abstraits*. Par opposition, les modules de termes dits *concrets* représenteront différentes familles de termes symboliques (les termes du premier ordre en sont l'exemple le plus trivial). Ainsi, étant donnée une famille de termes symboliques, il suffira de définir les fonctions qui leur sont propres pour disposer d'un démonstrateur automatique sur les clauses formées à partir de ces termes. Nous insisterons sur la modularité et sur l'ouverture, au détriment peut-être de l'efficacité.

Différents modules de termes concrets, que nous avons implémentés, seront présentés au chapitre 5 : les termes finis du premier ordre, les termes rationnels, et une certaine classe de termes du premier ordre modulo associativité et commutativité. Cette dernière classe de termes nous sera utile pour la vérification du protocole IKA.1 mentionné plus haut. En réalité, l'objectif sera de représenter des automates d'arbres bidirectionnels non alternants modulo associativité et commutativité, par un ensemble de clauses de Horn, le protocole IKA.1 étant modélisé sous cette forme. Cependant, nous verrons que, bien que la décidabilité d'une telle classe de clauses soit déjà établie, elle fournit une procédure peu utilisable en pratique. Nous montrerons que nous pouvons établir une procédure d'approximation, obtenant ainsi un algorithme de satisfiabilité d'automates d'arbres bidirectionnels modulo associativité et commutativité dont nous montrerons la complétude (l'approximation fonctionne aussi pour la variante *alternante* de cette classe qui est indécidable). En termes de vérification de protocoles cryptographiques, cela veut dire que si l'algorithme détermine qu'un ensemble S de clauses décrivant un protocole est satisfiable, alors il n'y a pas d'attaque sur le protocole. On obtiendra ainsi une preuve de sécurité. Nous décrirons enfin l'implémentation de cette approximation qui est non triviale.

Nous présenterons au chapitre 6 une modélisation du protocole cryptographique IKA.1 sous forme d'un ensemble de clauses de Horn décrivant un automate d'arbres bidirectionnel modulo associativité et commutativité. Nous décrirons les résultats pratiques que nous avons obtenus, puis nous présenterons des ouvertures sur différentes améliorations et optimisations à apporter. Au dernier chapitre, nous conclurons.

Première partie

État de l'art

Chapitre 1

Techniques de démonstration automatique

Le domaine de la démonstration automatique est vaste et présente un intérêt pour l'informatique. En effet, l'informatique est présente dans de multiples aspects de notre existence, notamment dans des secteurs aussi critiques que le transport ou la gestion de centrales nucléaires pour lesquels des vies humaines sont en jeu, mais aussi pour tout ce qui concerne les transactions financières, où l'on souhaite disposer de moyens sûrs.

C'est pourquoi on désire se doter de méthodes formelles de vérification, et dans ce sens, les techniques de démonstration automatiques y contribuent pour une large part.

Nous nous intéressons ici à un des multiples aspects de la démonstration automatique, la résolution ordonnée avec sélection, qui est un raffinement de la résolution inventée en 1965 par J.A. Robinson [Rob65], une des méthodes les plus efficaces pour décider l'insatisfiabilité d'un ensemble de clauses. Nous nous plaçons dans le cadre des clauses de la logique du premier ordre (les différentes démonstrations seront présentées dans un cadre plus général au chapitre 3). On pourra notamment consulter [BG01a] et [GLM97].

Nous introduirons tout d'abord quelques définitions utiles (section 1.1), avant d'en venir en section 1.2 à la méthode de résolution, puis à un de ces raffinements, la résolution ordonnée avec fonction de sélection (section 1.3). Nous verrons comment nous pouvons caractériser certaines clauses qui sont inutiles au processus de résolution, et que nous pourrons donc éliminer (section 1.4), puis nous introduirons deux techniques différentes de splitting qui sont des stratégies qui peuvent être utilisées conjointement à la résolution ordonnée avec sélection (section 1.5). Après avoir présenté quelques classes décidables de clauses de la logique du premier ordre (section 1.6), nous introduirons la notion de clauses de Horn, qui sont des clauses particulières très utiles en pratique, et qui permettent notamment de bien représenter les automates d'arbres que nous définirons (section 1.7).

1.1 Quelques définitions

Fixons un nombre infini dénombrable de symboles de prédicats n-aires. Un *atome* est un objet de la forme $P(t_1, \ldots, t_n)$, où P est un symbole de prédicat et $t_1, \ldots, t_n$ sont des termes du premier ordre sur une signature Σ.

Dans la suite, nous nous restreindrons toujours au cas $n = 1$, sauf mention contraire ; les atomes seront donc de la forme $P(t)$, où P est un symbole de prédicat unaire et t un terme du premier ordre sur la signature Σ. Ceci simplifiera les notations, et n'entache pas la généralité puisque l'on peut toujours coder $P(t_1, \ldots, t_n)$ en $P(f_P(t_1, \ldots, t_n))$, où P est maintenant unaire et il y a bijection entre les symboles P et les symboles n-aires f_P.

Un *littéral* est soit un littéral *positif* $+P(t)$, soit un littéral *négatif* $-P(t)$. Une *clause* est une disjonction de littéraux $\pm_1 P_1(t_1) \vee \ldots \vee \pm_k P_k(t_k)$. Si $k = 0$, on écrit $\Box$ cette disjonction ; il s'agit de la clause vide.

Certaines clauses sont particulièrement importantes, il s'agit des clauses *de Horn*, qui possèdent au plus un littéral positif. Les clauses de Horn sont regroupées en clauses définies et en clauses buts. Les *clauses définies* sont les clauses ayant exactement un littéral positif. On notera la clause définie $+P(t) \vee -P_1(t_1) \vee \ldots \vee -P_n(t_n)$ comme suit :

$$P(t) \Leftarrow P_1(t_1), \ldots, P_n(t_n)$$

Dans le cas où $n = 0$, on notera aussi cette clause $P(t)$ et on l'appellera un *fait*.

Un *but*, aussi appelé *clause négative*, est une clause ne contenant que des littéraux négatifs. Tout but est donc une clause de Horn. On écrit parfois le but $-P_1(t_1) \vee \ldots \vee -P_n(t_n)$ sous la forme :

$$\perp \Leftarrow P_1(t_1), \ldots, P_n(t_n)$$

où $\perp$ dénote le faux.

De manière symétrique, une clause positive est une clause ne contenant que des littéraux positifs.

Un terme, un atome, un littéral, une clause sont *clos* si et seulement s'ils ne contiennent aucune variable libre. Une *substitution* σ est une application des variables vers les termes. L'application d'une substitution σ à un terme t, notée $t\sigma$ est telle que $x\sigma = \sigma(t)$, $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$. Une *instance* de t est tout terme de la forme $t\sigma$, pour une substitution σ. On notera $\sigma\theta$ la *composition* des deux substitutions σ et θ, définie comme l'unique substitution telle que $t(\sigma\theta) = (t\sigma)\theta$ pour tout terme t : $\sigma\theta$ associe à toute variable x le terme $\sigma(x)\theta$. Un *renommage* est une substitution qui est une bijection des variables vers les termes.

L'*univers de Herbrand* H est l'ensemble de tous les termes clos. Une *interprétation de Herbrand* est une structure I qui est la donnée de l'univers de Herbrand H, de sous-ensembles I_P de H, un pour chaque prédicat P, et d'applications $I_f : H^n \to H$, une pour chaque fonction $f \in \Sigma$ d'arité n, qui envoie tout n-uplet de termes $t_1, \ldots, t_n$ vers le terme clos $f(t_1, \ldots, t_n)$. Étant donné un *environnement clos* ρ, c'est-à-dire une application qui associe à chaque variable un élément de H, la *valeur* $I[\![t]\!]\rho$ d'un terme t est définie par :

- $I[\![x]\!]\rho = \rho(x)$ pour chaque variable x
- $I[\![f(t_1, \ldots, t_n)]\!]\rho = I_f(I[\![t_1]\!]\rho, \ldots, I[\![t_n]\!]\rho)$.

On définit la relation $I, \rho \models P(t)$, et on dit que $P(t)$ est *vrai* dans I, ρ si et seulement si $I[\![t]\!]\rho$ est dans I_P.

Pour toute clause C, on pose $I, \rho \models C$ si et seulement s'il existe un littéral $+P(t)$ dans C tel que $I, \rho \models P(t)$, ou un littéral $-P(t)$ dans C tel que $I, \rho \not\models P(t)$.

La structure I est un *modèle de Herbrand* de la clause C si et seulement si pour tout environnement clos ρ, on a $I, \rho \models C$; on écrit alors $I \models C$. I est un modèle de Herbrand de l'ensemble de clauses S si et seulement si $I \models C$ pour toute clause C dans S ; on écrit alors $I \models S$.

Proposition – 1.1.1

Un ensemble de clauses S est satisfiable si et seulement s'il a un modèle de Herbrand.

1.2 La résolution

Une des techniques de démonstration automatique les plus connues et les plus efficaces pour détecter si un ensemble de clauses est insatisfiable, est la *résolution* inventée par J. Alan Robinson en 1965.

La résolution au premier ordre utilise la notion d'*unification*. Un *unificateur* σ de deux termes ou atomes s et t est une substitution telle que $s\sigma = t\sigma$. La substitution σ est *plus générale* que σ' si et seulement s'il existe une troisième substitution θ telle que $\sigma' = \sigma\theta$. Un *unificateur le plus* *général*, ou *mgu* (de l'anglais "*most general unifier*") de s et t est un unificateur qui est plus général que tout autre unificateur de s et t.

Si s et t sont unifiables, alors ils ont un unificateur le plus général. On notera $\mathsf{mgu}(s \doteq t)$ l'un de ces mgu, et $\mathsf{mgu}(t_1 \doteq t, \ldots, t_n \doteq t)$ un mgu σ qui unifie simultanément $t, t_1, \ldots, t_n$, c'est-à-dire tel que $t_1\sigma = t\sigma, \ldots, t_n\sigma = t\sigma$. On peut montrer que les unificateurs les plus généraux ne diffèrent que par un renommage. On peut aussi montrer qu'il existe parmi les unificateurs les plus généraux certains σ qui sont *idempotents*; autrement dit $\sigma\sigma = \sigma$. Cette condition est équivalente au fait qu'aucune variable du domaine $\mathsf{dom}(\sigma) = \{x \mid \sigma(x) \neq x\}$ n'est libre dans aucun terme $\sigma(x)$, $x \in \mathsf{dom}(\sigma)$.

La règle de *résolution binaire* est :

$$\frac{C \vee +A \quad C' \vee -A'}{C\sigma \vee C'\sigma}\ \sigma = \mathsf{mgu}(A \doteq A')$$

où la condition $\sigma = \mathsf{mgu}(A \doteq A')$ signifie que A et A' sont unifiables, et que σ est leur mgu. D'autre part, $C \vee +A$ en prémisse sous-entend que $+A$ n'est pas dans la disjonction C, et de façon similaire pour $C \vee -A'$. On sous-entend aussi que les deux prémisses ont des ensembles de variables disjoints, ce qui peut être effectué en renommant l'une ou l'autre des deux prémisses. On appelle la conclusion le *résolvant binaire* des deux prémisses.

Cette règle se spécialise dans le cas des clauses de Horn à la règle :

$$\frac{A \Leftarrow A_1, \ldots, A_m \quad B \Leftarrow B_1, \ldots, B_n}{A\sigma \Leftarrow B_1\sigma, \ldots, B_n\sigma, A_2\sigma, \ldots, A_m\sigma}\ \sigma = \mathsf{mgu}(A_1 \doteq B)$$

La règle de résolution binaire pour les clauses de Horn est complète : si un ensemble S de clauses de Horn est insatisfiable, alors on peut déduire la clause vide $\Box$ en un nombre fini d'étapes de résolution binaire.

Dans le cas général, la résolution binaire n'est pas complète. Par exemple, l'ensemble de clauses suivant est insatisfiable :

$$+P(f(a,x)) \vee +P(f(x,a)) \qquad -P(f(a,y)) \vee -P(f(y,a))$$

Ici a est une constante, et x, y sont des variables. Cependant, les seuls résolvants binaires de ces deux clauses sont $+P(f(x,a)) \vee -P(f(x,a))$ et $+P(f(a,a)) \vee -P(f(a,a))$, et à partir de ceux-là, aucune nouvelle clause n'est produite ; en particulier, la clause vide ne sera jamais produite.

Ceci est facile à corriger ; il suffit de nous donner la possibilité d'utiliser une règle additionnelle, dite de *factorisation* (positive) :

$$\frac{C \vee +A \vee +A'}{C\sigma \vee +A\sigma}\ \sigma = \mathsf{mgu}(A \doteq A')$$

On notera que cette règle n'est jamais applicable sur les clauses de Horn puisque celles-ci possèdent au plus un littéral positif.

On a alors que le résultat de la combinaison des deux règles de résolution binaire et de factorisation positive est complète : tout ensemble insatisfiable de clauses admet une dérivation de la clause vide $\Box$ en un nombre fini d'étapes à l'aide de ces deux règles. Cette combinaison des deux règles est appelée la *résolution*.

Dans l'exemple ci-dessus, on dérive $+P(f(a,a))$ par factorisation. En résolvant $-P(f(a,y)) \vee -P(f(y,a))$ avec $+P(f(a,a))$, on obtient $-P(f(a,a))$; en résolvant cette dernière clause avec $+P(f(a,a))$, on obtient la clause vide.

On appelle *facteur* (positif) d'une clause C toute clause C' que l'on peut obtenir en appliquant 0, une ou plusieurs fois la règle de factorisation (positive). On appelle *résolvant* de deux clauses C_1 et $C' \vee -A'$ tout résolvant binaire $C \vee C'$ entre un facteur $C \vee +A$ de C_1 et $C' \vee -A'$.

Lemme – 1.2.1

> **Correction** *Soit S un ensemble de clauses, et S' n'importe quel ensemble de clauses obtenues à partir de S par résolution binaire et factorisation positive. Tout modèle de S est un modèle de S'.*
> *En particulier, si $\Box$ est déductible par résolution à partir de S, alors S est insatisfiable.*

1.3 La résolution ordonnée avec sélection : un raffinement de la résolution

Il existe un certain nombre de restrictions de la résolution qui restent complètes :

- On peut restreindre la prémisse $C \vee +A$ de la règle de résolution binaire à être une clause positive ; cette restriction est appelée la règle d'*hyperrésolution positive*.
  Dans le cas des clauses de Horn, ceci revient à restreindre cette prémisse à être une clause réduite à un seul littéral positif $+A$, c'est-à-dire à être un fait.

- La restriction de la règle de résolution binaire où l'une des deux prémisses de la règle de résolution binaire est une clause *unitaire*, c'est-à-dire ne contenant qu'un littéral, est appelé *résolution unitaire*. La résolution unitaire est donc complète pour les clauses de Horn, mais ne l'est pas dans le cas général.
- On peut restreindre la prémisse $C \vee -A$ de la règle de résolution binaire à être une clause négative ; on obtient ainsi l'*hyperrésolution négative*.
- En général, la *résolution sémantique*, où une des prémisses est contrainte à être fausse dans une interprétation I fixée, est complète. On notera que l'on retrouve l'hyperrésolution positive en prenant pour I l'interprétation de Herbrand qui rend faux tous les atomes clos, et l'hyperrésolution négative en prenant l'interprétation de Herbrand qui rend vrais tous les atomes clos.
- La *résolution ordonnée* demande que :
  - A soit maximal dans la clause $C \vee +A$, et A' maximal dans la clause $C' \vee -A'$, où $C \vee +A$ et $C' \vee -A'$ sont les prémisses de la règle de résolution binaire.
  - A et A' soient tout deux maximaux dans la prémisse $C \vee +A \vee +A'$ de la règle de factorisation positive,

  ceci pour un ordre stable $\succ$ sur les atomes. On dit que $\succ$ est *stable* si et seulement si $A \succ B$ implique $A\sigma \succ B\sigma$ pour toute substitution telle $A\sigma$ et $B\sigma$ soient clos.
  Précisons qu'un atome A est *maximal* dans une clause C s'il n'existe pas de littéral $\pm B$ dans C tel que $B \succ A$.
  La résolution ordonnée est un raffinement complet de la résolution qui est très restrictif.
- La *résolution ordonnée avec sélection* est similaire mais utilise en plus une fonction sel, dite de *sélection*, qui à chaque clause C associe un sous-ensemble (possiblement vide) de ses littéraux négatifs. L'idée est que l'on utilise une fonction de sélection pour déterminer sur quel littéral résoudre ; on n'utilise l'ordre que par défaut. Précisément, la règle de résolution ordonnée avec sélection est décrite en figure 1.1.
  On a ici combiné résolution binaire et factorisation positive en une seule règle. Il est à noter que la fonction sel est *quelconque*, et n'est restreinte que par le fait qu'elle doit sélectionner des littéraux négatifs. (On pourrait les choisir positifs, ceci ne changerait rien en terme de complétude.)
  Si l'on choisit comme fonction de sélection la fonction qui retourne toujours l'ensemble vide, on retrouve la résolution ordonnée. Si au contraire $\mathsf{sel}(C)$ retourne l'ensemble de tous les littéraux négatifs de C, on obtient un raffinement de l'hyperrésolution négative où les littéraux positifs sur lesquels on résout sont restreints à être maximaux dans leur clause.

$$\frac{C \vee +A_1 \vee \ldots \vee +A_n \quad C' \vee -A'}{(C \vee C')\sigma}$$

où :

(i) $n \geq 1$;

(ii) $\sigma = \mathsf{mgu}\ (A_1 \doteq A', \ldots, A_n \doteq A')$;

(iii) $\mathsf{sel}(C \vee +A_1 \vee \ldots \vee +A_n) = \emptyset$ et $A_1, \ldots, A_n$ sont maximaux dans $C \vee +A_1 \vee \ldots \vee +A_n$;

(iv) $-A' \in \mathsf{sel}(C' \vee -A')$, ou bien $\mathsf{sel}(C' \vee -A') = \emptyset$ et A' est maximal dans $C' \vee -A'$.

Fig. 1.1: Résolution ordonnée avec sélection au premier ordre

Théorème – 1.3.1

> **Complétude** *La résolution ordonnée avec sélection est complète : pour tout ordre stable $\succ$, pour toute fonction de sélection, l'ensemble de clauses S est insatisfiable si et seulement si l'on peut dériver la clause vide $\Box$ à partir de S avec la seule règle de résolution ordonnée avec sélection.*

1.4 Stratégies d'élimination de clauses

En pratique, il est utile, voire fondamental, de combiner la règle de résolution avec différentes règles de *simplification* de l'ensemble de clauses courant.

1.4.1 Élimination de tautologies

On dit qu'une clause C est une *tautologie* si et seulement si elle est de la forme $C \vee +A \vee -A$ pour un certain atome A. On peut alors éliminer toutes les tautologies de S.

Lemme – 1.4.1

> *La résolution ordonnée avec sélection et élimination des clauses tautologiques est complète.*

1.4.2 Élimination de clauses subsumées

On dit qu'une clause C *subsume* une clause C' si et seulement si C' est de la forme $C\sigma \vee C_1$. On notera que C implique $C\sigma$, qui implique $C\sigma \vee C_1$. Donc, si C subsume C', alors C implique C'. On verra que l'on peut très souvent éliminer les clauses subsumées.

Le fait que $S \setminus \{C\}$ est satisfiable si et seulement si S est satisfiable est une bonne indication que l'on peut enlever C de S. Mais ce n'est pas une condition suffisante. Un cas intéressant est celui où l'on sépare la règle de résolution en résolution binaire et factorisation positive : tout facteur $C\sigma$ de C est subsumé par C, donc si l'on pouvait éliminer toute clause subsumée, alors la résolution binaire serait complète sans factorisation ; nous avons vu que ce n'était pas le cas.

Nous appellerons une clause subsumée $C\sigma \vee C_1$ telle que σ n'unifie aucune paire de littéraux de C, une clause *subsumée linéairement.* On a alors le résultat suivant :

Lemme – 1.4.2

> *La résolution ordonnée avec sélection et élimination des clauses subsumées linéairement est complète.*

1.4.3 Élimination des clauses pures

On dit qu'une clause est *pure* dans un ensemble de clauses S si et seulement si C est de la forme $C_1 \vee \pm A$, et il n'existe aucune clause de la forme $C_1' \vee \mp A'$ dans S telle que A et A' sont unifiables (on dit que $\pm A$ est *pur*).

Lemme – 1.4.3

La résolution ordonnée avec sélection et élimination des clauses pures est complète.

L'élimination de clauses pures est une stratégie d'élimination de clauses redondantes qui est parfois très efficace : supprimer une clause redondante (pure notamment) peut rendre pures d'autres clauses, et ainsi de suite en chaîne.

1.5 Splitting

Considérons une clause de la forme $C \vee C'$, où C et C' ne partagent aucune variable libre, et un ensemble de clauses de la forme $S \cup \{C \vee C'\}$. Ce dernier est insatisfiable si et seulement si $S \cup \{C\}$ et $S \cup \{C'\}$ sont insatisfiables.

Par exemple, $S' = S \cup \{+P(x) \vee +Q(y)\}$ est satisfiable par tout modèle qui satisfait S d'une part, et tel que P est vrai de tout x, ou bien Q est vrai de tout y. Donc S' est satisfiable si et seulement s'il existe un modèle de $S \cup \{+P(x)\}$, ou un modèle de $S \cup \{+Q(y)\}$. On remarquera qu'il est important que C (ici $+P(x)$) et C' (ici $+Q(y)$) ne partagent aucune variables.

Nous allons examiner deux façons d'exploiter cette remarque.

1.5.1 Splitting et tableaux

Une première approche consiste à découper l'ensemble de clauses $S \cup \{C \vee C'\}$ en deux ensembles de clauses $S \cup \{C\}$ et $S \cup \{C'\}$, et on aura alors besoin de manipuler plusieurs ensembles de clauses. Un *tableau* $\mathcal{T}$ est un multi-ensemble fini non vide d'ensembles de clauses. Le tableau formé des ensembles $S_1, \ldots, S_n$ de clauses sera noté $S_1|\ldots|S_n$, et l'union des multi-ensembles des tableaux $\mathcal{T}$ et $\mathcal{T}'$ sera noté $\mathcal{T}|\mathcal{T}'$. Chaque ensemble de clause S_i est appelé une *branche* du tableau $\mathcal{T}$. On dira qu'une branche est *close* si et seulement si elle contient la clause vide $\Box$, et qu'un tableau $\mathcal{T}$ est *clos* si et seulement si toutes ses branches sont closes. Notons $\rhd$ la relation binaire de déduction définie par $S \rhd S'$ si et seulement si S' est obtenu par l'ajout à S d'un résolvant ordonné avec sélection ou par l'élimination d'une clause tautologique ou d'une clause subsumée linéairement par une autre clause de S (il suffirait que $\rhd$ soit une relation binaire quelconque entre ensembles de clauses, mais nous nous plaçons ici dans le cas qui nous intéresse plus particulièrement). On peut alors formaliser le splitting à l'aide de la définition de la relation binaire $\rightsquigarrow$ entre tableaux par les règles suivantes :

$$\frac{S \rhd S'}{S|\mathcal{T} \rightsquigarrow S'|\mathcal{T}}\,(\rhd) \qquad \frac{C, C' \text{ n'ont aucune variable libre en commun}}{S \cup \{C \vee C'\}|\mathcal{T} \rightsquigarrow S \cup \{C\}|S \cup \{C'\}|\mathcal{T}}\,(Split)$$

On dit qu'une interprétation I *satisfait* un tableau $\mathcal{T}$ si et seulement si I satisfait au moins une branche du tableau $\mathcal{T}$. En particulier, $\mathcal{T}$ est *insatisfiable* si et seulement si toutes ses branches sont insatisfiables.

Notons que $\rhd$ est correcte au sens où si $S \rhd S'$ et S est satisfiable, alors S' est satisfiable, et qu'elle est complète dans un sens analogue.

Lemme – 1.5.1

La relation $\rightsquigarrow$ est correcte, au sens où si $\mathcal{T} \rightsquigarrow^* \mathcal{T}'$ et $\mathcal{T}$ est satisfiable, alors $\mathcal{T}'$ est satisfiable. En particulier, si $\mathcal{T} \rightsquigarrow^* \mathcal{T}'$ et $\mathcal{T}'$ est un tableau clos, alors $\mathcal{T}$ est insatisfiable.

Théorème – 1.5.1

> *La résolution ordonnée avec sélection, élimination de tautologies et de clauses subsumées linéairement et utilisation de la règle de splitting (Split), est complète.*

1.5.2 Splitting sans splitting

Nous allons examiner une autre approche qui permet de simuler en quelque sorte le splitting classique décrit ci-dessus : c'est le *splitting sans splitting*.

On peut simuler le splitting en utilisant l'équivalence entre $C \vee C'$ et $\exists q.(q \Rightarrow C) \wedge (\neg q \Rightarrow C')$, lorsque C et C' ne partagent aucune variable libre. Ceci permet de remplacer la disjonction $C \vee C'$ par la *conjonction* des deux clauses $-q \vee C$ et $+q \vee C'$, où q est un symbole de prédicat frais d'arité 0.

Ceci permet d'éviter d'avoir à raisonner sur des multi-ensembles de clauses (comme dans le cas du splitting classique), et de manipuler uniquement des ensembles de clauses. Dans la suite, pour avoir à éviter de parler de symbole de prédicat d'arité 0, on écrira $q(*)$ à la place de q, où $*$ est une constante fixée.

Pour formaliser ceci, nous avons besoin de préciser ce qu'un symbole de prédicat *frais* signifie. Fixons donc l'ensemble $\mathcal{P}$ des symboles de prédicats pouvant apparaître dans un ensemble de clauses S fourni en entrée de notre procédure de résolution. Soit $\mathcal{Q}$ un ensemble de symboles de prédicats disjoint de $\mathcal{P}$. Supposons de plus que $\mathcal{Q}$ est en bijection avec l'ensemble des clauses formées à l'aide de prédicats dans $\mathcal{P}$ seulement, modulo renommage. Autrement dit, il existe une application qui à chaque clause formée à l'aide de prédicats dans $\mathcal{P}$ uniquement associe un symbole $\ulcorner C \urcorner$ de $\mathcal{Q}$, telle que $\ulcorner C \urcorner = \ulcorner C' \urcorner$ si et seulement s'il existe un renommage ρ tel que $C = C'\rho$. Cette construction existe toujours : il suffit de choisir pour $\mathcal{Q}$ exactement le quotient de l'ensemble des clauses formées à l'aide de prédicats dans $\mathcal{P}$ par la relation d'équivalence reliant toute clause à son renommage $C\rho$.

La règle de *splitting sans splitting* est alors :

$$\frac{C \vee C'}{C \vee -\ulcorner C' \urcorner(*) \qquad +\ulcorner C' \urcorner(*) \vee C'}$$

par laquelle on entend que C et C' sont deux sous-clauses telles que :

- C et C' n'ont aucune variable en commun ;
- C' est formé de symboles de prédicats dans $\mathcal{P}$ uniquement ;
- C contient au moins un atome $P(t)$ avec $P \in \mathcal{P}$.

L'effet de la règle est de remplacer $C \vee C'$ par les deux clauses $C \vee -\ulcorner C' \urcorner(*)$ et $+\ulcorner C' \urcorner(*) \vee C'$.

On a des résultats de correction et de complétude pour la résolution ordonnée avec sélection, l'élimination de tautologies et de clauses subsumées linéairement, et la règle de splitting sans splitting. Nous en verrons la démonstration dans un cadre plus général en chapitre 3. La règle de splitting sans splitting apparaît pour la première fois dans [RV01] sous le nom de *splitting sans retour en arrière* et est utilisé dans les prouveurs Vampire [RV01] et Waldmeister [Hil03].

L'utilisation du splitting sans splitting a de nombreux avantages. D'abord, cela permet l'utilisation de toutes les stratégies d'élimination de clauses redondantes décrites en section 1.4, alors que le splitting classique interdit l'élimination des clauses d'une branche qui sont redondantes par rapport à des clauses d'une autre branche. Ensuite, en pratique, les démonstrateurs automatiques par résolution utilisent bien souvent des langages impératifs, et effectuer du splitting ordinaire dans ce contexte est parfois difficile : il ne faut pas qu'une étape de résolution sur une branche modifie par inadvertance les données d'une autre branche. Un des rares prouveurs effectuant le splitting classique est SPASS [Wei99a]. Finalement, l'utilisation du splitting sans splitting peut être plus efficace dans le cas des clauses de Horn, en utilisant des fonctions de sélection adaptée. Par contre, dans le cas de clauses qui ne sont pas de Horn, le splitting classique est plus efficace.

1.6 Quelques classes décidables de clauses

Il est bien connu que la logique du premier ordre est indécidable. Il existe cependant des classes pour lesquelles la décidabilité a été prouvée, et nous allons en présenter ici quelques unes. Pour un panorama plus complet, on pourra consulter [FLHT01].

1.6.1 Classe de Herbrand

La classe de Herbrand est la classe des ensembles de clauses n'ayant qu'un seul littéral. On les appelle les clauses *unitaires*. L'insatisfiabilité d'un tel ensemble de clauses S est trivialement décidable par résolution : soit il existe $+A$ et $-A$ unifiables dans l'ensemble de clauses S et S est insatisfiable, soit il n'en existe pas et S est satisfiable.

1.6.2 Classe d'Ackermann

Les clauses C de la classe d'Ackermann contiennent au plus une variable x et sont constituées de littéraux de la forme $\pm P(t_1, \ldots, t_n)$, où chaque t_i est une constante, ou la variable x, ou un terme de la forme $f(x)$ (avec la même variable x).

On peut montrer que la résolution ordonnée termine sur des ensembles de clauses de cette forme. On définit la *profondeur* $d(t)$ d'un terme ou un atome t par

- $d(x) = 0$;
- $d(a) = 0$, où a est une constante ;
- $d(f(t_1, \ldots, t_n)) = \max_{1 \leq i \leq n} d(t_i) + 1$.

Posons alors $A > B$ si et seulement si $d(A) > d(B)$. On peut montrer que cet ordre est stable sur l'ensemble des atomes de la forme définie précédemment, c'est-à-dire que pour tout A, B et pour toute substitution close σ, si $A > B$ alors $A\sigma > B\sigma$. En effet, soient A et B deux atomes de la forme décrite ci-dessus, alors si $d(A) > d(B)$, on a forcément $d(A) = 2$ et $d(B) = 1$, où au moins un des termes présent dans A est de la forme $f(x)$ ou est une constante, et tous les termes présents dans B sont la variable x. Soit σ une substitution qui à x associe un terme clos quelconque t, alors $d(A\sigma) = 2 + d(t)$ et $d(B\sigma) = 1 + d(t)$, donc $A\sigma > B\sigma$. Ainsi l'ordre $>$ est stable.

On peut ensuite montrer que tout résolvant ordonné $(C \vee C')\sigma$ de $C \vee +P(t_1, \ldots, t_n)$ et de $C' \vee -P(t'_1, \ldots, t'_n)$, où σ est un unificateur de $P(t_1, \ldots, t_n)$ et $P(t'_1, \ldots, t'_n)$ est toujours dans la classe d'Ackermann. En effet, si σ est un tel unificateur, alors pour tout i, $1 \leq i \leq n$, σ est un unificateur de t_i et t'_i. Les cas les plus simples sont les suivants : soit t_i et t'_i sont deux constantes, soit t_i et t'_i sont deux variables x et x', soit t_i et t'_i sont respectivement $f(x)$ et $f(x')$ (avec les même variables x et x' que précédemment). Dans tous ces cas, on aura alors un unificateur qui à une variable en associe une autre (ou trivialement la substitution vide dans le cas où tous les couples (t_i, t'_i) sont égaux). Cependant, le cas où on unifie deux littéraux ayant des constantes en arguments peut produire des clauses qui ne sont plus dans la classe. Considérons par exemple la résolution ordonnée binaire suivante :

$$\frac{+P(x) \vee +P(c) \qquad -P(c) \vee +P(x')}{+P(x) \vee +P(x')}$$

Les variables x et x' étant différentes (quitte à renommer une clause), la clause $+P(x) \vee +P(x')$ n'est pas dans la classe d'Ackermann. C'est pourquoi on a besoin d'utiliser du splitting, ce qui permettra de retrouver des clauses qui sont dans la classe.

Deux autres cas (ainsi que les cas symétriques) peuvent se présenter :

- si $t_i = x$ et $t'_i = f(x')$, soit σ un unificateur de t_i et t'_i qui à x associe $f(x')$. Comme t_i est une variable, on a $d(P(t_1, \ldots, t_n)) = 1$. Or, $P(t_1, \ldots, t_n)$ étant maximal pour $>$ dans la clause $C \vee +P(t_1, \ldots, t_n)$, on a nécessairement que tous les littéraux A présents dans C sont tels que $d(A) = 1$. Donc $C\sigma$ est de la bonne forme, et la clause $(C \vee C')\sigma$ aussi.
- si $t_i = x$ et $t'_i = c$, le même raisonnement s'applique.

Ainsi, le résolvant binaire ordonné est toujours dans la classe d'Ackermann.

Par ailleurs, il n'y a qu'un nombre fini de telles clauses. Donc, la résolution ordonnée termine.

1.6.3 Classe monadique

La classe monadique est habituellement définie comme la classe des formules de la forme

$$\begin{array}{rcl} F & := & \pm P(x) \\ & | & F \wedge F \\ & | & F \vee F \\ & | & \forall.F \\ & | & \exists.F \end{array}$$

À part la restriction que les arguments des symboles de prédicats sont des variables, comme dans les classes décidables mentionnées plus haut, la restriction essentielle de la classe monadique est que tout symbole de prédicat est d'arité 1.

On peut montrer directement que cette classe est décidable, mais nous préférons envisager cette classe sous la forme de clauses, ce qui ne sera pas sans lien avec la notion d'automates d'arbres que nous allons aborder à la section suivante.

Nous allons donner les idées générales permettant de transformer le problème de satisfiabilité de formules monadiques en un problème de satisfiabilité de clauses, que l'on peut résoudre par résolution.

Nous allons nous servir des notions de *mise sous forme prénexe* et de *skolémisation* de formules. Le lecteur non familiarisé avec ces notions pourra, parmi un éventail assez large, consulter notamment [GLM97, Lal90, CJ97].

La mise sous forme clausale d'une formule F de la classe monadique fournit des clauses qui ont une forme particulière. Si l'on met d'abord F en forme prénexe, on obtient une formule de la forme $Q_1 z_1 \ldots Q_n z_n.G$, où G est une formule sans quantificateur, et les Q_i sont des quantificateurs pris dans $\{\exists, \forall\}$. Si maintenant on skolémise, on obtient la formule $G\sigma$, où σ est la substitution qui à z_i associe z_i si $Q_i = \forall$, et $g_i(\overrightarrow{z}_{\forall,<i})$ si $Q_i = \exists$, où les symboles de fonction g_i sont distincts deux à deux, et $\overrightarrow{z}_{\forall,<i}$ dénote la liste de toutes les variables z_j telles que $j < i$ et $Q_j = \forall$. Il ne reste qu'à mettre $G\sigma$ sous une forme clausale S. Comme F est monadique, les atomes de $G\sigma$ et donc de S sont de la forme $P(z_i)$ (avec $Q_i = \forall$) ou de la forme $P(f(\overrightarrow{z}_{\forall,<i}))$. Renumérotons les z_i tels que $Q_i = \forall$, et appelons-les $x_1, \ldots, x_m$ dans cet ordre. Renumérotons aussi les g_i sous la forme $f_1, \ldots, f_p$. Les atomes de S sont donc de la forme $P(x_i)$ ou $P(f_i(x_1, \ldots, x_{n_i}))$, avec $0 \leq n_1 \leq \ldots \leq n_p \leq m$.

Appelons *terme plat* tout terme de la forme $f(X_1, \ldots, X_n)$, où f est un symbole de fonction n-aire, et $X_1, \ldots, X_n$ sont des variables non nécessairement distinctes. Notons $\sqsubseteq$ l'ordre sur les termes plats défini par : $f((X_1, \ldots, X_m) \sqsubseteq g(Y_1, \ldots, Y_n)$ si et seulement s'il existe un entier k, $0 \leq k \leq n$, tel que $\{X_1, \ldots, X_m\} = \{Y_1, \ldots, Y_k\}$. Par exemple, $f(X) \sqsubseteq g(X, Y)$, $a \sqsubseteq f(X)$, mais $g(X, Y) \not\sqsubseteq f(X)$. Il est clair que toute clause de S est formée d'atomes $P(t)$, où t est soit une variable, soit un terme plat.

Toute clause ainsi obtenue a donc une décomposition en blocs d'une des deux formes :

1. Un ϵ-*bloc* de la forme
$$\pm_1 P_1(X) \vee \ldots \vee \pm_n P_n(X)$$
où $n \geq 0$, et tous les prédicats P_i sont appliqués à la même variable X, on notera souvent $B(X)$ un ϵ-bloc dont la seule variable libre (si elle existe) est X.
2. Une *transition étendue* de la forme
$$\bigvee_{i=1}^{m} \pm_i P_i(t_i) \vee B_1(X_1) \vee \ldots \vee B_n(X_n)$$
où $m \geq 1$, les $B_j(x_j)$ sont des ϵ-blocs, les t_i sont des termes plats tels que
   – $t_1 \sqsubseteq t_2 \sqsubseteq \ldots \sqsubseteq t_m$;
   – et $X_1, \ldots, X_n$ sont toutes libres dans t_m.

En fait, la mise sous forme clausale de formules de la classe monadique produit même des clauses plus particulières, au sens où les X_i sont distinctes deux à deux dans les transitions étendues.

Soit $\succ_1$ l'ordre défini par : $P(t) \succ_1 P'(t')$ si et seulement si t' est un sous-terme strict de t. On peut alors montrer que la résolution ordonnée (pour $\succ_1$) avec sélection, élimination de de tautologies et de clauses subsumées linéairement, et splitting, termine sur des ensembles de clauses qui sont des mises sous forme clausale de formules de la classe monadique.

La section suivante présente les automates d'arbres comme des cas particuliers de clauses de Horn, dont on se rendra compte qu'elles présentent des similitudes avec les clauses que nous venons de présenter, dans le cas particulier où elles sont de Horn.

1.7 Clauses de Horn et automates d'arbres

1.7.1 Langages et reconnaissabilité

Les ensembles de clauses de Horn définissent des *langages de termes*, ce qui n'est pas sans rappeler la définition de langages reconnus par automates. Nous allons voir en effet que les automates d'arbres sont des cas particuliers d'ensembles de clauses de Horn.

Précisons que dans le cas des clauses de Horn, les structures de Herbrand peuvent être ordonnées par inclusion $\subseteq$. Il est alors établi que tout ensemble satisfiable de clauses de Horn a un *plus petit modèle de Herbrand*.

Soit S un ensemble satisfiable de clauses de Horn, et P un symbole de prédicat. Le langage $L_P(S)$ de S à l'*état* P est l'ensemble des termes clos t tels que $P(t)$ est dans le plus petit modèle de Herbrand de S. Les éléments de $L_P(S)$ sont appelés les termes *reconnus* à P de S.

Par abus de langage, on dira que P est *vide* dans S si et seulement si $L_P(S)$ est vide, et de même pour toute autre propriété.

Le lemme suivant caractérise la reconnaissabilité.

Lemme – 1.7.1

> *Soit S un ensemble satisfiable de clauses de Horn. Le terme clos t est reconnu à P si et seulement si $S \cup \{\perp \Leftarrow P(t)\}$ est insatisfiable.*

Le deuxième lemme caractérise la vacuité.

Lemme – 1.7.2

> *Étant donné un ensemble S satisfiable de clauses de Horn, P est vide dans S si et seulement si $S \cup \{\perp \Leftarrow P(x)\}$ est satisfiable.*

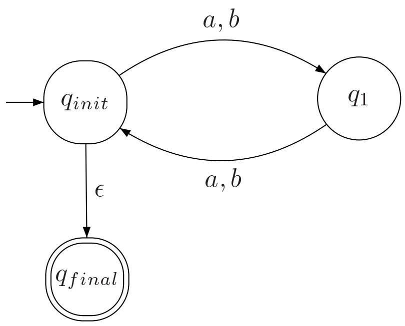
Le troisième lemme caractérise la *vacuité d'intersection*, c'est-à-dire, étant donné un nombre fini de symboles de prédicats $P_1, \ldots, P_n$, la question de la vacuité de $L_{P_1}(S) \cap \ldots \cap L_{P_n}(S)$ (on dira par abus de langage, que l'intersection de $P_1, \ldots, P_n$ est vide dans ce cas).

Lemme – 1.7.3

> *Étant donné un ensemble S satisfiable de clauses de Horn, l'intersection de $P_1, \ldots, P_n$ est vide dans S si et seulement si $S \cup \{\perp \Leftarrow P_1(x), \ldots, P_n(x)\}$ est satisfiable.*

1.7.2 Automates d'arbres

Considérons par exemple l'automate des mots formés d'un nombre pair de lettres. L'état initial est q_{init}, et le seul l'état final est q_{final}. Cet automate reconnaît tous les mots de longueur paire sur l'alphabet $\{a, b\}$.

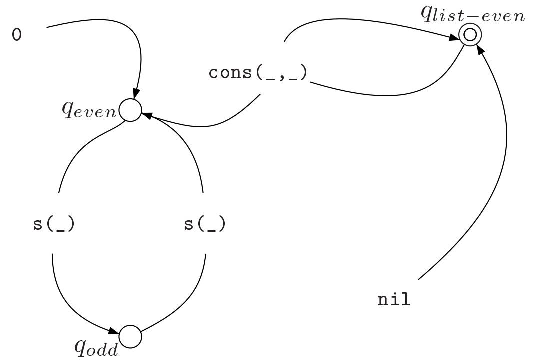


On peut exactement décrire l'ensemble des mots reconnus à chaque état q, sous forme de clauses de Horn, où $q(t)$ signifie que t est reconnu en q :

$$\begin{array}{rcl}
q_{init}(\epsilon) & & \\
q_{final}(x) & \Leftarrow & q_{init}(x) \\
q_1(a(X)) & \Leftarrow & q_{init}(X) \\
q_1(b(X)) & \Leftarrow & q_{init}(X) \\
q_{init}(a(X)) & \Leftarrow & q_1(X) \\
q_{init}(b(X)) & \Leftarrow & q_1(X)
\end{array}$$

Tout mot u est ici codé sous la forme du terme clos $u(\epsilon)$, où ϵ est une constante : si u est le mot vide, alors $u(t) = t$; si u est un mot de la forme va où a est une lettre, alors $u(t) = a(v(t))$.

On peut en général définir des automates d'*arbres*, qui sont des automates finis permettant de reconnaître des termes clos. L'automate d'arbres ci-contre, par exemple, reconnaît (en $q_{list-even}$) l'ensemble de toutes les listes d'entiers pairs. Comme plus haut, chaque transition donnera lieu à exactement une clause :



$$\begin{array}{rcl}
q_{even}(\mathtt{0}) & & \\
q_{list-even}(\mathtt{nil}) & & \\
q_{even}(\mathtt{s}(X)) & \Leftarrow & q_{odd}(X) \\
q_{odd}(\mathtt{s}(X)) & \Leftarrow & q_{even}(X) \\
q_{list-even}(\mathtt{cons}(X, Y)) & \Leftarrow & q_{even}(X), q_{list-even}(Y)
\end{array}$$

On peut remarquer que l'on n'a pas besoin de définir d'états initiaux dans un automate d'arbres : $\mathtt{0}$ par exemple est reconnu en q_{even}, en utilisant la transition $q_{even}(\mathtt{0})$, d'arité 0.

De manière générale, on appellera *automate d'arbres* tout ensemble fini de clauses de Horn de la forme :

$$P(f(X_1, \ldots, X_n)) \Leftarrow P_1(X_1), \ldots, P_n(X_n) \tag{1.1}$$

où les variables $X_1, \ldots, X_n$ sont distinctes deux à deux. Lorsque $n = 0$, on retrouve les clauses de la forme $q_{list-even}(\mathtt{nil})$.

Un langage de termes est dit *régulier* si et seulement s'il est de la forme $L_P(S)$, pour un prédicat P et un automate d'arbres S.

Test d'appartenance. On peut tester si un terme clos t est dans $L_P(S)$ en testant si S plus la clause $\bot \Leftarrow P(t)$ est insatisfiable, par le lemme 1.7.1. Soit $\rhd$ l'ordre *sous-terme*, c'est-à-dire le plus petit ordre strict tel que $f(t_1, \ldots, t_n) \rhd t_i$ pour tout i, $1 \leq i \leq n$, et tout f. Utilisons l'ordre $\succ_1$ sur les atomes défini par $P(t) \succ_1 P'(t')$ si et seulement si $t \rhd t'$. On voit alors que, sur toute branche du tableau obtenu par résolution ordonnée (avec $\succ_1$) et splitting, on ne générera que des clauses de la forme $\bot \Leftarrow Q(u)$, où Q est un symbole de prédicat et u un sous-terme de t. Chaque branche est donc de longueur polynomiale en la taille de t et S, et le problème est donc dans coNP. En réalité, le problème est même dans P : on peut extraire de toute dérivation de la clause vide par hyperrésolution *positive* une dérivation par hyperrésolution positive ne dérivant que des clauses unitaires de la forme $P(u)$, où u est un sous-terme de t ; de plus, si u est de la forme $f(u_1, \ldots, u_n)$, alors $P(u)$ est dérivé par une instance d'une clause (1.1) de la forme

$$P(f(u_1, \ldots, u_n)) \Leftarrow P_1(u_1), \ldots, P_n(u_n). \tag{1.2}$$

Il n'y a qu'un nombre polynomial de telles instances, lorsque u parcourt les sous-termes de t. De plus, ces instances sont closes, donc l'ensemble de ces instances est en fait un ensemble de clauses de Horn propositionnelles. Ceci reste vrai lorsque l'on rajoute la clause $\bot \Leftarrow P(t)$. Or, l'hyperrésolution positive sur des clauses de Horn propositionnelles termine en temps polynomial. Donc, l'appartenance d'un terme clos à $L_P(S)$ est décidable en temps polynomial.

Test du vide. On peut aussi tester si $L_P(S)$ est vide par résolution ordonnée et splitting, en utilisant le lemme 1.7.2. On ne génère que des clauses de la forme $Q(X)$, donc le problème est aussi dans coNP. Cependant, le problème de la vacuité est lui aussi dans P. En effet $L_P(S)$ est vide si et seulement si l'ensemble des clauses obtenues à partir de S en oubliant les arguments des symboles des prédicats (c'est-à-dire en passant de $Q(t)$ à Q) plus la clause $\perp \Leftarrow P$ est insatisfiable. On obtient encore ainsi un ensemble de clauses de Horn propositionnelles. Leur insatisfiabilité est donc décidable par hyperrésolution positive en temps polynomial.

1.7.3 Automates bidirectionnels alternants

On peut maintenant considérer des automates d'arbres enrichis, n'utilisant pas seulement les clauses (1.1), mais aussi d'autres formes de clauses :

ϵ-transitions. Un premier ajout est celui des *ϵ-transitions* :

$$P(X) \Leftarrow P'(X) \tag{1.3}$$

Ces clauses expriment simplement le fait que tout terme reconnu en P' est aussi reconnu en P.

Un ensemble de clauses de forme (1.1) et (1.3) est appelé un *automate d'arbres avec ϵ-transitions*. Les résultats de décidabilité et de complexité des tests d'appartenance et du vide tiennent toujours en présence d'ϵ-transitions ; les modifications à apporter aux arguments des preuves sont mineures.

Alternance. Un autre ajout est de considérer les *clauses intersection* :

$$P(X) \Leftarrow P_1(X), \ldots, P_n(X) \tag{1.4}$$

où $n \geq 1$. Les cas des ϵ-transitions est donc un cas particulier de clauses intersection.

Un ensemble de clauses de forme (1.1) et (1.4) est appelé un *automate d'arbres alternant*. La raison intuitive est que, pour reconnaître un terme clos t en P, on doit d'abord trouver (par un choix existentiel) une clause qui permet de dériver $P(t)$, et si cette clause est une clause intersection (1.4), on doit vérifier que t est reconnu en tout état P_i, $1 \leq i \leq n$ (c'est un choix universel). On reconnaît donc t en alternant des choix existentiels et universels.

Automates bidirectionnels. On peut enrichir les automates d'arbres avec des clauses de la forme :

$$P(X_i) \Leftarrow Q(f(X_1, \ldots, X_n)) \tag{1.5}$$

où $1 \leq i \leq n$ et les variables $X_1, \ldots, X_n$ sont distinctes deux à deux. Alors que les clauses (1.1) reconnaissent un terme $f(t_1, \ldots, t_n)$ à un état P en le décomposant, et en essayant ensuite de reconnaître t_1 en P_1, ..., t_n en P_n (en lisant la clause (1.1) de gauche à droite), les clauses (1.5) tentent de reconnaître un terme t en P en devinant un terme plus gros $f(t_1, \ldots, t_{i-1}, t, t_{i+1}, \ldots, t_n)$ reconnu en Q.

Nous verrons qu'en vérification de protocoles cryptographiques (sections 2.4 et 6.1), nous aurons besoin de clauses qui sont des extensions du format (1.5) :

$$P(X_i) \Leftarrow Q(f(X_1, \ldots, X_n), Q_1(X_{i_1}), \ldots, Q_k(X_{i_k}) \tag{1.6}$$

où $k \geq 0$, $1 \leq i_1, \ldots, i_k \leq n$ et les variables $X_1, \ldots, X_n$ sont distinctes deux à deux.

Un ensemble de clauses (1.1), (1.3) et (1.5) est appelé un *automate d'arbres bidirectionnel*. Un ensemble de clauses (1.1), (1.4) et (1.5) est appelé un *automate d'arbre bidirectionnel alternant*.

Un ensemble de clauses (1.1), (1.3) et (1.6) est appelé un *automate d'arbre bidirectionnel étendu*. Un ensemble de clauses de la forme (1.1), (1.4) et (1.6) est appelé un *automate d'arbre bidirectionnel alternant étendu*.

Remarque 1 *Un automate d'arbres bidirectionnel alternant étendu a en fait le même pouvoir expressif qu'un automate d'arbres bidirectionnel alternant non étendu. Il suffit de recoder la clause de format (1.6) avec les clauses suivantes :*

$$
\begin{aligned}
P(X_i) &\Leftarrow Q_{aux}(f(X_1, \ldots, X_n) \\
Q_{aux}(X) &\Leftarrow Q(X), Q'_{aux}(X) \\
Q'_{aux}(f(X_1, \ldots, X_n) &\Leftarrow Q_1(X_{i_1}), \ldots, Q_k(X_{i_k})
\end{aligned}
$$

où Q_{aux} et Q'_{aux} sont des symboles de prédicats frais.

Chapitre 2

Les protocoles cryptographiques

La sécurité des systèmes d'information est aujourd'hui un domaine en pleine expansion, ces systèmes devenant pratiquement omniprésents dans notre vie, et leur consommation se présentant à nous de manière de plus en plus indispensable. Parmi les aspects de la sécurité informatique, la cryptologie et les protocoles cryptographiques ont pris une importance considérable.

Ceci n'a bien sûr pas toujours été le cas, et pourtant la cryptologie est une science très ancienne, longtemps confinée à des applications militaires et diplomatiques, comme nous le verrons en section 2.1, mais qui répond aujourd'hui aux besoins du marché et constitue un domaine scientifique en pleine activité. Elle intervient dans de multiples applications, notamment celles liées au commerce électronique et au réseau internet.

Comment transmettre des données personnelles sans qu'elles soient lues par un tiers indésirable ? Comment être sûr que les données reçues sont bien celles qui ont été envoyées ? Comment être sûr de l'identité du correspondant ? Autant de questions, et d'autres encore, correspondant à des propriétés sur les protocoles cryptographiques, dont on souhaite une réponse sûre (section 2.2). C'est pourquoi beaucoup de modèles ont été construits afin de pouvoir se doter de systèmes de preuves permettant le mieux possible de démontrer les propriétés auxquelles sont destinées les différents protocoles. Nous nous attacherons plus particulièrement aux modèles à la Dolev-Yao et aux méthodes de preuves associées (section 2.3). Pour finir, nous présenterons trois modèles qui sont des extensions de Dolev-Yao, et qui nous intéressent plus particulièrement car ils utilisent un langage de clauses.

2.1 Un peu d'histoire

Les premières expressions de cryptologie se retrouvent dès l'Antiquité dans les documents hiéroglyphiques égyptiens ainsi que dans la Bible où elle est un élément constitutif d'un savoir initiatique sacré et des sciences occultes. Mais, dès cette période, Grecs et Romains, sachant y voir aussi un mode de communication fonctionnel, l'ont utilisée à des fins militaires et diplomatiques.

Au cours de la Renaissance, qui fut une longue période de redécouverte de l'Antiquité, les monarques européens commencent à s'entourer de cryptologues, comme en France, Henri IV avec le mathématicien Francis Viète l'un des pères de l'algèbre moderne.

A partir du 19ème siècle, la vocation militaire de la cryptologie (le *chiffre*) tend à dominer comme en témoigne la publication en 1883 de [Ker83] du linguiste August Kerckhoffs qui mit au jour les exigences requises en termes de sécurité, facilité et rapidité d'un système cryptographique dans le domaine militaire. Il s'agit de mettre au point le mécanisme de chiffrement, permettant le passage du texte clair au texte chiffré, inviolable s'il devait tomber aux mains de l'ennemi. La clef permet le déchiffrement et la restitution du texte clair et assure de même la sécurité du chiffrement. Toutefois, le décryptage, qui permet d'établir le texte clair sans aide de la clef, doit être en revanche matériellement voire mathématiquement impossible.

Des machines de cryptologie de plus en plus sophistiquées, alliant mécanique et électromagnétisme, sont conçues entre les deux guerres mondiales telles que l'Enigma allemande. Au cours de la Seconde Guerre mondiale, les Alliés constituent une équipe de chercheurs chargés de décrypter les chiffres allemands, et qui en 1941 parvient à lire les messages d'Enigma. Alan Turing, membre de cette équipe, fut à l'origine de la fabrication par les Britanniques de la machine Colossus, sorte de proto ordinateur car dotée de composantes électroniques, qui remporta plusieurs succès décisifs contre les machines allemandes.

Les Forces Alliées disposaient ainsi au sortir du conflit d'un savoir-faire en cryptologie aussi élaboré que secret n'autorisant l'accès qu'aux travaux les plus théoriques. La National Security Agency (N.S.A.), chargée du contrôle rigoureux de l'usage de la cryptologie, fut établie par le gouvernement américain en 1952 engagé désormais dans la Guerre froide.

Vingt ans plus tard, c'est sous l'effet du lobbying des milieux financiers et bancaires, que la NSA accepta d'étendre l'usage de la cryptographie au domaine civil débouchant au terme d'un long processus d'élaboration sur la création, en 1976, d'une norme commerciale de déchiffrement le Data Encryption Standard (D.E.S.).

A partir de cette date, la cryptologie devient un domaine de recherche ouvert en particulier grâce à la notion de *clef publique* (voir section 2.2) inventée par Whitfield Diffie et Martin Hellman ([DH76]), et la publication en 1978 d'un procédé de déchiffrement élaboré sur les travaux de Diffie et Hellman, le RSA [ARS78], du nom de leurs auteurs, Ronald Rivest, Adi Shamir et Leornard Adleman.

Parmi les différentes publications consacrées à ce thème, on pourra notamment se référer à [Ste98, Sch94] ; ainsi que consulter [enc00] dont est tirée cette brève histoire pour de plus amples détails.

2.2 Les notions principales

Nous allons ici définir quelques notions couramment employées par la suite, notamment des notions que nous avons d'ailleurs déjà abordé comme le chiffrement par exemple (section 2.2.1). Nous parlerons ensuite de propriétés de sécurité (section 2.2.2), et enfin de la notion d'attaques sur un protocole (section 2.2.3).

2.2.1 Définitions

La *cryptologie* se partage entre la *cryptographie* qui étudie les mécanismes destinés à assurer la confidentialité, et la *cryptanalyse* dont le but est de déjouer la protection des moyens ainsi mis en place. Le *chiffrement* est le procédé employé pour passer du message clair au message chiffré ou *cryptogramme*, et la *clef de chiffrement* est ce qui permet d'effectuer ce calcul. Un exemple très simple, connu sous le nom de "chiffre de César", est celui qui consiste à remplacer chaque lettre par celle située deux positions plus loin dans l'alphabet.

Le *déchiffrement* est l'opération qui consiste à passer du message chiffré au message en clair à l'aide d'un moyen prévu à cet effet, qui dans l'exemple du chiffre de César sera de remplacer chaque lettre par celle située deux positions avant dans l'alphabet. La *clef de déchiffrement* est le pendant de la clef de chiffrement.

Le *décryptage* est le fait de réussir à trouver le message clair à partir du message codé sans connaître au préalable le moyen de le déchiffrer.

Il existe deux procédés de chiffrement : le chiffrement à *clef symétrique* où la même clef sert à chiffrer et à déchiffrer les messages, et le chiffrement à *clef asymétrique* où une clef sert à chiffrer les messages, la *clef publique* qui n'est pas une clef secrète, et une autre sert à déchiffrer, la *clef privée*. Ainsi, chaque interlocuteur possède une paire de clefs, sa clef privée restituant les messages qui ont été chiffrés avec sa clef publique.

Le *protocole cryptographique* est la procédure permettant l'échange de messages.

2.2.2 Propriétés de sécurité

Comme nous l'avons dit précédemment, la cryptographie a longtemps servi à garantir le secret, mais les protocoles cryptographiques n'ont pas tous cet objectif. Nous allons donner un aperçu des objectifs les plus courants auxquels les différents protocoles sont dédiés :

- Le *secret*, comme nous l'avons déjà mentionné, consiste à garder inconnue pour une tierce personne une certaine donnée ; par exemple, pour le paiement par carte à l'aide de terminaux portables, l'utilisateur doit confirmer son identité en entrant son code PIN à quatre chiffre qui est une donnée qui doit restée secrète.
- L'*authentification* est mal définie, dans la mesure où il existe plusieurs définitions de cette notion. On peut cependant considérer que bon nombre d'entre elles partent de la même idée : l'authentification consiste à être certain de correspondre avec l'interlocuteur escompté (il serait par exemple ennuyeux d'envoyer son numéro de carte bancaire à quelqu'un qui se fait passer pour le vendeur). Les différences apparaissent dans les diverses définitions formelles qui en sont données ; on peut par exemple consulter [Low97].
- L'*anonymat* est le fait de ne pas être identifiable parmi un ensemble de personnes. Par exemple, lors d'un vote, on ne doit pas pouvoir établir de lien entre une personne et le contenu de son bulletin, ceci ne pouvant être garanti que s'il y a au moins deux votes différents pour l'élection.
- L'*intégrité* signifie usuellement que les données ne peuvent pas être corrompues.
- La *non-répudiation* consiste à éviter que des participants qui envoient et reçoivent des messages puissent le nier plus tard ; on peut prendre l'exemple du vote électronique où la non-répudiation assure que la personne qui a déjà voté ne puisse pas prétendre le contraire.

Il existe cependant pour chacune de ces propriétés des définitions formelles différentes, comme nous l'avons mentionné pour l'authenticité, mais nous n'entrerons pas dans ces détails.

2.2.3 Attaques

Un but recherché à travers beaucoup de protocoles cryptographiques est la garantie de secret, à laquelle nous allons plus particulièrement nous intéresser ici. En particulier, on peut se poser la question suivante : étant donnés deux principaux A et B (c'est-à-dire deux parties du protocole qui peuvent être des personnes, des cartes à puces, des serveurs etc.), comment vont ils pouvoir s'échanger une clef secrète K_{AB} pour communiquer de manière sécurisée ? On peut imaginer que A par exemple envoie tout simplement à B la clef K_{AB}. Cela suppose qu'ils disposent d'un canal de communication suffisamment sûr, mais en fait il est très facile pour une personne malveillante, que nous appellerons *intrus*, d'espionner les canaux, et même d'intercepter des messages. Il s'agit donc de trouver un moyen d'envoyer la clef K_{AB} à B sans qu'elle ne passe en clair sur le réseau, et de telle manière évidemment que B puisse la lire pour s'en servir. On peut alors se servir d'une clef dite *à long terme* partagée entre A et B qui est une clef très robuste, typiquement établie avec un algorithme tel que RSA [Sch94]. On distinguera alors les clefs à long terme K_{long} et les clefs de session telle que K_{AB} qui ont vocation à n'être utilisées que pour une seule exécution du protocole. A pourrait donc envoyer à B la clef K_{AB} chiffrée avec K_{long}, ce que nous écrirons dans un formalisme très couramment employé pour décrire les processus qui se contente de décrire les échanges de messages entre les principaux en présence :

$$A \rightarrow B : \{K_{AB}\}_{K_{long}}$$

où $\{K_{AB}\}_{K_{long}}$ désigne le chiffrement de K_{AB} avec K_{long}.

Supposons alors que l'intrus ait réussi à se procurer K_{old}, une vieille version de K_{AB} provenant d'une ancienne exécution du protocole (pas forcément avec les mêmes participants A et B), soit parce que la clef n'est pas très robuste, soit parce qu'un des participants l'a involontairement divulguée après son utilisation. Lors de la session de ce protocole, l'intrus a pu intercepter le message $\{K_{old}\}_{K_{long}}$. Il peut alors se faire passer pour A, et renvoyer ce même message à B :

$$I(A) \rightarrow B : \{K_{old}\}_{K_{long}}$$

où $I(A)$ est employé pour signifier que l'intrus I a pris le rôle de A. B va alors engager une communication avec A en utilisant la clef K_{old} qu'il croit secrète. L'intrus pourra alors déchiffrer les messages circulant entre A et B avec la clef K_{old}. Cette attaque est appelée une attaque par *rejeu*.

Pour déjouer ces attaques, on veut pouvoir s'assurer que le message reçu n'est pas un ancien message. La solution couramment envisagée est d'envoyer conjointement à K_{AB} une donnée qui garantisse sa fraîcheur. Un moyen proposé, par exemple dans le protocole de Denning et Sacco [DS81], est d'utiliser des estampilles temporelles (*timestamp*). A envoie à B le message $\{K_{AB}, T\}_{K_{long}}$ où T est calculée à partir de l'horloge interne du système utilisé par A. B vérifiera alors, après avoir déchiffré le message que la deuxième composante est suffisamment récente pour assurer une utilisation sans risque de la clef. Mais cela pose deux problèmes : d'une part celui de la synchronisation des deux horloges internes de A et de B, et d'autre part celui du "suffisamment récent" qui est imprécis. L'autre moyen employé dans beaucoup de protocoles est l'utilisation de *nonces*, qui sont des nombres générés aléatoirement par les principaux garantissant si ce n'est de manière complètement sûre mais tout du moins très acceptable, qu'une telle donnée n'a pas déjà été utilisée au cours d'une ou plusieurs sessions précédentes du protocole. Mais l'utilisation des nonces, même si à l'évidence ils servent à déjouer certaines attaques, n'empêche pas forcément les attaques par rejeu qui sont parfois très subtiles. Le très célèbre protocole de Needham et Schroeder à clefs publiques [NS78], pour lequel les données de A et B sont modélisées à l'aide de nonces, a été connu pendant 17 ans avant qu'une attaque ait été trouvée [Low95] ! Nous allons ici décrire ce protocole, ce qui nous donnera l'occasion de présenter un autre type d'attaque, connue sous le nom de *man-in-the-middle*, où l'intrus s'impose dans la communication entre les participants. Nous présentons en figure 2.1 une version simplifiée du protocole (nous ne retenons en fait que les échanges de messages qui sont utilisés dans l'attaque). Dans ce protocole, N_a et N_b désignent les

$$\begin{array}{lrcl} 1. & A \to B & : & \{N_a, A\}_{K_b} \\ 2. & B \to A & : & \{N_a, N_b\}_{K_a} \\ 3. & A \to B & : & \{N_b\}_{K_b} \end{array}$$

Fig. 2.1: Needham-Schroeder à clefs publiques (version simplifiée)

nonces créés respectivement par A et B. Chaque participant p possède une clef publique connue des autres participants, disons $PK(p)$, et une clef secrète connue de lui seul comme son nom le laisse à penser, $SK(p)$. Ces deux clefs sont telles que, si l'on note $decrypt(M, K)$ le déchiffrement de M avec la clef K, pour tout participant p et pour tout message M, $decrypt(\{M\}_{PK(p)}, SK(p)) = M$. Les clefs publiques de A et de B sont ici notées respectivement K_a et K_b.

À la première étape, A crée un nonce N_a et envoie le message composé de ce nonce et de son identité, le tout chiffré avec la clef publique de B. Quand B reçoit le message, il le déchiffre avec sa clef privée, et il est le seul à pouvoir le faire (dans la mesure où sa clef est bien secrète). Il dispose ainsi de l'identité A de son interlocuteur et d'un nonce N_a. À la deuxième étape, B crée un nonce N_b et envoie à A le message composé de N_a et de N_b chiffré avec la clef publique de A. Lorsque A reçoit le message, il la déchiffre avec sa clef secrète, et peut ainsi vérifier que la valeur de la première composante du message correspond bien au nonce N_a qu'il avait envoyé précédemment à B. La troisième étape consiste pour A à renvoyer le nonce N_b chiffré avec la clef publique de B ; celui-ci déchiffre le message avec sa clef privée et peut vérifier que la valeur correspond au nonce N_b.

À la fin du protocole, les participants sont censés être sûrs de l'identité de leur interlocuteur, être d'accord sur les valeurs N_a et N_b, et être sûrs que ces valeurs sont connues d'eux seuls.

Pourtant, l'attaque découverte par G. Lowe ([Low95]) nous montre une vulnérabilité du protocole. L'attaque utilise deux sessions simultanées du protocole : lors de la session 1, A établit une session normale avec l'intrus, noté I, et lors de la session 2, l'intrus prend le rôle de A pour établir une fausse session avec B, comme le présente la figure 2.2. On écrit $I(A)$ pour signifier que I joue

$$\begin{array}{lll} 1.1. & A \to I & : \{N_a, A\}_{K_i} \\ 2.1. & I(A) \to B & : \{N_a, A\}_{K_b} \\ 2.2. & B \to I(A) & : \{N_a, N_b\}_{K_a} \\ 1.2. & I \to A & : \{N_a, N_b\}_{K_a} \\ 1.3. & A \to I & : \{N_b\}_{K_i} \\ 2.3. & I(A) \to B & : \{N_b\}_{K_b} \end{array}$$

Fig. 2.2: Une attaque contre Needham-Schroeder à clefs publiques

le rôle de A, et 2.1, par exemple, pour représenter le message 1 dans la session 2. À l'étape 1.1, A débute une session du protocole avec I (il envoie donc un nonce nouvellement créé et son identité chiffrés avec la clef publique de I). L'intrus déchiffre le message avec sa clef secrète, obtenant ainsi N_a et A qu'il va utiliser, à l'étape 2.1, pour prendre le rôle de A et initier une fausse session avec B ; il envoie donc à B les deux données N_a et A chiffrées avec la clef publique de B, que celui-ci va déchiffrer avec sa clef secrète. À l'étape 2.2, B, croyant qu'il communique avec A, envoie le message correspondant à l'étape 2 du protocole chiffré avec la clef publique de A que l'intrus va intercepter. À l'étape 1.2, l'intrus envoie à A le message qu'il vient de recevoir de B, et ainsi, A va pouvoir vérifier, après l'avoir déchiffré avec sa clef secrète, que la première valeur de ce message correspond bien au nonce N_a qu'il a envoyé à l'étape 1.1. Il envoie donc à I, à l'étape 1.3, le nonce N_b chiffré avec la clef publique de I, que celui-ci va pouvoir obtenir. Et pour finir, à l'étape 2.3, I, jouant le rôle de A dans l'échange avec B, envoie le nonce N_b chiffré avec la clef publique de B, faisant croire à celui-ci que le protocole s'est déroulé normalement avec A. Ainsi, A et B pensent qu'ils ont communiqué ensemble et qu'ils partagent à eux seuls les nonces N_a et N_b, alors qu'ils sont connus d'une tierce personne.

Au passage, on peut noter que la découverte de cette attaque est accompagnée d'une modification du protocole qui permet de la prévenir. Il suffit d'inclure l'identité de B dans le message de l'étape 2 :

$$2. \quad B \to A \quad : \quad \{B, N_a, N_b\}_{K_a}$$

L'étape 2.2 de l'attaque devient alors :

$$2.2. \quad B \to I(A) \quad : \quad \{B, N_a, N_b\}_{K_a}$$

et l'intrus ne pourra pas rejouer le message à l'étape 1.2 de manière concluante. Ce protocole ainsi modifié est maintenant connu sous le nom de Needham-Schroeder-Lowe.

La découverte de telles attaques, notamment celle du protocole de Needham et Schroeder à clefs publiques, a poussé la communauté scientifique à développer des méthodes formelles et systématiques pour la vérification des protocoles cryptographiques ou pour la recherche d'attaques.

2.3 Modèles et méthodes

Il existe maintenant beaucoup de modèles et de méthodes développés pour détecter des attaques ou pour établir des preuves formelles de sécurité sur les protocoles cryptographiques. Parmi les modèles, ceux inspirés de Dolev-Yao sont très nombreux, et retiendrons notre attention dans la majeur partie de cette section. Cependant, d'autres modèles, comme le spi-calcul, basés sur les algèbres de processus, sont intéressants pour leur rigueur et leur précision, et nous en donnerons une brève présentation en section 2.3.1. Nous présenterons ensuite (section 2.3.2) les principes fondamentaux des modèles à la Dolev-Yao (section 2.3.2.1), ainsi que différentes méthodes de preuve et de vérification (section 2.3.2.2). En effet, les preuves de sécurité se heurtant à des problèmes d'indécidabilité de mieux en mieux établis, le mieux à espérer est soit d'identifier des sous-classes décidables, soit de développer des algorithmes de vérification corrects mais incomplets.

2.3.1 Le spi-calcul

Parmi la catégorie importante de modèles que sont les algèbres de processus, nous pouvons citer le spi-calcul [AG97], qui est une extension du pi-calcul [MPW92] aux primitives cryptographiques. Un aspect intéressant est la définition très précise du secret et de l'authentification qui ne dépend pas d'un système déductif décrivant les capacités de l'intrus. En effet, l'intrus est modélisé simplement comme un environnement arbitraire capable de procéder à tous les tests définissables dans le langage pour essayer de distinguer différentes instantiations d'un même protocole. En particulier, la génération de nonces et de clefs de session étant modélisée grâce à l'opérateur ν de création de nom du pi-calcul, l'intrus a aussi la capacité d'en créer. L'avantage ainsi de ne pas modéliser explicitement les capacités de l'intrus est d'éviter le risque d'omettre certains de ses comportements, dans les limites évidemment de l'expressivité du langage. En général, dans beaucoup de modèles, une donnée est secrète si l'intrus ne peut pas l'obtenir, mais en spi-calcul la propriété de secret est plus forte. Le secret d'un message M est défini comme l'impossibilité pour un environnement quelconque, donc pour l'intrus, de pouvoir dégager une différence entre deux instances d'un même protocole où le message M a été remplacé par un autre message M'. Cette différence est exprimée à l'aide d'une équivalence, la may-testing équivalence, la difficulté des preuves résidant dans la quantification universelle sur les environnements. Il existe des notions de bisimulations entre les processus qui caractérisent ou approximent la may-testing equivalence [BN02], qui n'utilise plus de quantification universelle sur les environnements, mais la décidabilité de telles bisimulations reste difficile à dégager.

2.3.2 Les modèles à la Dolev-Yao

Un modèle simple qui a marqué une étape importante de l'étude des protocoles cryptographiques est dû à Dolev et Yao [DY83]. Il a de nombreuses variantes et a servi de support à de nombreuses méthodes de vérification [AL00, Bol96, Bol97a, Bol97b, CKR+03, GK99, GL00, Kha01, Mon99, Pau98]. Une des caractéristiques importantes introduites par Dolev et Yao est l'hypothèse du *chiffrement parfait* : un message chiffré ne peut pas être décrypté, il ne peut être déchiffré qu'avec la clef appropriée. Même si l'on considère qu'on dispose depuis les années 80 d'algorithmes cryptographiques suffisamment sûrs, cette hypothèse est malgré tout assez forte. Certains modèles parmi ceux inspirés de Dolev et Yao essaient cependant de s'en dégager [CKRT03, CLC03a, CLS03, Ver03]. Une autre caractéristique importante de ces modèles est la modélisation de l'intrus qui a un contrôle total sur les communications et qui est capable de construire des messages. Nous y reviendrons plus en détail en section 2.4.

2.3.2.1 Principes fondamentaux

On peut pointer trois apports fondamentaux de ce modèle.

En premier lieu, le mode de communication entre les principaux a été très simplifié par la quasi omniprésence de l'intrus. Il n'est plus besoin de considérer des canaux de communication et de définir leurs propriétés, il suffit de supposer que l'intrus est très puissant et qu'il possède à lui seul l'ensemble des canaux de communication. L'intrus pouvant ainsi intercepter et espionner tout ce qui passe sur les canaux, tout message qui transite peut être récupéré et renvoyé sur le réseau. Le modèle de communication est ainsi beaucoup plus simple : tout message envoyé sera envoyé à l'intrus, et tout message reçu le sera de l'intrus.

En second lieu, Dolev et Yao ne représentent plus les messages par des suites de bits ou des entiers, mais les modélisent comme une algèbre de termes munie ou non d'une théorie équationnelle.

Nous présentons ici un modèle qui donne une idée générale de ce que peuvent être les modèles à la Dolev-Yao. Voici une définition des termes :

$$M ::= D \mid K \mid pair(M_1, M_2) \mid p_1(M) \mid p_2(M) \mid encrypt(M, K) \mid decrypt(M, K)$$

où D parcourt un ensemble un ensemble de données de base (réels, entiers, ...), K parcours un ensemble de clefs, $pair(M_1, M_2)$ est la paire formée des messages M_1 et M_2, $p_1(M)$ est la première composante de la paire M, $p_2(M)$ est la deuxième composante de la paire M, $encrypt(M, K)$ est le chiffrement du message M avec la clef K, $decrypt(M, K)$ est le déchiffrement du message chiffré M avec la clef K.

Ces termes sont à comprendre modulo la théorie équationnelle suivante :

$$\begin{array}{rcl} p_1(pair(M_1, M_2)) & = & M_1 \\ p_2(pair(M_1, M_2)) & = & M_2 \\ decrypt(encrypt(M, K), K) & = & M \end{array}$$

En effet, p_1 et p_2 ne peuvent décomposer que des messages qui sont des paires et seul un message chiffré avec la clef K peut être déchiffré avec K (c'est le cas du chiffrement symétrique).

En troisième et dernier lieu, l'intrus est représenté par un système déductif : il peut déduire des messages à partir de ceux qu'il intercepte, c'est-à-dire à partir de tout message circulant entre les principaux. Afin de décrire ce modèle déductif, notons $E \vdash M$ la relation signifiant "l'intrus peut déduire le message M à partir de l'ensemble E", ensemble représentant la totalité des messages déjà déduits.

$$\frac{}{E, M \vdash M} \qquad \frac{E \vdash M \quad E \vdash K}{E \vdash encrypt(M, K)} \qquad \frac{E \vdash M_1 \quad E \vdash M_2}{E \vdash pair(M_1, M_2)} \qquad \frac{E \vdash M}{E \vdash p_1(M)}$$

$$\frac{E \vdash M}{E \vdash p_2(M)} \qquad \frac{E \vdash M \quad E \vdash K}{E \vdash decrypt(M, K)} \qquad \frac{E \vdash M \quad M = N}{E \vdash N}$$

La première de ces règles exprime simplement le fait que de l'ensemble E augmenté de M l'intrus peut inférer M (c'est le simple rejeu). Les deux suivantes sont des règles de construction de messages (l'intrus peut chiffrer un message s'il possède une clef, il peut construire une paire), les trois suivantes sont des règles de destruction de messages (l'intrus peut extraire les composantes d'une paire et il peut tenter de déchiffrer un message avec une clef qu'il sait déduire), et la dernière dit que si M peut être déduit, alors tout message égal à M modulo la théorie équationnelle ci-dessus peut être déduit.

L'intrus peut ainsi produire des messages autant de fois et aussi souvent que nécessaire en utilisant ces règles, mais il ne peut en produire d'aucune autre façon.

2.3.2.2 Les méthodes

Les outils de vérification utilisent beaucoup le model-checking (vérification de modèle), à savoir : étant donné un protocole P et une propriété de sécurité ϕ, est-ce que P satisfait ϕ ? Des outils implémentant un tel procédé ont déjà été utilisés avec succès, le plus connu étant certainement l'outil FDR ayant notamment servi à détecter l'attaque sur le protocole Needham-Schroeder à clefs publiques [Low96]. Il est cependant très difficile de prouver la correction d'un protocole, notamment à cause du problème de l'explosion de l'espace des états ; en effet, les traces ayant une longueur non bornée à cause du nombre arbitraire d'instances du protocole (correspondant au nombre de session), l'arbre des traces est infini en profondeur. De plus, selon ce que l'intrus peut envoyer, il est à branchement infini.

De manière générale, le problème de la vérification des protocoles cryptographiques est un problème indécidable. Un résultat important [RT01] a permis de dégager une sous-classe décidable de protocoles, celle où on considère un nombre borné de sessions.

Si l'on souhaite travailler avec un nombre arbitraire de sessions, deux solutions sont couramment envisagées. La première consiste à dégager des sous-classes décidables. Un certains nombres de résultats ont ainsi été prouvés, parmi lesquels on peut citer [NDS99] (il n'y a pas de nonce et la taille des messages est bornée), [AC02] (le nombre de nonces est borné et la copie des messages est très restreinte), [CLC03a] (le nombre de nonces est borné et on ne peut faire qu'une seule copie des messages). La deuxième solution consiste à définir des méthodes incomplètes où l'on suppose qu'il n'y a pas de nonce (des abstraction seront alors utilisées pour les représenter) et que le nombre d'agents considérés est borné, cette dernière abstraction étant correcte [CLC03b]. Ces méthodes permettent une vérification correcte mais incomplète au sens où on peut trouver de fausses attaques [CLC03a, GL00, GK99, LBP03, Mon99]. Toujours en considérant un nombre arbitraire de sessions, des méthodes correctes et complètes, mais non terminantes, ont aussi été développées [Wei99b] et [Bla01] (dans ce dernier, la terminaison a été prouvé dans un cas typé [BP03]).

Il existe aussi des méthodes où les protocoles sont modélisés dans la logique du premier ordre [Bla01, Bol97a, Wei99b], ce qui permet de ramener le problème de la satisfiabilité d'une propriété de sécurité à celui de la satisfiabilité d'un ensemble de clause, ce qui est particulièrement intéressant au vu de toutes les méthodes de preuves déjà bien connues dans cette logique.

2.4 Extensions de modèles à la Dolev-Yao

Nous exposerons dans cette section quelques extensions de modèles à la Dolev-Yao qui nous intéressent plus particulièrement. Le premier, dû à B. Blanchet [Bla01], est basé sur des règles Prolog (section 2.4.1), le deuxième, dû à H. Comon-Lundh et V. Cortier [CLC03b], est un modèle de traces utilisant aussi des clauses de Horn (section 2.4.2), et enfin le dernier, dû à J. Goubault-Larrecq et N.K. Verma [GLV02], est un modèle représentant les protocoles par un certain type d'automates d'arbres (section 2.4.3), eux-mêmes représentés par des clauses de Horn.

Ces modèles nous intéressent particulièrement puisqu'ils utilisent tous des clauses, qui, comme nous le verrons par la suite (chapitre 3), sont à la base de notre travail.

2.4.1 Le modèle de B. Blanchet

Nous allons décrire ce modèle qui vise à prouver des propriétés de secret sur les protocoles cryptographiques, et nous pouvons d'ores et déjà en pointer quelques caractéristiques :

- Deux approximations sont effectuées :
  - Les valeurs fraîches sont des fonctions d'autres messages précédemment reçus, dans la même session, par le principal qui crée la valeur. Ainsi, elles sont différentes si et seulement si ces messages sont différents. Les mêmes nonces peuvent être réutilisés dans plusieurs sessions du protocole quand les messages précédents sont les mêmes.
  - Le nombre de fois qu'un message apparaît est oublié pour ne se rappeler que s'il est apparu ou non. Ainsi, une étape du protocole peut être exécutée plusieurs fois par session (au lieu d'une seule).
- Le nombre de sessions du protocole n'est pas limité, le nombre de répétitions étant simplement oublié.

Les approximations citées ci-dessus sont effectuées dans le bon sens : si une attaque existe dans un modèle plus précis, alors elle existe aussi dans ce modèle, le prix à payer étant les fausses attaques qui peuvent être trouvées (mais, au dire de l'auteur, l'approximation est suffisamment précise pour que ce cas ne se produise que rarement).

Les protocoles sont représentés par des règles Prolog dont la syntaxe est décrite en figure 2.3. Les termes représentent les messages échangés entre les participants au protocole : les variables, les noms (pour les nonces ou les clefs) considérés comme des fonction de messages précédemment reçus par le principal qui va créer le nom, et l'application de fonctions utilisée pour construire des termes. Les prédicats sont utilisés pour représenter des faits sur les messages, comme le prédicat $\mathsf{attacker}(M)$ beaucoup utilisé par la suite, qui signifie : l'intrus connaît le message M. Une règle $F_1 \wedge \ldots \wedge F_n \rightarrow F$, ou en d'autres termes une clause de Horn, signifie que si tous les faits $F_1, \ldots, F_n$ sont vrais, alors F est vrai.

Les protocoles sont représentés par trois ensembles de règles :

- Un ensemble de règles pour la représentation des primitives cryptographiques, celles qui construisent des termes et celles qui les détruisent.
- Un ensemble de règles représentant les capacités de l'intrus. Celui-ci peut utiliser tous les constructeurs et les destructeurs, il a des connaissances initiales (représentées par des faits).
- Un ensemble de règles pour représenter le protocole lui-même.

$M, N ::=$		termes
	x	variables
	$a[M_1, \ldots, M_n]$	noms
	$f(M_1, \ldots, M_n)$	application de fonction
$F ::=$		fait
	$p(M_1, \ldots, M_n)$	application de prédicat
$R ::=$		règle
	$F_1 \wedge \ldots \wedge F_n \to F$	implication

Fig. 2.3: Syntaxe de la représentation des protocoles

Nous donnons l'exemple en figure 2.4 du protocole Denning-Sacco [DS81] ici simplifié (il utilise normalement des estampilles temporelles et des certificats) présenté dans [Bla01].

Message 1. $A \to B : \{\{k\}_{sk_A}\}_{pk_B}$
Message 2. $B \to A : \{s\}_k$

Fig. 2.4: Protocole simplifié de Denning-Sacco

Le premier message consiste pour A à envoyer à B une clef k chiffrée avec sa clef secrète (on dit aussi que A *signe* la clef k avec sa clef privée sk_A), le tout chiffré avec la clef publique de B. Le second message ne fait pas partie du protocole, mais est ajouté pour vérifier si la clef partagée k peut être utilisée pour échanger des secrets entre A et B. Il existe en fait une attaque à ce protocole [BAN89].

Afin d'illustrer la représentation des protocoles, nous présentons en figure 2.5 la modélisation du protocole de Denning-Sacco dans le modèle de B. Blanchet. Nous retrouvons ici les trois ensembles de règles décrites précédemment.

Capacités de l'intrus :

pencrypt	$\mathsf{attacker}(m), \mathsf{attacker}(pk) \to \mathsf{attacker}(\mathsf{pencrypt}(m, pk))$
pk	$\mathsf{attacker}(\mathsf{pencrypt}(m, \mathsf{pk}(sk))), \mathsf{attacker}(sk) \to \mathsf{attacker}(m)$
sign	$\mathsf{attacker}(m), \mathsf{attacker}(sk) \to \mathsf{attacker}(\mathsf{sign}(m, sk))$
	$\mathsf{attacker}(\mathsf{sign}(m, sk)) \to \mathsf{attacker}(m)$
sencrypt	$\mathsf{attacker}(m), \mathsf{attacker}(k) \to \mathsf{attacker}(\mathsf{sencrypt}(m, k))$
	$\mathsf{attacker}(\mathsf{sencrypt}(m, k)), \mathsf{attacker}(k) \to \mathsf{attacker}(m)$

Connaissances initiales de l'intrus :

$\mathsf{attacker}(\mathsf{pk}(sk_A[])), \quad \mathsf{attacker}(\mathsf{pk}(sk_B[])), \quad \mathsf{attacker}(a[])$

Protocole :

message 1 :	$\mathsf{attacker}(\mathsf{pk}(x)) \to \mathsf{attacker}(\mathsf{pencrypt}(\mathsf{sign}(k[\mathsf{pk}(x)], sk_A[]), \mathsf{pk}(x)))$
message 2 :	$\mathsf{attacker}(\mathsf{pencrypt}(\mathsf{sign}(k', sk_A[]), \mathsf{pk}(sk_B[]))) \to \mathsf{attacker}(\mathsf{sencrypt}(s[], k'))$

Fig. 2.5: Représentation du protocole Denning-Sacco

Chaque règle utilise l'unique symbole de prédicat $\mathsf{attacker}$, l'idée étant que tout message passe par l'intrus (qui est une des caractéristiques importante des modèles à la Dolev-Yao comme nous l'avons souligné en section 2.3.2) ; le fait $\mathsf{attacker}(M)$ signifie, comme nous l'avons mentionné plus haut, que l'intrus connaît le message M.

Le premier ensemble décrivant les capacités de l'intrus, on y voit apparaître les représentations des primitives cryptographiques. La fonction $\mathsf{pencrypt}(m, pk)$ représente le chiffrement à clef publique du message m avec la clef publique k, et la première règle nous dit que l'intrus peut effectuer des chiffrements à clef publique. La fonction $\mathsf{pk}(sk)$ construit la clef publique à partir de la clef privée sk, et la règle correspondante signifie que l'intrus peut déchiffrer, avec une clef secrète, un message qui a été chiffré avec la clef publique correspondante. La fonction $\mathsf{sign}(m, sk)$ représente la *signature* (c'est-à-dire un chiffrement avec une clef secrète) du message m avec la clef secrète sk, la première règle associée signifie que l'intrus peut effectuer des signatures, et la deuxième que l'intrus peut récupérer un message qui a été signé (les clefs publiques étant par définition disponibles à tous, il peut déchiffrer une signature). La fonction $\mathsf{sencrypt}(m, k)$ représente le chiffrement du message m avec la clef symétrique k, la première règle correspondante signifie que l'intrus peut effectuer un tel chiffrement s'il possède un message m et une clef symétrique k, et la deuxième que l'intrus peut déchiffrer $\mathsf{sencrypt}(m, k)$ s'il connaît la clef symétrique k.

Le deuxième ensemble de règles représente les connaissances initiales de l'intrus. Les clefs secrètes $sk_A[]$ de A et $sk_B[]$ de B sont représentées comme des noms ne prenant aucun argument ; ainsi ces clefs sont des constantes. Les deux premières règles, qui sont des faits, signifient que l'intrus connaît les clefs publiques de A et de B. Le nom a dans le fait $\mathsf{attacker}(a[])$ est un nom donné à l'intrus qui représentera tous les noms qu'il peut générer.

Le troisième ensemble de règles décrit les échanges de messages du protocole. Le premier message envoyé par A est le message $\mathsf{pencrypt}(\mathsf{sign}(k[\mathsf{pk}(x)], sk_A[]), \mathsf{pk}(x))$ pour x quelconque. En fait, c'est l'intrus qui initialise le protocole avec le participant de son choix (ici c'est A), en lui envoyant la clef publique de l'autre participant ($\mathsf{pk}(x)$, pour un x quelconque). Le participant choisi envoie alors son message que l'intrus peut intercepter. Une nouvelle clef k est générée à chaque session du protocole dépendant de la clef $\mathsf{pk}(x)$ reçu par A, celle-ci est signée avec la clef secrète de A, et le tout est chiffré avec la clef $\mathsf{pk}(x)$. Lors de la deuxième étape, B s'attend à recevoir un message de la forme $\mathsf{pencrypt}(\mathsf{sign}(k', sk), \mathsf{pk}(sk_B[]))$. Il vérifie que c'est bien A qui a signé le message, et ceci est vrai seulement si $sk = sk_A[]$. Il peut ainsi supposer que la clef k' n'est connue que de A, et il envoie alors le secret s chiffré avec la clef symétrique k', c'est-à-dire le message $\mathsf{sencrypt}(s[], k')$.

La propriété de secret est alors la suivante : l'intrus obtient le secret s si et seulement si le fait $\mathsf{attacker}(s[])$ peut être dérivé des règles (dans un sens que nous allons exposer) ; de manière générale, pour un terme M qui doit rester secret dans le protocole considéré, le fait que M n'est pas secret s'exprimera par $\mathsf{attacker}(M)$.

Notons que les règles employées, notées $H \to C$, sont des clauses de Horn du premier ordre, où H est un ensemble de littéraux tels qu'aucun littéral n'apparaît plus d'une fois dans H. Nous parlerons indifféremment de règles ou de clauses.

L'idée principale de l'algorithme est d'effectuer une recherche de preuve, basée sur la résolution binaire, cette dernière étant correcte est complète dans le cas des clauses de Horn, et guidée de manière à gagner en efficacité. L'algorithme, dont la correction est prouvée, ne termine cependant pas toujours. Pour résoudre ce problème, certains cas de non-terminaison sont détectés afin de pouvoir contourner le problème, même si cette détection n'est pas pour le moment systématique.

L'algorithme comporte deux étapes, la première consistant à transformer l'ensemble B de clauses de base en un ensemble B' impliquant les mêmes faits (rappelons qu'un fait est une clause de Horn positive), la deuxième étant une recherche de preuve arrière en profondeur d'abord.

Un ensemble fini S de faits est fixé, avec par défaut $S = \{\mathsf{attacker}(x)\}$, et $F \in_r S$ si et seulement si il existe un fait dans S égal à F modulo renommage. Par abus de langage, on dira que F appartient à S (et de manière similaire que F n'appartient pas à S si $F \notin_r S$). Soit $R \circ_{F_0} R'$ la résolution binaire entre les règles R et R' sur un littéral négatif F_0 de R'.
L'idée sera d'unifier seulement les faits n'appartenant pas à S. Voici une description de l'algorithme.

I. Transformation de l'ensemble de clauses de base en un autre qui implique les même faits en 3 phases :

1. Élimination des règles qui sont impliquées par d'autres règles (ce qui est en fait de la subsomption avant et arrière)
2. Ajout de clauses créées par résolution de R et R' (et simplification au fur et à mesure par la phase 1.) si :

   a. tous les littéraux négatifs de R sont dans S,

   b. le littéral négatif de R' sur lequel on unifie n'est pas dans S,

   cette procédure étant itérée jusqu'à trouver un point fixe.
3. Ne sont gardées que les clauses dont tous les littéraux négatifs sont dans S (par défaut de la forme $\mathsf{attacker}(x)$).

II. Recherche arrière en profondeur d'abord.
Soit B' l'ensemble de clauses obtenu par l'étape I.
Soit R une clause dont on veut montrer les hypothèses et dont la conclusion est une instance du fait que l'on veut initialement prouver.
Soit B'' l'ensemble des clauses déjà examinées pendant la recherche.
Le processus suivant prend (R, B'') en paramètre :

1. S'il existe une clause $R' \in B''$ telle que R est conséquence de R', cela signifie que la branche courante de la recherche échoue, alors il faut revenir en arrière pour essayer de trouver une dérivation du but ;
2. Si R n'a pas d'hypothèse, la recherche est un succès : la conclusion de R est prouvée ;
3. Sinon, soit $R' \in B'$ avec laquelle on essaye de prouver une des hypothèses F_0 de R ; si $R' \circ_{F_0} R$ est défini, le processus est réitéré avec $R' \circ_{F_0} R$ et $\{R \cup B''\}$.

Initialement, ce processus est appelé avec $(\{F\} \to F, \emptyset)$.

2.4.2 Modèle de trace basé sur des clauses de Horn

H. Comon-Lundh et V. Cortier [CLC03b] définissent un modèle basé sur des clauses de Horn où les termes sont les messages. Bien que similaire à celui de B. Blanchet, celui-ci modélise des traces sur des clauses de Horn, où les termes sont les messages. Les traces possibles sont les éléments de $T_{\mathcal{H}}$, l'interprétation du prédicat unaire T dans le plus petit modèle de Herbrand $\mathcal{H}$, $T(t)$ signifiant que t est une trace valide.

Les traces, représentées par le prédicat T, sont en fait des suites de paires dont l'une représente un évènement (un envoi, une réception ou l'ajout de données à un état de mémoire local), et l'autre un numéro de session. Le modèle est ainsi multi-session et les nonces sont des fonctions de certains messages. L'ensemble des clauses représentant les connaissances de l'intrus sera noté $\mathcal{C}_I$, ces clauses utilisant le prédicat binaire I, qui prend deux arguments : un message m et une trace t ; $I(m, t)$ signifie que le message m est connu de l'intrus après l'exécution de t.

Toute clause représentant une étape du protocole a le prédicat T en partie négative et positive de la clause, permettant ainsi pour chaque évènement de l'ajouter à la trace. Ces clauses font partie de l'ensemble de clauses dépendants du protocole.

L'ensemble de clauses représentant le protocole P sera noté $\mathcal{C}_P$, et est formé de sorte qu'il ne contienne aucune clause négative. $\mathcal{C}_P$ a alors un plus petit modèle de Herbrand $\mathcal{H}_P$.

Une *trace valide* pour un protocole P est un élément de l'interprétation de T dans $\mathcal{H}_P$. Ainsi, une propriété ϕ (s'écrivant avec le même formalisme que les autres clauses) est *satisfaite* par le protocole P si et seulement si $\mathcal{H}_P \models \phi$, et de manière duale une *attaque* sur P pour ϕ est un sous-ensemble fini $\mathcal{H}_0$ de $\mathcal{H}_P$ tel que $\mathcal{H}_0 \not\models \phi$.

Cependant, la décision de ce modèle n'est prouvée que pour un intrus passif (*eavesdropper*), c'est-à-dire un intrus qui ne peut qu'analyser les messages envoyés sur le réseau.

Ce modèle étant trop expressif pour être décidable avec un intrus plus puissant, H. Comon-Lundh et V. Cortier en donnent une approximation [CLC03a], consistant essentiellement à abstraire les nonces par des constantes.

2.4.3 Automates d'arbres

Nous avons vu en section 1.7.3 comment sont représentés les automates d'arbres alternants bidirectionnels. Dans [GLV02], J. Goubault-Larrecq et N.K. Verma définissent les $\mathcal{E}$-automates alternants bidirectionnels, où $\mathcal{E}$ est une théorie équationnelle, par un ensemble de clauses de Horn :

Clauses pop :

$$P(f(x_1,\ldots,x_n)) \Leftarrow P_1(x_1),\ldots,P_n(x_n) \tag{2.1}$$

où $x_1,\ldots,x_n$ sont toutes différentes. Une clause pop définit un *$\mathcal{E}$-automate d'arbre* (si $x_1,\ldots,x_n$ ne sont pas tous différents, cette clause définit un $\mathcal{E}$-automate d'arbre avec *contrainte d'égalité entre frères*).

Clauses intersection :

$$P(x) \Leftarrow P_1(x),\ldots,P_n(x) \tag{2.2}$$

Une clause intersection définit un *$\mathcal{E}$-automate d'arbre alternant*.

ϵ-clause :

$$P(x) \Leftarrow P_1(x) \tag{2.3}$$

Une telle clause est un cas particulier de clause intersection pour $n = 1$.

Clauses push :

$$P_i(x_i) \Leftarrow P(f(x_1,\ldots,x_n)), P_{i_1}(x_{i_1}),\ldots,P_{i_k}(x_{i_k}) \tag{2.4}$$

où $1 \leq i \leq n$ et $1 \leq i_1 < \ldots < i_k \leq n$
si $k = 0$, c'est une clause push *standard*,
si $k \neq 0$, c'est une clause push *conditionnelle*. Une clause push définit un *$\mathcal{E}$-automate bidirectionnel*.

La résolution ordonnée avec splitting le plus tôt possible termine sur un tel ensemble de clauses modulo la théorie équationnelle vide, donnant ainsi une procédure de décision. Le format des clauses est généralisé de la façon suivante :

Clauses simples ou blocs :

$$\bigvee_{i=1}^{m} \pm_i P_i(x) \tag{2.5}$$

On notera souvent $B(x)$ le bloc dont la seule variable libre est x.

Clauses complexes :

$$\bigvee_{i=1}^{m} \pm_i P_i(f(x_1,\ldots,x_n)) \vee \bigvee_{j=1}^{n} B_j(x_j) \tag{2.6}$$

où x_j est l'unique variable libre de $B_j(x_j)$, s'il y en a une, $m \geq 1$, et toutes les variables libres de la clause sont dans $\{x_1,\ldots,x_n\}$.

Le résultat suivant est établi : la résolution ordonnée avec splitting le plus tôt possible termine sur des clauses simples et complexes, c'est-à-dire de la forme (2.5) ou (2.6), avec un ordre sur la taille des termes et modulo la théorie équationnelle vide, et en temps exponentiel non déterministe (le splitting le plus tôt possible est important car les clauses simples et complexes produisent parfois des résolvants qui ne sont ni des clauses simples ni des clauses complexes, mais grâce au splitting on ré-obtient de telles clauses).

Soit Σ une signature contenant des symboles binaires associatifs et commutatifs (AC) $+_1,\ldots,+_p$. Les autres symboles de fonction de Σ sont dits *libres*.

Une clause $+_i$*-pop* est une clause de la forme :

$$P(x +_i y) \Leftarrow P_1(x), P_2(y) \tag{2.7}$$

Une clause $+_i$*-push standard* est une clause de la forme :

$$P(x) \Leftarrow P_1(x +_i y) \tag{2.8}$$

Définition – 2.4.1

Un AC-automate d'arbres standard non-alternant bidirectionnel sur la signature Σ est un ensemble de clauses d'une des formes suivantes :

- *Clauses pop (2.1) libres : $P(f(x_1, \ldots, x_n)) \Leftarrow P_1(x_1), \ldots, P_n(x_n)$, avec f libre.*
- *Clauses $+_i$-pop (2.7) : $P(x +_i y) \Leftarrow P_1(x), P_2(y)$*
- *Clauses push (2.4) libres (possiblement conditionnelle) : $P_i(x_i) \Leftarrow P(f(x_1, \ldots, x_n)), P_{i_1}(x_{i_1}), \ldots, P_{i_k}(x_{i_k})$, avec f libre et $i \notin \{i_1, \ldots, i_k\}$.*
- *Clauses $+_i$-push standards : $P(x) \Leftarrow P_1(x +_i y)$*
- *ϵ-clauses (une clause (2.2) avec $n = 1$) : $P(x) \Leftarrow P_1(x)$*

Les *clauses intersection finale* sont de la forme

$$\perp \Leftarrow P_1(x), \ldots, P_k(x) \tag{2.9}$$

La décidabilité des AC-automates non-alternants bidirectionnel avec clauses intersection finales est alors établie.

Pour l'application aux protocoles cryptographiques, le choix du protocole IKA.1 est pris pour illustrer, et la modélisation de ce protocole aboutit à des clauses de Horn définissant des AC-automates non-alternants bidirectionnel avec clauses intersection finales. Nous reviendrons avec de plus amples détails sur cet exemple en section 6.1.

On peut remarquer que la plupart des clauses du modèle de B. Blanchet sont des clauses d'automates d'arbres. Pour reprendre l'exemple de la modélisation du protocole Denning-Sacco (figure 2.5), les clauses

$$\begin{aligned}\mathsf{attacker}(m), \mathsf{attacker}(pk) &\rightarrow \mathsf{attacker}(\mathsf{pencrypt}(m, pk))\\ \mathsf{attacker}(m), \mathsf{attacker}(sk) &\rightarrow \mathsf{attacker}(\mathsf{sign}(m, sk))\\ \mathsf{attacker}(m), \mathsf{attacker}(k) &\rightarrow \mathsf{attacker}(\mathsf{sencrypt}(m, k))\end{aligned}$$

sont des clauses pop (2.1), la clause

$$\mathsf{attacker}(\mathsf{sign}(m, sk)) \rightarrow \mathsf{attacker}(m)$$

est une clause push standard (2.4), la clause

$$\mathsf{attacker}(\mathsf{sencrypt}(m, k)), \mathsf{attacker}(k) \rightarrow \mathsf{attacker}(m)$$

est une clause push (2.4), et les clauses

$$\mathsf{attacker}(\mathsf{pk}(sk_A[])) \quad \mathsf{attacker}(\mathsf{pk}(sk_B[])) \quad \mathsf{attacker}(a[])$$

sont des clauses pop (2.1) où $n = 0$.

Quant aux autres clauses, on peut utiliser des méthodes d'approximation pour les transformer en clauses d'automates d'arbres, comme nous le ferons en section 6.1.

Il existe notamment une méthode automatique d'approximation d'ensembles de clauses quelconques sous forme de clauses d'automates d'arbres dans [FSVY91], où les programmes logiques sont approximés par une certaine classe de programmes avec des prédicats unaires uniquement représentant un système *descriptif* de types, c'est-à-dire un système qui laisse le programmeur écrire ses programmes sans avoir à mentionner ou à définir de type ; c'est le compilateur qui ensuite produit une inférence de types et vérifie la correction des types. Une analogie est faite entre cette classe de programmes logiques et les automates d'arbres bidirectionnels, nous donnant ainsi une approximation de programmes logiques sous forme d'automates (les programmes logiques étant des ensembles de clauses).

Le résultat de décidabilité sur les AC-automates d'arbres bidirectionnels avec clauses intersection finales nous permet alors d'étendre ces méthodes d'approximation à des ensembles de clauses possédant une théorie équationnelle.

On pourra aussi consulter [CLC03a] pour une extension du modèle de traces présenté en section 2.4.2 à la théorie équationnelle du ou exclusif (xor).

Deuxième partie

Un outil modulaire de résolution

Chapitre 3

Résolution abstraite

Nous avons présenté au chapitre 1 un raffinement de la résolution, la résolution ordonnée avec sélection, pour la logique du premier ordre, comme une technique de démonstration automatique. Cependant, on a souvent besoin de théories équationnelles, notamment en vérification de protocoles cryptographiques. Nous verrons en effet au chapitre 6 que la vérification du protocole IKA.1 nécessite l'utilisation d'opérateurs associatifs et commutatifs dans le cadre de la logique du premier ordre. De plus, on peut aussi vouloir faire de la résolution sur des ensembles de clauses qui ne sont pas des clauses du premier ordre. Or développer un outil de vérification à chaque fois que l'on a besoin de travailler sur une certaine théorie équationnelle, ou sur une certaine famille de termes, peut se révéler fastidieux. Notre objectif est de faire un prouveur plus général, basé la résolution ordonnée avec sélection, qui pourrait être utilisé pour différentes théories équationnelles et pour différentes familles de termes. Pour cela, nous allons définir la notion de *termes symboliques* comme étant des termes formés sur un ensemble de variables répondant à certaines conditions. Pour donner une idée, les termes du premier ordre ou les λ-termes modulo β sont des cas particuliers de familles de termes symboliques. Ceci nous permettra d'établir, pour un ensemble de *clauses symboliques*, c'est-à-dire des clauses formées à partir de termes symboliques, une procédure de résolution, la *résolution abstraite ordonnée avec sélection*.

Nous définissons les termes et les clauses symboliques en section 3.1, puis nous donnons des éléments de sémantique (section 3.2) qui nous permettrons de montrer la correction et la complétude de la résolution abstraite ordonnée avec sélection (section 3.3). Nous verrons ensuite qu'avec certaines stratégies d'élimination de clauses (section 3.4), avec le splitting classique ou le splitting sans splitting (section 3.5.2), cette résolution abstraite reste correcte et complète. Ceci nous permettra d'utiliser cette procédure de résolution pour différentes familles de termes symboliques comme nous le verrons au chapitre 5.

3.1 Termes et clauses symboliques

Nous allons ici définir le langage des termes symboliques que nous utiliserons tout au long de ce chapitre, puis les clauses sur ces même termes symboliques qui seront les objets que nous manipulerons dans la résolution abstraite ordonnée avec fonction de sélection.

3.1.1 Termes symboliques

Fixons $\mathcal{V}$ un ensemble infini dénombrable de *variables d'individus*.

Définition – 3.1.1

> *On dit qu'une famille d'ensembles* $(T_A)_{\substack{A\subseteq\mathcal{V}\\ A \text{ fini}}}$ *est une famille de* termes symboliques *si et seulement si :*
> - $A \subseteq T_A$
> - *si* $A \subseteq B$*, alors* $T_A \subseteq T_B$
> - *pour tout* $A, B \subseteq \mathcal{V}$*, il existe une application* $Subst_{A,B} : (A \to T_B) \to (T_A \to T_B)$ *telle que :*
>
> $$\forall A' \supseteq A. \forall t \in T_A \ Subst_{A',B}(\sigma)(t) = Subst_{A,B}(\sigma_{|A})(t) \quad (3.1)$$
> $$\text{où } \sigma_{|A} = \lambda x \in A.\sigma(x)$$
> $$\forall B' \supseteq B. \forall t \in T_A \ Subst_{A,B}(\sigma)(t) = Subst_{A,B'}(\sigma)(t) \quad (3.2)$$
> $$Subst_{A,A}(\lambda x \in A.x)(t) = t \quad (3.3)$$
> $$\forall C \subseteq \mathcal{V}. Subst_{B,C}(\sigma')(Subst_{A,B}(\sigma)(t)) = Subst_{A,C}(\lambda x \in A.Subst_{B,C}(\sigma')(\sigma(x)))(t) \quad (3.4)$$

Ainsi, T_A est un ensemble de termes symboliques de variables libres prises dans A, et les équations (3.1) à (3.4) définissent une application qui généralise la notion de substitution telle que nous l'avons vu pour les termes du premier ordre (chapitre 1).

Par la suite, afin d'alléger les notations, et lorsqu'il n'y aura pas d'ambiguïté, nous écrirons :
- $(T_A)_A$ pour $(T_A)_{\substack{A\subseteq\mathcal{V}\\ A \text{ fini}}}$,
- $\bigcup_A$ pour $\bigcup_{\substack{A\subseteq\mathcal{V}\\ A \text{ fini}}}$,
- $\bigcap_A$ pour $\bigcap_{\substack{A\subseteq\mathcal{V}\\ A \text{ fini}}}$.

Définition – 3.1.2

> *Une famille de termes symboliques* $(T_A)_A$ *est dite* normale *si et seulement si pour tout* $A, B \subset \mathcal{V}$*,* A *et* B *finis,* $T_A \cap T_B = T_{A\cap B}$.

Pour une famille quelconque de termes symboliques, $T_A \cap T_B$ contient $T_{A\cap B}$. En effet, par définition, $T_{A\cap B} \subseteq T_A$ et $T_{A\cap B} \subseteq T_B$, donc trivialement, $T_{A\cap B} \subseteq T_A \cap T_B$. Dans une famille normale de termes symboliques, il n'y a pas de *reliquats*, c'est-à-dire pas de termes qui sont $T_A \cap T_B$ et qui ne sont pas dans $T_{A\cap B}$.

Proposition – 3.1.1

> *Si* $(T_A)_A$ *est une famille normale de termes symboliques, alors, pour tout* $t \in \cup_A T_A$*, il existe un unique ensemble* A *minimal tel que* $t \in T_A$.

Démonstration. Si $t \in \cup_A T_A$, alors il existe A_0 tel que $t \in T_{A_0}$. Posons $A \hat{=} \bigcap_{B\subseteq A_0/t\in T_B} B$.

De manière évidente, on a $t \in \bigcap_{B\subseteq A_0/t\in T_B} T_B$. De plus, A étant défini comme une intersection finie et non vide, par normalité de la famille de termes symboliques, $\bigcap_{B\subseteq A_0/t\in T_B} T_B = T_{\bigcap_{B\subseteq A_0/t\in T_B}} = T_A$, donc $t \in T_{A_0}$.

Montrons que A est minimal. Soit A' un ensemble tel que $t \in T_{A'}$ et $A' \subseteq A$. Comme $t \in T_{A'}$, on a $\bigcap_{B\subseteq A_0/t\in T_B} B \subseteq A'$, soit $A \subseteq A'$. Donc $A = A'$. □

Définition – 3.1.3

> *Si $(T_A)_A$ est une famille normale de termes symboliques, alors, pour tout $t \in \cup_A T_A$, l'ensemble des* variables libres *de t est l'unique ensemble minimal A tel que $t \in T_A$. Notons $fv(t)$ cet ensemble.*

En pratique, toutes les familles de termes que nous considérerons seront des familles de termes symboliques normales. C'est pourquoi, pour la suite, nous faisons l'hypothèse que les familles de termes symboliques sont normales.

Ainsi, quand nous écrirons qu'un terme symbolique t appartient à l'ensemble T_A, cela sous-entendra, sauf mention explicite du contraire, que A *est* l'ensemble des variables libres de t.

3.1.2 Quelques familles de termes symboliques

Pour illustrer notre propos, nous allons présenter quelques exemples de familles de termes symboliques, dont nous pouvons vérifier qu'ils rentrent dans le cadre de la définition que nous avons donné.

3.1.2.1 Les termes finis du premier ordre

Étant donnés un ensemble $\mathcal{V}$ infini dénombrable de variables d'individus, notées par x, y, z, un ensemble $\mathcal{F}$ au plus dénombrable de symboles de fonction d'arité fixée, notés f, g, h, un terme fini du premier ordre est défini comme suit :

- x est un terme fini du premier ordre ;
- si $t_1, \ldots, t_n$ sont des termes finis du premier ordre, alors $f(t_1, \ldots, t_n)$ est un terme fini du premier ordre pour tout f dans $\mathcal{F}$ d'arité n.

Notons $\mathcal{T}(F, V)$, avec $F \subseteq \mathcal{F}$ et $V \subset \mathcal{V}$ fini, l'ensemble des termes finis du premier ordre construits sur F et V.

Rappelons que l'*ensemble des variables libres d'un terme fini du premier ordre t* sont définies par l'ensemble $fv(t)$ tel que si t est une variable x, alors $fv(t) = \{x\}$, et si t est de la forme $f(t_1, \ldots, t_n)$ alors $fv(t) = fv(t_1) \cup \ldots \cup fv(t_n)$.

Une substitution est une application des variables vers les termes, et le résultat de l'application d'une substitution à un terme, notée $t\sigma$, est telle que $x\sigma = \sigma(x)$ et $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$. Le domaine d'une substitution σ, noté $\mathsf{dom}(\sigma)$ est l'ensemble fini des variables $x \in \mathcal{V}$ telles que $\sigma(x) \neq x$ et son co-domaine, noté $\mathsf{codom}(\sigma)$ est l'ensemble fini des termes $\{\sigma(x) \mid x \in \mathsf{dom}(\sigma)\}$.

Proposition – 3.1.2

> *Les termes finis du premier ordre sont une famille de termes symboliques.*

Démonstration. Notons T^1_A l'ensemble des termes construits à partir d'un ensemble fini $A \subset \mathcal{V}$ et de $\mathcal{F}$. Pour toute substitution σ, il existe un ensemble fini $A \subset \mathcal{V}$ tel que $\mathsf{codom}(\sigma) \subseteq T^1_A$ (il suffit de prendre pour A n'importe quel ensemble incluant l'union des ensembles des variables libres de tous les termes du co-domaine de σ). Notons $[]$ la substitution identité, c'est-à-dire la substitution σ telle que pour toute variable $x \in \mathcal{V}$, $\sigma(x) = x$.
Alors, pour tout A, B finis $A, B \subset \mathcal{V}$:

- $A \subseteq T^1_A$
- si $A \subseteq B$, alors $T^1_A \subseteq T^1_B$
- $\forall A \subseteq A'. \forall t \in T^1_A\ t\sigma = t\sigma_{|A}$
- $\forall \sigma : A \to T^1_B\ \forall B \subseteq B'.\ \forall \sigma' : A \to T^1_{B'}$ telle que $\sigma_{|A} = \sigma'_{|A}$ alors $\sigma = \sigma'$
- $t[] = t$
- la *composition* de deux substitutions σ et θ, notée $\sigma\theta$, est l'unique substitution telle que $t(\sigma\theta) = (t\sigma)\theta$ pour tout terme t : $\sigma\theta$ associe à toute variable x le terme $\sigma(x)\theta$.

Les différentes conditions et les équations (3.1) à (3.4) de la définition 3.1.1 des termes symboliques sont ainsi vérifiées. □

Proposition – 3.1.3

> *Les termes finis du premier ordre sont une famille normale de termes symboliques.*

Démonstration. Soient A, B des sous-ensembles finis de $\mathcal{V}$ et F un sous-ensemble de $\mathcal{F}$ tels qu'il existe un terme t dans $\mathcal{T}(F, A) \cap \mathcal{T}(F, B)$. A et B contiennent alors les variables libres de t, donc $A \cap B$ aussi, et, par définition, t est dans $\mathcal{T}(F, A \cap B)$. Ainsi, $\mathcal{T}(\mathcal{F}, \mathcal{V})$ est une famille normale de termes symboliques. □

3.1.2.2 Les termes modulo une théorie équationnelle quelconque

Soient Σ une signature, X un ensemble de variables et E une théorie équationnelle quelconque. Notons $T(\Sigma, X)/_E$ les termes formés à partir de Σ modulo la théorie équationnelle E.

On pourrait définir assez naturellement $T_A = \{[t]_E \mid \exists u.t \approx_E u.\ fv(u) \subseteq A\}$. Mais cette famille ainsi définie n'est pas forcément normale. Prenons par exemple $E = \{f(x+y) = g(y+z)\}$, alors le terme $f(x+y)$ est dans $T_{\{x,y\}}$. Il est donc dans $T_{\{y,z\}}$ car $f(x,y) \approx_E g(y+z)$, mais il n'est pas dans $T_{\{x,y\} \cap \{y,z\}} = T_{\{y\}}$. Guidé par cet exemple, on voit que même si $f(x,y)$ dépend syntaxiquement de x, le fait que $f(x+y) \approx_E g(y+z)$ montre que, modulo E, $f(x+y)$ ne peut pas varier en fonction de x. Nous allons voir dans la démonstration de la proposition suivante, que nous allons modifier la définition de T_A de telle sorte que ce problème ne se présente plus. L'idée sera de ne considérer que les variables qui sont libres dans tous les termes égaux modulo E.

On dit que E est *dégénérée* si et seulement s'il existe deux variables distinctes x et y telles que $x \approx_E y$.

Proposition – 3.1.4

> *Si E est une théorie équationnelle non dégénérée, alors $T(\Sigma, X)/_E$ définit une famille normale de termes symboliques.*

Démonstration. Lorsque deux termes t et u sont égaux modulo la théorie équationnelle E, on note $t \approx_E u$. Si t est un terme de $T(\Sigma, X)$, on note $[t]_E$ l'ensemble des termes de $T(\Sigma, X)$ égaux à t modulo E.

Posons :
- $T_A = \{[t]_E \mid t \in T(\Sigma, X).\ \forall x \in fv(t) - A. \exists u.t \approx_E u.x \notin fv(u)\}$. On identifie A au sous-ensemble des $[x]_E$, $x \in A$, de T_A, ceci étant possible car E est non dégénérée.
- $Subst_{A,B}(\sigma)([t]_E) = [t\sigma']_E$, où pour tous $x \in A$, on choisit l'élément $\sigma'(x)$ dans $\sigma(x) \in T_B$.

Montrons que $Subst_{A,B}(\sigma)([t]_E)$ est bien défini. Pour cela, montrons que :
- $[t\sigma']_E$ ne dépend ni du choix de t, ni du choix de $\sigma'(x)$ pour $x \in A$.
  Soient t_1 et t_2 deux éléments de $[t]_E$, et soient, pour tout $x \in A$, $\sigma'_1(x)$ et $\sigma'_2(x)$ deux éléments de $\sigma(x)$. Alors $t_1 \approx_E t_2$ et, pour tout $x \in A$, $\sigma'_1(x) \approx_E \sigma'_2(x)$. Donc $t_1\sigma'_1 \approx_E t_2\sigma'_2$, et ainsi $[t_1\sigma'_1]_E = [t_2\sigma'_2]_E$.
- $[t\sigma']_E$ est dans T_B.
  $[t\sigma']_E \in T_B$ si et seulement si $\forall y \in fv(t\sigma') - B.\ \exists u.\ t\sigma' \approx_E u.\ y \notin fv(u)$. Pour tout $x \in fv(t)$, on a deux cas :
  - si $y \in fv(x\sigma')$, alors $y \in fv(x\sigma') - B$. Comme $\sigma(x) \in T_B$, $\exists w_x.\ w_x \approx_E \sigma'(x).\ y \notin fv(w_x)$.
  - sinon, posons $w_x = x\sigma'$, ainsi $y \notin fv(w_x)$.

  Posons σ'' la substitution qui à x associe w_x ; par les deux cas précédents, on sait que $w_x \approx_E \sigma'(x)$ et $y \notin fv(w_x)$.
  Comme $y \in fv(t\sigma'')$, alors $y \in fv(t)$ et $y \notin \mathsf{dom}(\sigma'')$. Comme $\mathsf{dom}(\sigma'') = \mathsf{dom}(\sigma') = A$, alors $y \notin A$. Donc $y \in fv(t) - A$, et comme $[t]_E \in T_A$, on a $\exists v.\ t \approx_E v.\ y \notin fv(v)$. Alors $y \notin fv(v\sigma'')$ et $t\sigma' \approx_E v\sigma''$. Ainsi, $\forall y \in fv(t\sigma') - B.\ \exists u.\ t \approx_E u.\ y \notin fv(u)$, donc $[t\sigma']_E$ est dans B.

Montrons que $(T_A)_{\substack{A \subseteq X \\ A \text{ fini}}}$ est une famille de termes symboliques.
- On a $A \subseteq T_A$ par construction.

– Montrons que si $A \subseteq B$ alors $T_A \subseteq T_B$. Si $[t]_E \in T_A$, alors pour tout x dans $fv(t) - A$, il existe u tel que $u \approx_E t$ et $x \notin fv(u)$. Comme $A \subseteq B$, on a $fv(t) - B \subseteq fv(t) - A$, donc pour tout $x \in fv(t) - B$, il existe u tel que $u \approx_E t$ et $x \notin fv(u)$. Ainsi, $[t]_E \in T_B$, et $T_A \subseteq T_B$.
– Montrons que pour tout A, A' tels que $A \subseteq A'$ et pour tout $[t]_E \in T_A$, on a $Subst_{A',B}(\sigma)([t]_E) = Subst_{A,B}(\sigma_{|A})([t]_E)$ (l'équation (3.1)), c'est-à-dire que $[t\sigma']_E = [t\sigma'']_E$, où pour tout $x \in A'$, $\sigma'(x)$ est choisi dans $\sigma(x)$, et pour tout $x \in A$, $\sigma''(x)$ est choisi dans $\sigma_{|A}(x)$. Donc, pour tout $x \in A$, $[\sigma'(x)]_E = [\sigma''(x)]_E$. Posons $\sigma'' = [x_1/u_1, \ldots, x_m/u_m]$ et $\sigma' = [x_1/u_1, \ldots, x_n/u_m, x_{m+1}/u_{m+1}, \ldots, x_n/u_n]$, avec $x_{m+1}, \ldots, x_n \in A' - A$. Montrons par récurrence sur $n - m$ que $t\sigma' \approx_E t\sigma''$.
Si $n = m$, c'est évident. Sinon, on a

$$t\sigma' \begin{cases} = t[x_1/u_1, \ldots, x_n/x_n] & \text{si } x_n \notin fv(t) \\ \approx_E v[x_1/u_1, \ldots, x_n/u_n] & \text{si } x_n \in fv(t), \text{ car } x_n \notin A \text{ donc } \exists v.\ t \approx_E t.\ x \notin fv(v) \end{cases}$$

Or,
$$\begin{aligned} t[x_1/u_1, \ldots, x_n/x_n] &= t[x_1/u_1, \ldots, x_{n-1}/u_{n-1}] \\ &\approx_E t\sigma'' \text{ par hypothèse de récurrence.} \end{aligned}$$

De plus,
$$\begin{aligned} v[x_1/u_1, \ldots, x_n/u_n] &= v[x_1/u_1, \ldots, x_n/x_n] \text{ car } x_n \notin fv(v) \\ &\approx_E t[x_1/u_1, \ldots, x_n/x_n] \text{ car } v \approx_E t \\ &= t[x_1/u_1, \ldots, x_{n-1}/u_{n-1}] \\ &\approx_E t\sigma'' \text{ par hypothèse de récurrence.} \end{aligned}$$

Ainsi, $t\sigma' \approx_E t\sigma''$, et l'équation est vérifiée.
– Montrons que pour tout B, B' tels que $B \subseteq B'$, pour tout $[t]_E \in T_A$ et pour tout $\sigma : A \to T_B$, on a $Subst_{A,B}(\sigma)([t]_E) = Subst_{A,B'}(\sigma)([t]_E)$ (l'équation (3.2)).
Soit σ' une substitution telle que pour tout x dans A, on a $\sigma'(x) \in \sigma(x)$, où $\sigma(x) \in T_B$, et soit σ'' telle que pour tout x dans A on a $\sigma''(x) \in \sigma(x)$, où $\sigma(x) \in T_{B'}$. Donc $\sigma'(x) \approx_E \sigma''(x)$ pour tout x dans A, donc $t\sigma' \approx_E t\sigma''$.
– Montrons que $Subst_{A,A}(\lambda x \in A.x)([t]_E) = [t]_E$ (l'équation (3.3)).
Posons $\sigma_0 = \lambda x \in A.x$, on veut alors montrer que $[t\sigma'_0]_E = [t]_E$, où pour tout $x \in A$, $\sigma'_0(x)$ est choisi dans $\sigma_0(x)$. Or, pour tout $x \in A$, $\sigma_0(x) = x$, donc $\sigma'_0(x) \approx_E x$, et ainsi $t\sigma'_0 \approx_E t$, donc $[t\sigma'_0]_E = [t]_E$.
– Soient $\sigma_2 : A \to T_B$ et $\sigma_1 : B \to T_C$, avec $A, B, C \subseteq X$, et soit $t \in T_A$. Montrons l'équation (3.4), soit $Subst_{B,C}(\sigma_1)(Subst_{A,B}(\sigma_2)([t]_E)) = Subst_{A,C}(\lambda x \in A.Subst_{B,C}(\sigma_1)(\sigma_2(x)))([t]_E)$. Posons $\sigma_3 = \lambda x \in A.Subst_{B,C}(\sigma_1)(\sigma_2(x))$. Soient σ'_1 telle que pour tout y dans B, $\sigma'_1(y) \in \sigma_1(y)$, et σ'_2 telle que pour tout x dans A, $\sigma'_2(x) \in \sigma_2(x)$. Soit σ'_3 telle que pour tout x dans A, $\sigma'_3(x) \in \sigma_3(x)$, où $\sigma_3(x) = Subst_{B,C}(\sigma_1)(\sigma_2(x))$. Donc $\sigma_3(x) = [\sigma'_2(x)\sigma'_1]_E$, d'où $\sigma'_3(x) \approx_E \sigma'_2(x)\sigma'_1$ pour tout x dans A. Ainsi, $[t\sigma'_3]_E = [t\sigma'_2\sigma'_1]$.
De plus,

$$\begin{aligned} Subst_{B,C}(\sigma_1)(Subst_{A,B}(\sigma_2)([t]_E)) &= [(t\sigma'_2)\sigma'_1]_E \\ &= [t\sigma'_2\sigma_1]_E \\ &= [t\sigma'_3]_E \end{aligned}$$

Ainsi, la dernière équation est vérifiée, donc $T(\Sigma, X)/_E$ définit bien une famille de terme symbolique.

Montrons maintenant la normalité de la famille.
Si $[t]_E \in T_A \cap T_B$, alors pour tout $x \in fv(t) - (A \cap B)$, soit $x \in fv(t) - A$, soit $x \in fv(t) - B$. Dans le premier cas, $x \in fv(t) - A$, donc il existe u tel que $u \approx_E t$ et $x \notin fv(u)$, et dans le deuxième cas, $x \in fv(t) - B$, donc il existe u tel que $u \approx_E t$ et $x \notin fv(u)$. Donc $t \in T_{A \cap B}$. □

Réciproquement, on peut noter que toute famille normale de termes symboliques $(T_A)_A$ peut-être définie comme un certain $T(\Sigma, X)/_E$. En effet, fixons une énumération $x_1, \ldots, x_n, \ldots$ d'éléments de $\mathcal{V}$. Soit $A = \{x_{i_1}, \ldots, x_{i_k}\}$; pour tout $t \in T_A$ tel que $A = fv(t)$, on crée un symbole de fonction $\ulcorner t \urcorner$ d'arité k, et on pose $\overline{t} = \ulcorner t \urcorner(x_{i_1}, \ldots, x_{i_k})$. Soit alors la théorie équationnelle E définie par :

$$\begin{aligned}
\overline{Subst_{A',B}(\sigma)(t)} &= \overline{Subst_{A,B}(\sigma_{|A})(t)} \\
&\quad \text{pour chaque } t \in T_A, A \subseteq A',\ \sigma : A' \to T_B \\
\overline{Subst_{A,B}(\sigma)(t)} &= \overline{Subst_{A,B'}(\sigma)(t)} \\
&\quad \text{pour chaque } t \in T_A,\ B \subseteq B'\ \sigma : A' \to T_B \\
\overline{Subst_{A,A}(\lambda x \in A.x)(t)} &= \overline{t} \\
&\quad \text{pour chaque } t \in T_A \\
\overline{Subst_{B,C}(\sigma')(Subst_{A,B}(\sigma)(t))} &= \overline{Subst_{A,C}(\lambda x \in A.Subst_{B,C}(\sigma')(\sigma(x)))(t)} \\
&\quad \text{pour chaque } t \in T_A,\ \sigma : A \to T_B,\ \sigma' : B \to T_C
\end{aligned}$$

3.1.2.3 Les termes rationnels du premier ordre

Les termes rationnels permettent de décrire des structures cycliques. Il existe diverses façons de les définir. Nous en donnerons une définition en section 5.2. Dans [CJ97], les termes du premier ordre sont définis à l'aide de suites ordonnées de mots, appelées des *ensembles de positions*, un terme étant alors vu comme une application d'un ensemble de positions vers l'union des ensembles des symboles de fonctions et de variables respectivement. Nous allons présenter plus formellement toutes ces notions, permettant ensuite de définir les termes rationnels comme des termes ayant un nombre fini de sous-termes disjoints.

Tout d'abord, présentons la notion d'ensemble de positions. Soit $\mathbb{N}$ l'ensemble des entiers naturels, $\mathbb{N}_+$ l'ensemble des entiers naturels non nuls, $\mathbb{N}_+^*$ l'ensemble des suites finies de mots sur le vocabulaire $\mathbb{N}_+$, et $\Lambda \in \mathbb{N}_+^*$ le mot vide. Chaque $i \in \mathbb{N}+$ est identifié à la suite contenant l'unique élément i. Si $p, q \in \mathbb{N}_+^*$, alors $p.q$ dénote leur concaténation, opération interne dont la suite vide est élément neutre à droite et à gauche. $\mathbb{N}_+^*$ est muni de la relation d'ordre $\leq_{pref}$ définie par : $\forall p, q \in \mathbb{N}_+^*, p \leq_{pref} q \Leftrightarrow \exists r \in \mathbb{N}_+^*, p.r = q$. On dit que p est *préfixe* de q lorsque $p \leq_{pref} q$. On écrit que $p \bowtie q$ lorsque p et q sont incomparables, c'est-à-dire que ni p n'est pas préfixe de q, ni q de p. L'ordre *hiérarchique* $\leq_{hier}$ étend l'ordre préfixe en un ordre total : $\forall p, q \in \mathbb{N}_+^*, p \leq_{hier} q \Leftrightarrow p \leq_{pref} q \vee (\exists r, i, p', j, q' \in \mathbb{N}_+^*, p = r.i.p', q = r.j.q' \wedge i <_{\mathbb{N}_+^*} j)$.

Un ensemble de *positions* est un sous-ensemble non vide de $\mathbb{N}_+^*$ fermé par ordre hiérarchique, donc contenant Λ, appelée *racine*, et dont les positions maximales pour l'ordre préfixe sont appelées les *feuilles*.

Soit F un ensemble non vide de symboles de fonctions d'arité fixée et X un ensemble non vide disjoint de F de variables d'individus. On notera F_n les fonctions de F d'arité n.

Définition – 3.1.4

> *Un* arbre étiqueté*, encore appelé* terme*, est une application t d'un ensemble de positions noté $Pos(t)$ dans $F \cup X$, qui respecte l'arité des fonctions :*
> - *$t(p) \in F_0 \cup X$ si et seulement si p est une feuille de $Pos(t)$;*
> - *$t(p) = f$ avec $f \in F_n$ si et seulement si pour tout $i \in \mathbb{N}$, $1 \leq i \leq n$, $p.i \in Pos(t)$.*

Un terme est *fini* si et seulement si son ensemble de positions est fini. On note $T(F)$ l'ensemble des termes finis sur le vocabulaire F, et on note $T(F, X)$ l'ensemble $T(F \cup X)$. Les termes de $T(F)$ sont appelés des termes *clos*. On remarquera que $T(F, X)$ définit exactement l'ensemble des termes finis du premiers ordre tels que nous les avons présentés en section 3.1.2.1.

Étant donnés un terme t et une position $p \in Pos(t)$, le *sous-terme* de t à la position p, noté $t|_p$, est défini par :

- $Pos(t|_p) = \{q \in \mathbb{N}_+^* \mid p.q \in Pos(t)\}$
- $\forall q \in Pos(t|_p)$, $(t|_p)(q) = t(p.q)$

Définition – 3.1.5

> *Un terme* rationnel *est un terme n'ayant qu'un nombre fini de sous-termes distincts. On note $RT(F,X)$ l'ensemble des termes rationnels.*

Définition – 3.1.6

> *Une* substitution *est une application σ de l'ensemble X des variables vers l'ensemble $RT(F,X)$ des termes rationnels telle que pour tout terme u, l'application $u\sigma$ est définie par :*
> - $Pos(u\sigma) = \{q \in Pos(u) \mid u(q) \notin X\} \cup \{q.p \mid q \in Pos(u), u(q) \in X, p \in Pos(\sigma(u(q)))\}$;
> - $(u\sigma)(q) = u(q)$ *si* $u(q) \notin X$;
> - $(u\sigma)(q.p) = \sigma(u(q))(p)$ *si* $q \in Pos(u)$, $u(q) \in X$, $p \in Pos(\sigma(u(q)))$.

On définit le *domaine d'une substitution* σ, noté $\mathsf{dom}(\sigma)$ comme l'ensemble $\mathsf{dom}(\sigma) = \{x \in X \mid \sigma(x) \neq x\}$.

Proposition – 3.1.5

> *L'ensemble des termes rationnels est une famille normale de termes symbolique.*

Démonstration. Posons T_A l'ensemble des termes rationnels t dont les variables libres sont dans $A \subseteq X$, c'est-à-dire l'ensemble des applications t de $P(t)$ dans $F \cup A$ telles que

1. $t(p) \in F_0 \cup A$ si et seulement si p est une feuille de $Pos(t)$;
2. $t(p) = f$ si et seulement si pour tout $n \in \mathbb{N}$, $1 \leq i \leq n$, $p.i \in Pos(t)$;
3. t a un nombre fini de sous-termes distincts.

Posons $Subst_{A,B}(\sigma)(t) = t\sigma$, avec $t \in T_A$ et $\sigma : A \to T_B$, et montrons que T_A est une famille de termes symboliques.

- Par définition, si $x \in A$, alors $x \in T_A$, donc $A \subseteq T_A$.
- Soient $A, B \subseteq X$ tels que $A \subset B$. montrons que $T_A \subset B$.
  Soit $t \in T_A$, alors $t(p) \in F_0 \cup A$ si et seulement si p est une feuille de $Pos(t)$, donc comme $A \subset B$ et $t \in T_A$, alors $t(p) \in F_0 \cup A$ si et seulement si p est une feuille de $Pos(t)$. Les points 2. et 3. étant vérifiés trivialement, on a $t \in T_B$.
- Montrons que pour tout A, A' tels que $A \subseteq A'$ et pour tout $t \in T_A$, on a $Subst_{A',B}(\sigma)(t) = Subst_{A,B}(\sigma_{|A})(t)$ (l'équation (3.1)).
  On a $Pos(t\sigma_{|A}) = \{q \in Pos(t) \mid t(q) \notin X\} \cup \{q.p \mid q \in Pos(t), t(q) \in X, p \in Pos(\sigma_{|A}(t(q)))\} = \{q \in Pos(t) \mid t(q) \notin \mathsf{dom}(\sigma_{|A})\} \cup \{q.p \mid q \in Pos(t), t(q) \in \mathsf{dom}(\sigma_{|A}), p \in Pos(\sigma_{|A}(t(q)))\}$. Or, pour tout $q \in Pos(t)$, si $t(q) \notin \mathsf{dom}(\sigma_{|A})$, alors, comme $t \in T_A$ et $A \subseteq A'$, on a $\{q \in Pos(t) \mid t(q) \notin \mathsf{dom}(\sigma_{|A})\} = \{q \in Pos(t) \mid t(q) \notin \mathsf{dom}(\sigma)\}$.
  De plus, si $q \in Pos(t)$ et $t(q) \in \mathsf{dom}(\sigma_{|A})$, alors $t(q) \in \mathsf{dom}(\sigma)$ car $A \subseteq A'$, et $\sigma_{|A}(t(q)) = \sigma(t(q))$, donc $Pos(\sigma_{|A}(t(q))) = Pos(\sigma(t(q)))$.
  Ainsi, $Pos(t\sigma_{|A}) = Pos(t\sigma)$.
  Par les mêmes arguments, on montre que $t\sigma_{|A}(q) = (t\sigma)(q)$ et $t\sigma_{|A}(q.p) = t\sigma(q.p)$.
  Donc $t\sigma_{|A} = t\sigma$.
- Montrons que pour tout B, B' tels que $B \subseteq B'$, pour tout $t \in T_A$ et pour tout $\sigma : A \to T_B$, on a $Subst_{A,B}(\sigma)(t) = Subst_{A,B'}(\sigma)(t)$ (l'équation (3.2)).
  $Subst_{A,B}(\sigma)(t) = t\sigma$ avec $t\sigma \in T_B$. Comme $B \subseteq B'$, $T_B \subseteq T_{B'}$, donc $t\sigma \in T_{B'}$ et $Subst_{A,B'}(\sigma)(t) = t\sigma$. Ainsi, l'équation est vérifiée.
- Montrons que $Subst_{A,A}(\lambda x \in A.x)(t) = t$ (l'équation (3.3)), soit que $t(\lambda x \in A.x) = t$.
  Notons $\sigma_0 = \lambda x \in A.x$, et étendons σ_0 tel que $\sigma_0 = \lambda x \in X.x$, alors clairement $Pos(t\sigma_0) = Pos(t)$. De plus, $(t\sigma_0)(q) = t(q)$ si $t(q) \notin X$, et $(t\sigma_0)(q.p) = \sigma_0(t(q))(p) = t(q.p)$ car si $q \in Pos(t)$ et $t(q) \in X$, alors $\sigma_0(t(q)) = t(q)$ et $Pos(\sigma_0(t(q))) = Pos(t(q))$.
  Ainsi, $t\sigma_0 = t$.
- Montrons enfin l'équation (3.4), c'est-à-dire $(t\sigma)\sigma' = t(\lambda x \in A.(\sigma(x))\sigma')$ pour $t \in T_A$, $\sigma : A \to T_B$ et $\sigma' : B \to T_C$.

$$
\begin{array}{rcll}
Pos((t\sigma)\sigma') & = & \{q \in Pos(t\sigma) \mid t\sigma(q) \notin X\} & \\
 & & \cup\{q.p \mid q \in Pos(t\sigma),\ t\sigma(q) \in X,\ p \in Pos(\sigma'(t\sigma(q)))\} & \\
 & = & \{q' \in Pos(t) \mid t(q') \notin X,\ t\sigma(q') \notin X\} & (1)\\
 & & \cup\{q'.p' \mid q' \in Pos(t),\ t(q') \in X,\ p' \in Pos(\sigma(t(q))),\ t\sigma(q'.p') \notin X\} & (2)\\
 & & \cup\{q'.p \mid q' \in Pos(t),\ t(q') \notin X,\ q' \in Pos(t\sigma),\ t'\sigma(q) \in X, & (3)\\
 & & \qquad p \in Pos(\sigma'(t\sigma(q')))\} & \\
 & & \cup\{q'.p'.p \mid q' \in Pos(t),\ t(q') \in X,\ p' \in Pos(\sigma(t(q))),\ q'.p' \in Pos(t\sigma), & (4)\\
 & & \qquad t\sigma(q'.p') \in X,\ p \in Pos(\sigma'(t\sigma(q'.p')))\} &
\end{array}
$$

$(1) = \{q' \in Pos(t) \mid t(q') \notin X\}$ car si $t(q') \notin X$, alors $t\sigma(q') = t(q')$
$(2) = \{q'.p' \mid q' \in Pos(t),\ t(q') \in X,\ p' \in Pos(\sigma(t(q'))),\ \sigma(t(q'))(p') \notin X\}$
car $t\sigma(q'.p') = \sigma(t(q'))(p')$ dans ce cas
$(3) = \emptyset$ car $t\sigma(q') \in X$ et comme $t(q') \notin X$, alors $t\sigma(q') = t(q')$
$(4) = \{q'.p'.p \mid q' \in Pos(t),\ t(q') \in X,\ p' \in Pos(\sigma(t(q))),\ \sigma(t(q'))(p') \in X,$
$p \in Pos(\sigma'(\sigma(t(q')))(p'))\}$
car $t\sigma(q'.p') = \sigma(t(q'))(p')$ dans ce cas,
et $p' \in Pos(\sigma(t(q')))$ implique $q'.p' \in Pos(t\sigma)$.

Par ailleurs,

$$
\begin{array}{rcl}
Pos(t(\lambda x \in A.\sigma(x)\sigma')) & = & \{q \in Pos(t) \mid t(q) \notin X\} \\
 & & \cup\{q.p \mid q \in Pos(t),\ t(q) \in X,\ p \in Pos(\sigma(t(q))\sigma')\} \\
 & = & \{q' \in Pos(t) \mid t(q') \notin X\} \\
 & & \cup\{q.q' \mid q \in Pos(t),\ t(q) \in X,\ q' \in Pos(\sigma(t(q))),\ \sigma(t(q)) \notin X\} \\
 & & \cup\{q.q'.p' \mid q \in Pos(t),\ t(q) \in X,\ q' \in Pos(\sigma(t(q))), \\
 & & \qquad \sigma(t(q))(q') \in X, \\
 & & \qquad p' \in Pos(\sigma'(\sigma(t(q)))(q'))\}
\end{array}
$$

On en conclut que $Pos((t\sigma)\sigma') = Pos(t(\lambda x \in A.x))$.
Considérons l'ensemble des positions des deux termes substitués, pour lequel nous avons trois cas (correspondant à (1), (2) et (4)) :

a) $\{q \in Pos(t) \mid t(q) \notin X\}$

b) $\{q'.p', \mid q' \in Pos(t),\ t(q') \in X,\ p' \in Pos(\sigma(t(q'))),\ \sigma(t(q'))(p') \notin X\}$

c) $\{q'.p'.p \mid q' \in Pos(t),\ t(q') \in X,\ p' \in Pos(\sigma(t(q))),\ \sigma(t(q'))(p') \in X$
$p \in Pos(\sigma'(\sigma(t(q')))(p'))\}$

Considérons ces trois cas :

a) Comme $q \in Pos(t)$, alors $q \in Pos(t\sigma)$.
Comme $q \in Pos(t)$ et $t(q) \notin X$, alors $(t\sigma)(q) = t(q)$, et en particulier, $(t\sigma)(q) \notin X$.
Donc $((t\sigma)\sigma')(q) = (t\sigma)(q) = t(q)$.
Or, $(t(\lambda x \in A.\sigma(x)\sigma'))(q) = t(q)$.

b) $q' \in Pos(t),\ p' \in Pos(\sigma(t(q'))),\ t(q') \in X$, donc $\sigma(t(q'))(p') = t\sigma(q'.p') \notin X$.
Donc $(t\sigma)\sigma'(q'.p') = (t\sigma)\sigma'(q'.p') = t\sigma(q'.p')$.
Par ailleurs, $t(\lambda x \in A.\sigma(x)\sigma')(q'.p') = \sigma(t(q'))\sigma'(p')$.
Or, $\sigma(t(q'))(p') \in X$, donc $\sigma(t(q'))\sigma'(p') = \sigma(t(q'))(p') = t\sigma(q'.p')$.
Donc $(t\sigma)\sigma'(q'.p') = t(\lambda x \in A.\sigma(x)\sigma')(q'.p')$.

c) On a

$$
\begin{array}{rcl}
(t\sigma)\sigma'(q'.p'.p) & = & \sigma'(t\sigma(q'.p'))(p) \\
 & = & \sigma'(\sigma(t(q'))(p'))(p) \\
(t(\lambda x \in A.\sigma(x)\sigma'))(q'.p'.p) & = & (\lambda x \in A.\sigma(x)\sigma')(t(q'))(p'.p) \\
 & = & \sigma(t(q'))\sigma'(p'.p) \\
 & = & \sigma'(\sigma(t(q'))(p'))(p)
\end{array}
$$

Ainsi, $(t\sigma)\sigma' = t(\lambda x \in A.(\sigma(x))\sigma')$.

Ainsi, T_A est une famille de termes symboliques. Montrons que c'est une famille normale.
Soit $t \in T_A \cap T_B$ avec $A, B \subseteq X$. Le point 1. de la définition de T_A nous permet de le montrer. En effet, $t(p) \in F_0 \cup A$ si et seulement si p est une feuille de $Pos(t)$ et $t(p) \in F_0 \cup B$ si et seulement si p est une feuille de $Pos(t)$. Ainsi, $t(p) \in F_0 \cup (A \cap B)$ si et seulement si p est une feuille de $Pos(t)$. Donc $t \in T_{A \cap B}$. □

3.1.2.4 Les λ-termes typés et non typés modulo β ou $\beta\eta$

Nous allons tout d'abord définir les λ-termes typés, puis nous définirons les λ-termes non typés. Nous verrons que ces familles de termes sont des familles de termes symboliques. On pourra consulter [Lal90] et [Bar84] pour plus de détails.

On définit un ensemble de *types* T inductivement par :

1. Une collection de *variables de types* $\alpha, \gamma, \ldots$
2. Si σ et τ sont des types, alors :
   (a) $(\sigma \times \tau)$ est un type (qu'on appelle un type *produit*).
   (b) $(\sigma \to \tau)$ est un type (qu'on appelle un type *fonction*).

Définition – 3.1.7

> *L'ensemble des* λ-termes typés *sont générés comme suit. On écrit* $t : \sigma$ *pour signifier que* t *a le type* σ.
> *1. Une collection de* variables typées $x : \sigma, \ldots$
> *2. Si* $u : \sigma$ *et* $v : \tau$ *sont des* λ*-termes typés, alors le* produit $\langle u, v\rangle : \sigma \times \tau$ *est un* λ*-terme typé.*
> *3. Si* $t : \sigma \times \tau$ *est un* λ*-terme typé, alors* $\mathsf{fst}(t) : \sigma$ *et* $\mathsf{snd}(t) : \tau$ *sont des* λ*-termes typés.*
> *4. Si* $v : \tau$ *est un lambda-terme typé et* $x : \sigma$ *une variable, alors l'*abstraction $\lambda x : \sigma.v : \sigma \to \tau$ *est un* λ*-terme typé.*
> *5. Si* $t : \sigma \to \tau$ *et* $u : \sigma$ *sont des* λ*-termes typés, alors l'*application $(tu) : \tau$ *est un* λ*-terme typé.*

La version non typée, appelée le *λ-calcul non typé* ou plus simplement le *λ-calcul*, est la même que celle présentée ici, excepté le fait qu'elle admet plus de termes puisque la construction ne dépend pas des types.

Lorsque cela ne présentera pas d'ambiguïté, nous pourrons omettre le type dans l'écriture d'un λ-terme. Par exemple, nous écrirons $\lambda x.t$ pour désigner l'abstraction $\lambda x : \sigma.t$.

Les conventions suivantes sont adoptées :

- L'opérateur d'application associe à gauche, et on note les termes sans parenthèse ; par exemple, le terme $(((uv)t)w)$ devient $uvtw$, et le terme $(u(vt))$ devient $u(vt)$.
- On regroupe les abstractions successives sous un seul λ ; par exemple, le terme $\lambda x.\lambda y.\lambda z.t$ devient $\lambda xyz.t$.
- Il y a une notion de renommage qui nous permet d'écrire les termes en utilisant différents noms. Par exemple, $\lambda x : \sigma.x$ et $\lambda z : \sigma.z$ définissent la même fonction. Cette opération de renommage est connue sous le nom d'*α-conversion*, et n'est pas aussi triviale qu'elle pourrait paraître. Par exemple, un renommage de la variable x en y dans le terme $\lambda x.xy$ nous donnerait $\lambda y.yy$ qui ne désigne pas la même fonction que le terme précédent. Pour plus de détails sur cette notion, on pourra consulter les ouvrages précédemment cités [Bar84, Lal90].

Notons qu'un terme $\lambda y.yy$ n'est pas bien formé en tant que terme typé car y devrait alors avoir le type σ et $\sigma \to \sigma$, ce qui n'a pas de sens.

L'*ensemble des variables libres* d'un λ-terme typé (FV) est défini par :

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda x.t) &= FV(t) \backslash \{x\} \\
FV(tu) &= FV(\langle t, u\rangle) = FV(t) \cup FV(u) \\
FV(\mathsf{fst}(t)) &= FV(\mathsf{snd}(t)) = FV(t)
\end{aligned}
$$

L'*ensemble des variables liées* d'un λ-terme typé (BV) est défini par :

$$
\begin{aligned}
BV(x) &= \emptyset \\
BV(\lambda x.t) &= BV(t) \cup \{x\} \\
BV(tu) &= BV(\langle t, u\rangle) = BV(t) \cup BV(u) \\
BV(\mathsf{fst}(t)) &= BV(\mathsf{snd}(t)) = BV(t)
\end{aligned}
$$

L'ensemble des variables d'un λ-terme est l'union de l'ensemble de ses variables libres et de l'ensemble de ses variables liées.
On a la *règle de réduction* suivante pour le λ-calcul, qui définit une relation de réduction, notée $\rightarrow_\beta$, et appelée la *β-réduction* :

$$
\begin{aligned}
(\lambda x.t)u &\rightarrow_\beta t[u/x] \\
\mathsf{fst}(\langle t, u\rangle) &\rightarrow_\beta t \\
\mathsf{snd}(\langle t, u\rangle) &\rightarrow_\beta u
\end{aligned}
$$

où la *substitution* $t[u/x]$ est définie par :

$$
\begin{aligned}
x[v/x] &= v \\
y[v/x] &= y \\
(\lambda y.t)[v/x] &= \lambda y.(t[v/x]) \\
(tu)[v/x] &= (t[v/x])(u[v/x]) \\
\langle t, u\rangle[v/x] &= \langle t[v/x], u[v/x]\rangle \\
\mathsf{fst}(t)[v/x] &= \mathsf{fst}(t[v/x]) \\
\mathsf{snd}(t)[v/x] &= \mathsf{snd}(t[v/x])
\end{aligned}
$$

On adopte la convention qui établit le fait que toutes les variables liées sont différentes des variables libres. Ainsi, pour le troisième cas de la définition de la substitution, il n'y a pas d'occurrence de la variable y dans le terme v qui pourrait devenir liée. Ceci élimine le problème de l'α-conversion mentionné plus haut.

On a aussi la règle de réduction suivante qui définit la *η-réduction* notée $\rightarrow_\eta$:

$$
\begin{aligned}
\lambda x.tx &\rightarrow_\eta t \ (x \notin FV(t)) \\
\langle \mathsf{fst}(t), \mathsf{snd}(t)\rangle &\rightarrow_\eta t
\end{aligned}
$$

La notation $\rightarrow_{\beta\eta}$ est utilisée pour l'union des relations $\rightarrow_\beta$ et $\rightarrow_\eta$, mais nous écrirons simplement $\rightarrow$ lorsqu'il n'y aura pas d'ambiguïté. La notation $t \rightarrow u$ établit que que t est réécrit en u en une seule étape. La notation $\rightarrow^*$ est utilisée pour la fermeture transitive et réflexive de $\rightarrow$, et $\rightarrow^+$ représente une étape ou plus de réduction. On définit la notion de *conversion*, notée $=_\beta$, comme étant la fermeture symétrique de $\rightarrow^*_\beta$.

Un *contexte* $C[]$ est défini par :

$$
\begin{array}{lcl}
C[] & ::= & [] \\
 & | & tC[] \\
 & | & C[]t \\
 & | & \lambda x.C[] \\
 & | & \mathsf{fst}C[] \\
 & | & \mathsf{snd}C[] \\
 & | & \langle C[], t\rangle \\
 & | & \langle t, C[]\rangle
\end{array}
$$

La relation de réduction s'applique à un contexte quelconque, et on a le lemme suivant :

Lemme – 3.1.1

Si $t \to u$, alors pour tout contexte $C[]$, $C[t] \to C[u]$.

La démonstration de ce lemme se fait simplement par récurrence sur la structure des contextes.

Finalement, faisons une remarque sur les substitutions. Lorsque l'on écrit des termes avec des substitutions multiples, il faut être prudent. Le lemme suivant nous donne un résultat sur une façon de les réordonner.

Lemme – 3.1.2

(lemme de substitution) *Si x et y sont des variables distinctes et $x \notin FV(v)$, alors*

$$t[u/x][v/y] = t[v/y][u[v/y]/x]$$

La démonstration de ce lemme se fait assez simplement par récurrence sur la structure de t (consulter [Bar84], pages 27-28).

Le λ-calcul typé, non typé modulo β, et $\beta\eta$ étant défini, ainsi que les substitutions sur ces termes, nous allons maintenant pouvoir établir le lien avec les termes symboliques.

On définit l'ensemble des variables libres d'un terme modulo β ou $\beta\eta$ de la même façon que nous l'avons fait pour les théories équationnelles en section 3.1.2.2.

Proposition – 3.1.6

Le λ-calcul typé ou non typé, modulo β, $\beta\eta$, définit une famille normale de termes symboliques.

Démonstration.

Les deux premiers items de la définition des familles de termes symboliques ($A \subseteq T_A$ et si $A \subseteq B$ alors $T_A \subseteq T_B$) sont ici trivialement vérifiés.

Vérifions que les équations (3.1) à (3.4) de la définition des termes symboliques sont vraies pour les substitutions du λ-calcul typé ou non, modulo $\beta\eta$.

Notons T_A les λ-termes dont l'ensemble des variables libres est dans A.

Les deux premières (3.1) et (3.2) sont évidentes, et la troisième (3.3) qui définit la substitution vide (ou identité) aussi. La dernière (3.4) définissant la composition des substitutions est établie simplement de la manière suivante pour un cas particulier que l'on peut généraliser :
Soient $\sigma = [u/x]$ et $\sigma' = [v/y]$ tels que $x \notin FV(v)$.
On a donc $\sigma : \{x\} \to T_{FV(u)}$ et $\sigma' : \{y\} \to T_{FV(v)}$. Par (3.1) et (3.2), on a $\sigma : \{x\} \to T_{FV(u)\cup\{y\}}$ et $\sigma' : FV(u) \cup \{y\} \to T_{FV(v)}$, ce qu'on étend de manière naturelle à $\sigma : T_{\{x\}} \to T_{FV(u)\cup\{y\}}$ et $\sigma' : T_{FV(u)\cup\{y\}} \to T_{FV(v)}$.
On a alors $Subst_{FV(u)\cup\{y\},FV(v)}(\sigma')(Subst_{\{x\},FV(u)\cup\{y\}}(\sigma)(t)) = (t[u/x])[y/v]$
et $Subst_{\{x\},FV(v)}(\lambda z \in \{x\}.Subst_{FV(u)\cup\{y\},FV(v)}(\sigma')(\sigma(z)))(t) = t([u/x][y/v])$.
Or, le lemme de substitution nous permet d'établir que $(t[u/x])[y/v] = t([u/x][y/v]) = t[v/y][u[v/y]/x]$, donc l'équation (3.4) est vérifiée.

Montrons maintenant que c'est une famille normale. Comme nous l'avons montré en section 3.1.1, pour toute famille de termes symboliques, on a $T_{A\cap B} \subseteq T_A \cap T_B$. Il suffit donc de montrer que $T_A \cap T_B \subseteq T_{A\cap B}$. Si $t \in T_A \cap T_B$, alors $FV(t) \subseteq T_A \cap T_B$, donc $FV(t) \subseteq T_A$ et $FV(t) \subseteq T_B$, donc $FV(t) \subseteq A$ et $FV(t) \subseteq B$; en d'autres termes, $FV(t) \subseteq A \cap B$, donc $T_{FV(t)} \subseteq T_{A\cap B}$. Comme $t \in T_{FV(t)}$, alors $t \in T_{A\cap B}$. D'où la normalité de la famille de termes. □

3.1.3 Clauses symboliques

Nous introduisons ici la notion de clause symbolique, accompagnée de quelques définitions dont nous aurons besoin par la suite.

Fixons un ensemble infini dénombrable $\mathcal{P}$ de symboles de prédicats.

Définition – 3.1.8

Un atome symbolique *est un objet de la forme* $P(t_1, \ldots, t_n)$*, où* P *est un symbole de prédicat d'arité* n *et les* t_i*,* $1 \leq i \leq n$*, sont des termes symboliques.*

Définition – 3.1.9

Un littéral symbolique *est soit un littéral symbolique* positif $+P(t_1, \ldots, t_n)$*, soit un littéral symbolique* négatif $-P(t_1, \ldots, t_n)$.

Définition – 3.1.10

Une clause symbolique *est une disjonction finie de littéraux* $\pm_1 P_1(t_{1_1}, \ldots, t_{n_1}) \vee \ldots \vee \pm_k P_k(t_{1_k}, \ldots, t_{n_k})$*. Si* $k = 0$*, on écrit cette disjonction* $\Box$ *; il s'agit de la* clause vide.

Une *clause symbolique construite sur une famille de termes symboliques* $(T_A)_A$ est une clause symbolique de la forme $\bigvee_{i=1}^{k} \pm_i P_i(t_{1_i}, \ldots, t_{n_i})$ où pour tout i, j, $1 \leq i \leq k$, $1 \leq j \leq n$, t_{j_i} est un terme symbolique de la famille $(T_A)_A$.

Définition – 3.1.11

*L'*ensemble des variables libres d'un atome symbolique $P(t_1, \ldots, t_n)$ *est défini comme l'union des variables libres des* t_i*,* $1 \leq i \leq n$*, soit* $fv(P(t_1, \ldots, t_n)) = fv(t_1) \cup \ldots \cup fv(t_n)$.
*L'*ensemble des variables libres d'un littéral symbolique*,* $fv(\pm P(t_1, \ldots, t_n))$*, est l'ensemble des variables libres de l'atome symbolique, soit* $fv(P(t_1, \ldots, t_n))$.
*L'*ensemble des variables libres d'une clause symbolique*,* $fv(\bigvee_{i=1}^{k} \pm_i P_i(t_{1_i}, \ldots, t_{n_i}))$*, est l'union des ensembles de variables libres des littéraux symboliques* $\pm_i P_i(t_{1_i}, \ldots, t_{n_i})$*,* $1 \leq i \leq k$.

Définition – 3.1.12

Une instance d'un terme symbolique $t \in T_A$ *est tout terme symbolique* $Subst_{A,B}(\sigma)(t)$*, noté* $Subst_{A,B}(\sigma)t$*, pour tout* $B \subset \mathcal{V}$ *fini et pour tout* $\sigma : A \to T_B$.
Une instance d'un atome symbolique $P(t_1, \ldots, t_n)$ *est tout atome de la forme* $P(Subst_{A,B}(\sigma)(t_1), \ldots, Subst_{A,B}(\sigma)(t_n))$*, noté* $Subst_{A,B}(\sigma)P(t_1, \ldots, t_n)$*, pour tout* A *contenant les variables libres de* $P(t_1, \ldots, t_n)$*, pour tout* $B \subset \mathcal{V}$ *fini, et pour tout* $\sigma : A \to T_B$.
Une instance d'une clause symbolique $C = \bigvee_{i=1}^{k} \pm_i P_i(t_{1_i}, \ldots, t_{n_i})$*, notée* $Subst_{A,B}(\sigma)C$*, est toute clause de la forme* $\bigvee_{i=1}^{k} \pm_i P_i(Subst_{A,B}(\sigma)(t_{1_i}), \ldots, Subst_{A,B}(\sigma)(t_{n_i}))$*, pour tout* A *contenant les variables libres de* C*, pour tout* $B \subset \mathcal{V}$ *fini, et pour tout* $\sigma : A \to T_B$.
Une instance close *d'une clause symbolique* C*, est une instance* $Subst_{A,\emptyset}(\sigma)C$ *telle que l'ensemble de ses variables libres est vide.*

3.1.4 Unification et ordre

Afin de pouvoir procéder à la résolution abstraite ordonnée avec sélection telle que nous allons la définir, nous avons besoin d'introduire un certain nombre de notions.

3.1.4.1 Unification de termes symboliques

En particulier, nous aurons besoin de résoudre des équations sur l'ensemble des termes symboliques, c'est-à-dire trouver tous les $B, \sigma : A \to T_B$ tels que $Subst_{A,B}(\sigma)(s) = Subst_{A,B}(\sigma)(t)$. C'est ce que nous allons définir avec la notion d'unification de termes symboliques.

Définition – 3.1.13

Toute application $\sigma : A \to T_B$*, pour tout* $A, B \subset \mathcal{V}$*, est appelée une* substitution symbolique.

Définition – 3.1.14

Un unificateur *de* s *et* t*,* $s, t \in T_A$*, est une paire* (B, σ) *où* $B \subseteq \mathcal{V}$ *et* $\sigma : A \to T_B$ *est telle que* $Subst_{A,B}(\sigma)(t) = Subst_{A,B}(\sigma)(s)$.

Par abus de langage, on notera un tel unificateur σ au lieu (B, σ).

Définition – 3.1.15

> *Soient s et t deux termes symboliques appartenant à T_A, soient $\sigma : A \to T_B$ et $\sigma' : A \to T_B$ deux unificateurs de s et t. On dit que σ est* plus général *que σ' si et seulement si :*
> $\exists\theta : B \to T_B.\forall u \in T_B.\ Subst_{A,B}(\sigma')(u) = Subst_{B,B}(\theta)(Subst_{A,B}(\sigma)(u))$.
> *Un* unificateur le plus général*, ou* mgu *("most general unifier"), de s et t est un unificateur de s et t qui est plus général que tout autre unificateur de s et t.*

Notons que, dans cette définition, les deux unificateurs σ et σ' sont tous deux à valeurs dans T_B. Nous pouvons en effet faire cette supposition : si σ est à valeurs dans T_{B_1} et σ' à valeurs dans T_{B_2}, alors σ et σ' sont à valeurs dans T_B avec $B = B_1 \cup B_2$.

Définition – 3.1.16

> *Soient s et t deux termes symboliques appartenant à T_A. Un* ensemble complet d'unificateurs *de s et t, noté $\mathsf{csu}(s \doteq t)$ ("complete set of unifiers") est un ensemble d'unificateurs $\sigma : A \to T_B$ tels que, pour tout $C \in \mathcal{V}$ et pour tout $\sigma' : A \to T_C$ tels que $Subst_{A,C}(\sigma')(t) = Subst_{A,C}(\sigma')(s)$, il existe σ dans $\mathsf{csu}(s \doteq t)$ tel que σ est plus général que σ'. On notera aussi $\mathsf{csu}(t \doteq t_1, \ldots, t \doteq t_n)$ un ensemble complet d'unificateurs unifiant simultanément les termes symboliques t et t_i pour tout i tels que $1 \leq i \leq n$.*

En pratique, il n'y a pas nécessairement d'unificateur le plus général pour s et t, même quand ceux-ci sont unifiables, ce qui est le cas par exemple de l'unification en λ-calcul modulo β ou $\beta\eta$. Si par contre il existe, on a une théorie *0-aire*. Les théories *infinitaires* sont celles pour lesquelles il existe un ensemble complet d'unificateurs qui est minimal pour l'inclusion mais infini en général. Par exemple, l'unification modulo la théorie équationnelle de l'associativité donne lieu à des csu infinis. Les théories *finitaires* sont celles pour lesquelles $\mathsf{csu}(s \doteq t)$ est fini ; par exemple, l'unification modulo la théorie équationnelle de l'associativité et de la commutativité produit des csu finis. Les théories *unitaires* sont telles que $\mathsf{csu}(s \doteq t)$ a au plus un élément ; un exemple est donné avec l'unification des termes du premier ordre. Pour plus de détails sur l'unification, on pourra se référer à [JK90].

On peut remarquer que l'approche par csu n'est pas aussi générale qu'on le voudrait. En λ-calcul modulo β ou $\beta\eta$, une approche par contraintes est plus efficace [Hue73]. On pourra se référer à [CDJK99] pour un aperçu de la résolution par contraintes, où on trouve notamment un algorithme dit d'*unification par contraintes* pour les termes du premier ordre. Nous ne nous attacherons cependant pas ici à cet aspect du problème, et considérerons l'approche par unification telle que nous venons de la présenter.

3.1.4.2 Ordre sur les atomes symboliques

Nous aurons besoin aussi pour la résolution ordonnée avec sélection (et comme son nom le laisse à penser) d'un ordre sur les atomes symboliques.

Supposons donc que nous disposons d'un ordre strict $\succ$ sur les atomes.

Définition – 3.1.17

> *On dit qu'un ordre $\succ$ est* stable *si et seulement si pour tout $A \in \mathcal{V}$, pour toute substitution symbolique $\sigma : A \to T_\emptyset$, pour tous atomes $P_1(t_1, \ldots, t_n)$, $P_2(s_1, \ldots, s_m)$, si $P_1(t_1, \ldots, t_n) \succ P_2(s_1, \ldots, s_m)$, alors $Subst_{A,\emptyset}(\sigma)P_1(t_1, \ldots, t_n) \succ Subst_{A,\emptyset}(\sigma)P_2(s_1, \ldots, s_m)$.*

3.2 Sémantique

3.2.1 Sémantique à la Tarski

Nous allons ici introduire une sémantique inspirée de la sémantique de Tarski (voir [GLM97], page 73).

Nous appellerons *domaine* un ensemble non vide.

Définition – 3.2.1

> *Un* environnement *$\rho : A \to D$ est une application qui associe à chaque variable de A un élément d'un ensemble non vide, un* domaine*, D. L'ensemble A est alors noté $\mathsf{dom}(\rho)$.*

Une structure I est la donnée d'un domaine D, de sous-ensembles I_P de D, un pour chaque prédicat $P \in \mathcal{P}$, et d'une fonction, appelée *interprétation de termes symboliques*, $[\![\_]\!]_A\_ : T_A \times (A \to D) \to D$, telle que :

$$[\![Subst_{A,B}(\sigma)(t)]\!]_B\rho = [\![t]\!]_A(\lambda x \in A.[\![\sigma(x)]\!]_B\rho) \tag{3.5}$$

$$\forall A \forall t \in T_A, \forall B \supseteq A.[\![t]\!]_B\rho = [\![t]\!]_A\rho_{|A} \tag{3.6}$$

Par la suite, lorsque t est un terme symbolique clos, on notera $[\![t]\!]$ au lieu de $[\![t]\!]_{\rho_0}$, où ρ_0 est l'unique environnement tel que $\mathsf{dom}(\rho_0) = \emptyset$. On notera $P(\widetilde{t^n})$ le prédicat $P(t_1, \ldots, t_n)$, et on écrira $fv(\widetilde{t^n})$ l'union des ensembles $fv(t_i)$ pour tout i, $1 \leq i \leq n$.
On définit la relation $I, \rho \models_A P(\widetilde{t^n})$, et l'on dit que $P(\widetilde{t^n})$ est *vraie* dans I, ρ si et seulement si pour tout i, $1 \leq i \leq n$, $[\![t_i]\!]_A\rho$ est dans I_P.

On peut cependant remarquer que la relation $I, \rho \models_A P(\widetilde{t^n})$ est indépendante de A si $(T_A)_A$ est une famille normale. En effet, pour tout $\rho : A \to D$, pour tout $A' \subseteq A$ et pour tout $\widetilde{t^n} \in T_{A'}$, on a $I, \rho \models_A P(\widetilde{t^n})$ si et seulement si $I, \rho_{|_{A'}} \models_{A'} P(\widetilde{t^n})$. De plus, les familles de termes symboliques considérées étant normales, on a $I, \rho \models P(\widetilde{t^n})$ si et seulement si $I, \rho_{|_{fv(\widetilde{t^n})}} \models_{fv(\widetilde{t^n})} P(\widetilde{t^n})$. On écrira donc $I, \rho \models P(\widetilde{t^n})$ au lieu de $I, \rho \models_A P(\widetilde{t^n})$.

Pour toute clause C, on pose $I, \rho \models C$ si et seulement s'il existe un littéral $+P(\widetilde{t^n})$ dans C tel que $I, \rho \models P(\widetilde{t^n})$, ou un littéral $-P(\widetilde{t^n})$ dans C tel que $I, \rho \not\models P(\widetilde{t^n})$.

La structure I est un *modèle* de la clause C si et seulement si $I, \rho \models C$ pour tout environnement ρ tel que $fv(C) \subseteq \mathsf{dom}(\rho)$; on écrit alors $I \models C$. I est un modèle d'un ensemble S de clauses si et seulement si $I \models C$ pour toute clause C de S ; on écrit $I \models S$ dans ce cas.

On dit qu'un ensemble de clauses S est *satisfiable* si et seulement s'il a un modèle.

3.2.2 Sémantique à la Herbrand

Définition – 3.2.2

> *L'ensemble des* termes symboliques clos *est l'ensemble $T_\emptyset$.*
> *L'univers de Herbrand est l'ensemble $T_\emptyset$ de tous les termes clos.*

Lemme – 3.2.1

> *Si $T_\emptyset \neq \emptyset$, alors la fonction $[\![t]\!]_A\rho \hat{=} Subst_{A,\emptyset}(\rho)(t)$ de $T_A \times (A \to T_\emptyset)$ dans $T_\emptyset$ définit une interprétation de termes symboliques.*

Démonstration. Il suffit de montrer que la fonction ainsi définie vérifie les équations (3.5) et (3.6).
Supposons que $T_\emptyset \neq \emptyset$. Soient $A, B \subseteq \mathcal{V}$.

$$\begin{array}{rcll}
[\![Subst_{A,B}(\sigma)(t)]\!]_B\rho & = & Subst_{B,\emptyset}(\rho)(Subst_{A,B}(\sigma)(t)) & \text{par définition} \\
& = & Subst_{A,\emptyset}(\lambda x \in A.Subst_{B,\emptyset}(\rho)(\sigma(x)))(t) & \text{par (3.4)} \\
& = & Subst_{A,\emptyset}(\lambda x \in A.[\![\sigma(x)]\!]_B\rho)(t) & \text{par définition} \\
& = & [\![t]\!]_A(\lambda x \in A.[\![\sigma(x)]\!]_B\rho) & \text{par définition.}
\end{array}$$

Ainsi, (3.5) est vérifiée.

Soient $A \subseteq \mathcal{V}$, $t \in T_A$ et B quelconque tel que $A \subseteq B$. Alors :

$$\begin{array}{rcll}
[\![t]\!]_B\rho & = & Subst_{B,\emptyset}(\rho)(t) & \text{par définition} \\
& = & Subst_{A,\emptyset}(\rho_{|A})(t) & \text{par (3.1)} \\
& = & [\![t]\!]_A\rho_{|A} & \text{par définition.}
\end{array}$$

Ainsi, (3.6) est vérifiée. □

Définition – 3.2.3

> *On appellera* structure de Herbrand *toute interprétation de termes symboliques telle que, si $T_\emptyset \neq \emptyset$, alors :*
>
> $$[\![t]\!]_A\rho = Subst_{A,\emptyset}(\rho)(t) \tag{3.7}$$

Par la suite, on parlera indifféremment de structure de Herbrand ou d'*interprétation de Herbrand*. Un *modèle de Herbrand* d'un ensemble de clauses symboliques S est une structure de Herbrand qui est un modèle de S.

Proposition – 3.2.1

> *Supposons que $(T_A)_A$ est une famille normale de termes symboliques telle que $T_\emptyset \neq \emptyset$. Soit S un ensemble de clauses symboliques sur $(T_A)_A$.*
> *Alors, S est satisfiable si et seulement s'il a un modèle de Herbrand.*

Démonstration. Si S a un modèle de Herbrand, S est clairement satisfiable. Réciproquement, supposons que S est satisfiable, et soit I un modèle de S. Par normalité de la famille de termes symboliques, l'ensemble des variables libres d'une clause symbolique existe. Notons alors A l'union des variables libres de toutes les clauses de S. Par le lemme 3.2.1, il suffit de définir I'_P pour chaque symbole de prédicat P afin d'obtenir une structure de Herbrand. Soit $I'_P = \{u \in T_\emptyset \mid [\![u]\!]\rho_0 \in I_P\}$, où ρ_0 est l'unique fonction de $\emptyset$ vers $T_\emptyset$. Comme I est un modèle de S, pour toute clause symbolique C de S, pour tout environnement $\rho : A \to D$ tel que $A \supseteq fv(C)$, on a $I, \rho \models C$.

Fixons $\sigma : A \to T_\emptyset$ quelconque, et soit $\rho = \lambda x \in A.[\![\sigma(x)]\!]_\emptyset\rho_0$. Comme $I, \rho \models C$, il existe un littéral symbolique $+P(\widetilde{t^n})$ dans C tel que pour tout i, $1 \leq i \leq n$, $[\![t_i]\!]_A\rho \in I_P$ ou un littéral symbolique $-P(\widetilde{t^n})$ dans C tel que pour tout i, $1 \leq i \leq n$, $[\![t_i]\!]_A\rho \notin I_P$. Or, par (3.5) et par définition de ρ, on a $[\![t_i]\!]_A\rho = [\![Subst_{A,\emptyset}(\sigma)(t_i)]\!]_\emptyset\rho_0$ pour tout i, $1 \leq i \leq n$. Donc pour tout i

$$\begin{array}{rcll}
[\![t_i]\!]_A\rho \in I_P & \iff & [\![Subst_{A,\emptyset}(\sigma)(t_i)]\!]_\emptyset\rho_0 \in I_P & \\
& \iff & Subst_{A,\emptyset}(\sigma)(t_i) \in I'_P & \text{par définition de } I'_P \\
& \iff & [\![t_i]\!]_A\sigma \in I'_P & \text{par (3.7)} \\
& \iff & I', \sigma \models t_i. &
\end{array}$$

Donc $I', \sigma \models C$ pour toute clause symbolique C de S. Donc S a un modèle de Herbrand. □

Remarque 2 *Une structure de Herbrand I peut être caractérisée d'une autre façon par un ensemble d'atomes symboliques clos : à savoir les atomes symboliques clos $P(\widetilde{t^n})$ tels que $I \models P(\widetilde{t^n})$ (comme les termes symboliques t_i sont clos, $[\![t_i]\!]\rho$ est indépendant de ρ qui est donc superflu, donc omis). Réciproquement, si E est un ensemble d'atomes symboliques clos, on peut définir l'interprétation de Herbrand correspondante I par $I_P = \{t_i \in T_\emptyset \mid 1 \leq i \leq n \text{ et } P(\widetilde{t^n}) \in E\}$. On considérera dans la suite, sans le dire explicitement, qu'une structure de Herbrand est un ensemble d'atomes symboliques clos.*

3.2.3 Clauses de Horn et plus petits modèles

Rappelons qu'une *clause de Horn* est une clause ayant au plus un littéral positif, comme cela a été défini au premier ordre en section 1.1. Il en va de même pour les clauses symboliques.

Proposition – 3.2.2

> *Soit $(T_A)_A$ une famille de termes symboliques telle que $T_\emptyset \neq \emptyset$, et soit S un ensemble de clauses symboliques de Horn sur $(T_A)_A$.*
> *Alors, si S est satisfiable, il a un plus petit modèle de Herbrand.*

Démonstration. Les structures de Herbrand peuvent être ordonnées par inclusion $\subseteq$.

Il suffit de montrer que l'intersection des modèles de Herbrand est encore un modèle de Herbrand. Ce sera alors le plus petit pour l'ordre d'inclusion.

S étant satisfiable, par la proposition 3.2.1, il a un modèle de Herbrand.

Soit H_0 l'intersection de tous les modèles de Herbrand de S. Soit H un modèle quelconque de Herbrand qui satisfait S, et soit C clause symbolique quelconque de S. Soit $A \subseteq \mathcal{V}$, et soit $\sigma : A \to T_\emptyset$. Si $\pm L$ est un littéral symbolique, on notera $Subst_{A,\emptyset}(\sigma)L$ une instance close quelconque de L.

Or $H \models C$ si et seulement s'il existe un littéral symbolique $+L$ dans C tel que $Subst_{A,\emptyset}(\sigma)L$ est dans H ou il existe un littéral symbolique $-L$ dans C tel que $Subst_{A,\emptyset}(\sigma)L$ n'est pas dans H. Autrement dit, $H \models C$ si et seulement s'il existe un littéral symbolique $+L$ dans C tel que $Subst_{A,\emptyset}(\sigma)L$ est dans H ou pour tout littéral $-L$ dans C, $Subst_{A,\emptyset}(\sigma)L$ est dans H. Or, C étant une clause de Horn, elle a au plus un littéral positif.

Si c'est une clause négative, pour tout H satisfaisant S, $H \models C$ si et seulement si tous les littéraux $-L$ de C sont tels que $Subst_{A,\emptyset}(\sigma)L$ est dans H, donc en particulier dans H_0.

Si C possède un littéral positif $+L$, pour tout H satisfaisant S, $H \models C$ si et seulement si $Subst_{A,\emptyset}(\sigma)L$ est dans H, donc en particulier dans H_0, ou pour tout littéral $-L$ dans C, $Subst_{A,\emptyset}(\sigma)L$ est dans H, donc dans H_0.

Donc H_0 est un modèle de S. □

Ceci n'est en général pas vrai pour un ensemble de clauses qui n'est pas de Horn. Effectivement, dans le cas d'une clause $-L \vee +L_1 \vee L_2$ ayant plus d'un littéral positif, si elle est satisfiable dans un modèle H_1 avec une instance close de L_1 appartenant au modèle, elle peut être satisfiable dans un modèle H_2 avec une instance close de L_2 dans H_2. Dans ce cas, $H_1 \cap H_2$ n'est pas forcément un modèle de la clause.

Une conséquence immédiate est que tout ensemble S de clauses symboliques définies, où toutes les clauses de S sont construites sur une famille normale de termes symboliques dont l'ensemble des termes clos est non vide, a un plus petit modèle. En effet, S a un modèle, à savoir celui qui contient tous les atomes symboliques clos.

3.2.4 Langages et reconnaissabilité

Nous avons donné un certain nombre de résultats pour la logique du premier ordre en section 1.7.1 à ce sujet. Nous allons ici généraliser ces résultats aux clauses symboliques de Horn.

Soit S un ensemble satisfiable de clauses symboliques de Horn, et soit P un symbole de prédicat. Rappelons que le *langage* $L_P(S)$ de S à l'*état* P est l'ensemble des termes symboliques clos t tels que $P(t)$ est dans le plus petit modèle de Herbrand de S. Les éléments de $L_P(S)$ sont appelés les termes symboliques *reconnus à* P par S. Par abus de langage, on dira que P est *vide* dans S si et seulement si $L_P(S)$ est vide, et de même pour toute autre propriété.

Le premier lemme caractérise la reconnaissabilité.

Lemme – 3.2.2

> *Soit S un ensemble satisfiable de clauses symboliques de Horn. Le terme symbolique clos t est reconnu à P par S si et seulement si S plus la clause symbolique $\perp \Leftarrow P(t)$ est insatisfiable.*

Démonstration. Si $t \in L_P(S)$, alors par définition $P(t)$ est dans le plus petit modèle de Herbrand de S, donc $S \cup \{\perp \Leftarrow P(t)\}$ est insatisfiable.

Réciproquement, si $S \cup \{\perp \Leftarrow P(t)\}$ est insatisfiable, alors aucun modèle de S ne doit satisfaire $\perp \Leftarrow P(t)$, et doit donc contenir $P(t)$; donc $t \in L_P(S)$. □

Le deuxième lemme caractérise la vacuité.

Lemme – 3.2.3

> *Étant donné un ensemble S satisfiable de clauses symboliques de Horn, P est vide dans S si et seulement si $S \cup \{\bot \Leftarrow P(x)\}$ est satisfiable.*

Démonstration. Si P est vide, alors le plus petit modèle de Herbrand de S ne contient pas d'atome symbolique clos de la forme $P(t)$, donc satisfait $\bot \Leftarrow P(t)$.

Réciproquement, si $S \cup \{\bot \Leftarrow P(x)\}$ est satisfiable, alors son plus petit modèle de Herbrand ne contient pas d'atome symbolique clos de la forme $P(t)$. Comme tout modèle de $S \cup \{\bot \Leftarrow P(x)\}$ est aussi un modèle de S, le plus petit modèle de Herbrand de S est inclus dans celui de $S \cup \{\bot \Leftarrow P(x)\}$, donc ne contient pas d'atome symbolique clos de la forme $P(t)$ non plus ; c'est-à-dire que P est vide dans S. □

Le troisième lemme caractérise la *vacuité d'intersection*, c'est-à-dire, étant donné un nombre fini de symboles de prédicats $P_1, \ldots, P_n$, la question de la vacuité de $L_{P_1}(S) \cap \ldots \cap L_{P_n}(S)$ (on dira par abus de langage, que l'intersection de $P_1, \ldots, P_n$ est vide dans ce cas).

Lemme – 3.2.4

> *Étant donné un ensemble S satisfiable de clauses symbolique de Horn, l'intersection de $P_1, \ldots, P_n$ est vide dans S si et seulement si $S \cup \{\bot \Leftarrow P_1(x), \ldots, P_n(x)\}$ est satisfiable.*

Démonstration. Si l'intersection de $P_1, \ldots, P_n$ est vide dans S, alors le plus petit modèle de Herbrand de S ne contient pas d'atome symbolique clos de la forme $P_1(t), \ldots, P_n(t)$, donc satisfait $\bot \Leftarrow P_1(t), \ldots, P_n(t)$.

Réciproquement, si $S \cup \{\bot \Leftarrow P_1(x), \ldots, P_n(x)\}$ est satisfiable, alors son plus petit modèle de Herbrand ne contient pas tous les atomes symboliques clos de la forme $P_1(t), \ldots, P_n(t)$ pour aucun terme symbolique clos t. Comme tout modèle de $S \cup \{\bot \Leftarrow P_1(x), \ldots, P_n(x)\}$ est aussi un modèle de S, le plus petit modèle de Herbrand de S est inclus dans celui de $S \cup \{\bot \Leftarrow P_1(x), \ldots, P_n(x)\}$, donc ne contient pas non plus tous les atomes symboliques clos de la forme $P_1(t), \ldots, P_n(t)$; c'est-à-dire que l'intersection de $P_1, \ldots, P_n$ est vide. □

Nous avons défini dans les sections précédentes (3.1 et 3.2) les notions de termes et de clauses symboliques, d'unification sur ces termes, et établi un certain nombre de résultats s'y rapportant. Nous allons pouvoir définir dans la section suivante la résolution abstraite ordonnée avec sélection.

3.3 Résolution abstraite ordonnée avec sélection

Nous avons présenté en section 1.3 la résolution ordonnée avec sélection dans le cadre de la logique du premier ordre. Nous allons ici étendre cette notion à des ensembles de clauses symboliques.

Les familles de termes symboliques $(T_A)_A$ que nous utiliserons par la suite sont supposées normales. De plus, l'ensemble des termes symboliques clos, $T_\emptyset$, sera supposé non vide.

Rappelons que nous avons défini la notion d'ensemble complet d'unificateurs pour les termes symboliques (noté csu), et supposé qu'il existe un ordre strict stable $\succ$ sur les atomes symboliques (section 3.1.4). On suppose de plus que $\mathsf{csu}(t \doteq t_1, \ldots, t \doteq t_n)$, pour tous termes symboliques $t, t_1, \ldots, t_n$, est fini et calculable.

Définition – 3.3.1

> *Une* fonction de sélection *est une application* sel *définie sur les clauses symboliques telle que si C est une clause symbolique,* $\mathsf{sel}(C)$ *est un ensemble quelconque possiblement vide de littéraux symboliques négatifs de C.*

La figure 3.1 présente la règle de *résolution abstraite ordonnée avec sélection* telle qu'elle est introduite dans [BG01a] dans le cas particulier du premier ordre. La règle demande en particulier qu'aucun littéral ne soit sélectionné dans les clauses symboliques de gauche. Si la clause symbolique de droite a une sélection vide, il s'agit alors simplement d'une résolution ordonnée entre ces clauses symboliques, ceci assurant une certaine équité dans le choix des clauses symboliques à résoudre, et assure en particulier que les clauses symboliques dont la sélection est vide ne soient pas laissées de côté dans le processus de résolution.

Définition – 3.3.2

> *On dira alors qu'un atome A est maximal dans une clause symbolique C s'il n'existe pas de littéral $\pm B$ dans C tel que $B \succ A$.*

$$\frac{C_1 \vee +A_{11} \vee \ldots \vee +A_{1k_1} \ldots C_n \vee +A_{1n} \ldots \vee +A_{nk_n} \qquad -A_1 \vee \ldots \vee -A_n \vee D}{C_1\sigma \vee \ldots \vee C_n\sigma \vee D\sigma}$$

où :

(i) $\forall i,\ 1 \leq i \leq n,\ \sigma \in \mathsf{csu}(A_{i1} \doteq A_i, \ldots, A_{ik_i} \doteq A_i)$;

(ii) soit $A_1, \ldots, A_n$ sont sélectionnés dans $-A_1 \vee \ldots \vee -A_n \vee D$, ou alors rien n'est sélectionné, $n = 1$, et A_1 est maximal dans D ;

(iii) chaque atome A_{ii} est maximal dans C_i ;

(iv) aucune clause $C_i \vee -A_{i1} \ldots \vee -A_{ik_i}$ ne contient de littéral sélectionné.

Fig. 3.1: Résolution abstraite ordonnée avec fonction de sélection

Remarque 3 *Par normalité de l'ensemble des termes symboliques, pour toute clause symbolique C, l'ensemble $fv(C)$ existe. L'instance close $Subst_{A,\emptyset}(\sigma)C$ de C, pour $A \subseteq \mathcal{V}$ et $\sigma : A \to \emptyset$, est donc égale, par (3.1), à $Subst_{fv(C),\emptyset}(\sigma)C$. Ainsi, $fv(C)$ et $\emptyset$ deviennent superflus dans la notation, nous pouvons donc les omettre et établir la définition ci-dessous.*

Notation – 3.3.1

> *Soit C une clause symbolique. Par la remarque précédente, nous pouvons établir la notation suivante : une instance close de C sera écrite $C\sigma$.*

Nous allons établir un théorème de complétude pour la résolution abstraite ordonnée avec sélection, mais avant cela nous allons montrer un résultat intermédiaire important, le théorème de Herbrand pour des ensembles de clauses symboliques.

Théorème – 3.3.1

> **(Herbrand)** *Étant donné un ensemble S de clauses symboliques, les trois conditions suivantes sont équivalentes :*
>
> 1. *S est insatisfiable ;*
> 2. *l'ensemble $S\downarrow$ des instances closes de S est insatisfiable ;*
> 3. *il existe un ensemble fini S_0 d'instances closes de S qui est insatisfiable.*

Nous pouvons démontrer ce théorème dans un cadre général où l'ensemble des termes symboliques n'est pas dénombrable. Cependant, toutes les familles de termes que nous considérerons en pratique seront dénombrables. De plus, les notions introduites dans le cas d'ensembles dénombrables nous seront très utiles par la suite.

C'est pourquoi nous allons donner deux démonstrations. La première supposera que l'ensemble de termes symboliques est dénombrable, et la deuxième non.

Démonstration 1. Introduisons tout d'abord quelques notions qui nous serons utiles.

Soit $A_1, \ldots, A_i, \ldots$ une énumération des atomes symboliques clos (elle existe car l'ensemble des termes symboliques est dénombrable). Une *interprétation partielle* sur cette énumération est une liste finie $\pm_1 A_1, \pm_2 A_2 \ldots, \pm_k A_k$. Si A_i apparaît avec le signe $+$, on dit que A_i est *vrai* dans l'interprétation partielle ; il est *faux* s'il apparaît avec le signe $-$. Sinon A_i est *indéfini* dans l'interprétation partielle.

L'*arbre de Herbrand* est l'arbre dont les sommets sont les interprétations partielles. L'interprétation partielle $\pm_1 A_1, \ldots, \pm_k A_k$ a deux fils, qui sont $\pm_1 A_1, \ldots, \pm_k A_k, -A_{k+1}$ et $\pm_1 A_1, \ldots, \pm_k A_k, +A_{k+1}$ (ceci bien sûr si A_{k+1} existe, sinon l'interprétation partielle n'a pas de fils). L'interprétation partielle ϵ à la racine de l'arbre est l'interprétation partielle vide.

Les branches maximales (infinies en général) de l'arbre de Herbrand sont naturellement en bijection avec les interprétations de Herbrand. Si I est une interprétation de Herbrand, on obtient une branche maximale qui passe par les nœuds ϵ, puis $\pm_1 A_1$, puis $\pm_2 A_2$, etc., où $\pm_i$ est le signe $+$ si $A_i \in I$, $-$ sinon. Réciproquement, sur toute branche maximale, tout atome symbolique A_i apparaît avec un signe unique $\pm_i$ dans toute interprétation partielle de longueur au moins i ; on obtient donc une interprétation de Herbrand contenant exactement les A_i tels que $\pm_i$ est le signe $+$.

Venons en maintenant à la démonstration proprement dite. Pour plus de clarté, nous présentons une démonstration dans laquelle les prédicats considérés sont unaires, mais elle est facilement généralisable au cas n-aire.

Supposons S satisfiable, et I un modèle de Herbrand de S (cf. proposition 3.2.1). Alors, pour toute instance close $C\rho$ d'une clause symbolique C de S, comme $I, \rho \models C$, il existe un littéral symbolique $+P(t)$ dans C tel que $[\![t]\!]_A \rho$ (pour tout $A \supseteq fv(t)$) est dans I_P ou un littéral symbolique $-P(t)$ dans C tel que $[\![t]\!]_A \rho$ (pour tout $A \supseteq fv(t)$) n'est pas dans I_P. Or, par l'équation (3.7), $[\![t]\!]_A \rho \in I_P$ si et seulement si $Subst_{A,\emptyset}(\rho)(t)$, donc on a un littéral $+P(Subst_{A,\emptyset}(\rho)(t))$ dans $C\rho$ tel que $Subst_{A,\emptyset}(\rho)(t)$ est dans I_P ou bien un littéral $-P(Subst_{A,\emptyset}(\rho)(t))$ dans $C\rho$ tel que $Subst_{A,\emptyset}(\rho)(t)$ n'est pas dans I_P. Ainsi, $I \models C\rho$. Donc I satisfait $S \downarrow$. Donc 2 implique 1.

Réciproquement, si I est une interprétation de Herbrand qui satisfait $S \downarrow$, par l'équation (3.7) de nouveau, I satisfait S. Donc 1 implique 2.

Clairement, 3 implique 2. Il ne reste qu'à démontrer que 2 implique 3. Disons qu'une clause symbolique close C est *fausse* au nœud $N = \pm_1 A_1, \pm_2 A_2, \ldots, \pm_k A_k$ si et seulement si pour tout littéral $\pm A$ de C, le littéral opposé $\mp A$ est dans la liste N. On remarque que si C est fausse au nœud N, alors C est fausse dans toute interprétation passant par le nœud N. Réciproquement, si C est fausse dans l'interprétation I, alors il existe un nœud N où C est fausse : si l'on pose $C = \pm_1 A_{i_1}, \ldots, \pm_m A_{i_m}$, $i_1 < \ldots < i_m$, alors on peut prendre pour N n'importe quel nœud qui contient $\mp_1 A_{i_1}, \ldots, \mp_m A_{i_m}$.

On remarque maintenant que, pour toute interprétation de Herbrand I, par 2, il existe une instance close $C\sigma$ d'une clause symbolique C de S qui est fausse dans I. Par la remarque ci-dessus, il existe alors un nœud N de l'arbre sémantique qui rend fausse $C\sigma$. Appelons *nœud d'échec* pour S tout nœud minimal rendant fausse au moins une instance close $C\sigma$, d'une clause symbolique C de S (on entend minimal pour la longueur du nœud vu comme liste). L'*arbre sémantique clos* est le sous-arbre de l'arbre sémantique de même racine ϵ, et dont les feuilles sont les nœuds d'échec. L'arbre sémantique clos n'a pas de branche infinie, sinon cette branche définirait une interprétation I pour laquelle au moins une instance close $C\sigma$ d'une clause symbolique C de S serait fausse, menant à l'existence d'un nœud d'échec sur la branche I, ce qui serait absurde. D'autre part, l'arbre sémantique clos est à branchement fini, donc, d'après le lemme de König, il est fini (le lemme de König nous dit que tout arbre à branchement fini dont toutes les branches sont fines est fini). En particulier, il a un nombre fini de feuilles, c'est-à-dire de nœuds d'échec. Choisissant pour chaque nœud d'échec N une instance close $C_N\theta_N$ d'une clause symbolique C_N de S qui est fausse en N, on obtient que l'ensemble des $C_N\theta_N$, est insatisfiable. □

Démonstration 2. Dans le cas non dénombrable, nous pouvons garder une partie de la preuve ci-dessus ($2 \Rightarrow 1$, $1 \Rightarrow 2$ et $3 \Rightarrow 2$). Nous allons montrer ici l'implication $1 \Rightarrow 3$, en utilisant le théorème de Tychonoff (le produit d'une famille quelconque d'espaces topologiques compacts est compact).

Équipons l'ensemble des valeurs de vérité $\mathbb{B} = \{\bot, \top\}$ de la topologie discrète (celle pour laquelle tout sous-ensemble est ouvert). $\mathbb{B}$ est alors compact.

Munissons l'espace $\mathbb{B}^{\mathcal{A}}$ (où $\mathcal{A}$ est l'ensemble des atomes symboliques clos) des interprétations de Herbrand de la topologie produit. Pour rappel, la topologie produit est la plus petite topologie telle que $\forall A \in \mathcal{A}.\forall O$ ouvert (respectivement fermé) de $\mathbb{B}$, on a $\{I \in \mathbb{B}^{\mathcal{A}} \mid I(A) \in O\}$ est ouvert (respectivement fermé).

Par le théorème de Tychonoff, $\mathbb{B}^{\mathcal{A}}$ est compact.

Par définition de la topologie produit, pour tout $A \in \mathcal{A}$, $\{I \mid I \models A\}$ est ouvert et fermé. Pour toute clause symbolique close $C = \pm A_1 \vee \ldots \vee \pm A_n$, $\{I \mid I \models C\} = \bigcap_I \{I \mid I \models \pm A_i\}$ est donc ouvert et fermé.

Par l'équation (3.7), pour toute clause symbolique C, $\{I \mid I \models C\} = \bigcap_{\sigma / C\sigma \text{ close}} \{I \mid I \models C\sigma\}$ est fermé.

Donc $\{I \mid I \models S\} = \bigcap_{C \in S, \sigma / C\sigma \text{ close}} \{I \mid I \models C\sigma\}$ est une intersection de fermés qui est vide (car S est insatisfiable par 1). $\mathbb{B}^{\mathcal{A}}$ étant compact, il existe une sous-intersection finie vide $\bigcap_{C \in S, \sigma / C\sigma \in S_0} \{I \mid I \models C\sigma\} = \{I \mid I \models S_0\}$, où S_0 est un ensemble fini d'instances closes de S. □

Nous aurons besoin, dans la démonstration du théorème de complétude, des définitions et des lemmes intermédiaires suivants :

Définition – 3.3.3

Soient $\sigma_1 : A_1 \to T_{B_1}$ et $\sigma_2 : A_2 \to T_{B_2}$ telles que $A_1 \cap A_2 = \emptyset$. Une union de σ_1 et σ_2 *est une application $\sigma : A \to T_B$ telle que*
- $A \supseteq A_1 \cup A_2$ *et* $B \supseteq B_1 \cup B_2$;
- $\forall t \in T_{A_1}.\ Subst_{A,B}(\sigma)(t) = Subst_{A_1,B}(\sigma_1)(t)$;
- $\forall t \in T_{A_2}.\ Subst_{A,B}(\sigma)(t) = Subst_{A_2,B}(\sigma_2)(t)$;

Lemme – 3.3.1

Soient $\sigma_1 : A_1 \to T_{B_1}$ et $\sigma_2 : A_2 \to T_{B_2}$ telles que $A_1 \cap A_2 = \emptyset$. Alors, la substitution $\sigma : A \to T_B$ définie par

$$Subst_{A,B}(\sigma_1 \cup \sigma_2)(x) = \begin{cases} Subst_{A_1,B}(\sigma_1)(x) & si\ x \in A_1 \\ Subst_{A_2,B}(\sigma_2)(x) & si\ x \in A_2 \end{cases}$$

où $A = A_1 \cup A_2$ et $B = B_1 \cup B_2$, est une union de σ_1 et σ_2.

Démonstration.

$$\begin{aligned} Subst_{A_1 \cup A_2, B_1 \cup B_2}(\sigma_1 \cup \sigma_2)(t) &= \left\{ \begin{array}{ll} Subst_{A_1,B_1}(\sigma_1)(t) & \text{si } t \in T_{A_1} \\ Subst_{A_2,B_2}(\sigma_2)(t) & \text{si } t \in T_{A_2} \end{array} \right. \\ & \qquad \text{par l'équation (3.1)} \\ &= \left\{ \begin{array}{ll} Subst_{A_1,B_1 \cup B_2}(\sigma_1)(t) & \text{si } t \in T_{A_1} \\ Subst_{A_2,B_1 \cup B_2}(\sigma_2)(t) & \text{si } t \in T_{A_2} \end{array} \right. \\ & \qquad \text{par l'équation (3.2)} \end{aligned}$$

□

Toute fonction $\varrho : A \to B$ définit une substitution canonique de A vers T_B puisque $B \subseteq T_B$.

Définition – 3.3.4

> *Un* renommage *est une application $\varrho : A \to B$, vue comme une substitution de A vers T_B, qui définit une bijection de A sur B. On notera encore ϱ cette substitution.*

Lemme – 3.3.2

> *Soit S un ensemble de clauses symboliques. Pour toute clause symbolique C dans S, pour tout $A \supseteq fv(C)$, pour tout renommage $\varrho : A \to B$, $B \subseteq \mathcal{V}$, alors $I \models C$ si et seulement si $I \models Subst_{A,B}(\varrho)C$.*

Démonstration. Si I est un modèle de C, on peut supposer que I est un modèle de Herbrand (proposition 3.2.1).

$I \models Subst_{A,B}(\varrho)C$ si et seulement si pour tout environnement ρ tel que $fv(Subst_{A,B}(\varrho)C) \subseteq \mathsf{dom}(\rho)$, il existe un littéral symbolique $+P(Subst_{A,B}(\varrho)(t_1), \ldots, Subst_{A,B}(\varrho)(t_n))$ dans $Subst_{A,B}(\varrho)C$ tel que pour tout i, $1 \leq i \leq n$, $[\![Subst_{A,B}(\varrho)(t_i)]\!]_B\rho$ est dans I_P, en particulier pour tout environnement de la forme $\rho = (\lambda x \in B.\rho'(\varrho^{-1}(y))$ où $fv(C) \subseteq \mathsf{dom}(\rho')$. Or, pour tout i, $1 \leq i \leq n$, on a

$$
\begin{aligned}
[\![Subst_{A,B}(\varrho)(t_i)]\!]_B\rho &= [\![t_i]\!]_A(\lambda x \in A.[\![\varrho(x)]\!]_B(\lambda x \in B.\rho'(\varrho^{-1}(y)))) \\
&\qquad\qquad \text{par l'équation (3.5)} \\
&= [\![t_i]\!]_A(\lambda x \in A.Subst_{B,\emptyset}(\lambda x \in B.\rho'(\varrho^{-1}(y)))(\varrho(x))) \\
&\qquad\qquad \text{par l'équation (3.7)} \\
&= [\![t_i]\!]_A(\lambda x \in A.\rho'(\varrho^{-1}(\varrho(x)))) \\
&= [\![t_i]\!]_A\rho' \\
&\qquad \text{car } \varrho \text{ est une bijection de } A \text{ vers } B
\end{aligned}
$$

Donc pour tout i, $1 \leq i \leq n$, $[\![t_i]\!]_A\rho'$ est dans I_P si et seulement si $[\![Subst_{A,B}(\varrho)(t_i)]\!]_B\rho$ est dans I_P.

Ainsi, $I \models Subst_{A,B}(\varrho)C$ si et seulement si $I \models C$. □

Ce lemme nous permet alors d'établir le corollaire suivant :

Corollaire – 3.3.1

> *Soient C et C' deux clauses symboliques quelconques. Alors on peut toujours supposer que $fv(C) \cap fv(C') = \emptyset$.*

Démonstration. On peut en effet faire cette supposition tout d'abord parce que l'ensemble $\mathcal{V}$ des variables est infini, ce qui nous permet de choisir A et B dans $\mathcal{V}$ tels que $A \cap B = \emptyset$, et ensuite, par le lemme 3.3.2, on peut toujours renommer une clause symbolique sans que son interprétation en soit affectée. Il suffit alors de renommer la clause C avec le renommage $\varrho : fv(C) \to B$ tel que $fv(C) \cap B = \emptyset$. □

Théorème – 3.3.2

> **(Complétude)** *La résolution abstraite ordonnée avec sélection est complète : pour tout ordre stable $\succ$, pour toute fonction de sélection, l'ensemble de clauses symboliques S est insatisfiable si et seulement si on peut dériver la clause vide □ à partir de S avec la seule règle de résolution abstraite ordonnée avec sélection.*

Démonstration. Nous allons démontrer la complétude dans le cas restreint où la règle de résolution abstraite ordonnée avec sélection ne contient que deux clauses en prémisse, c'est-à-dire dans le cas où $n = 1$. La preuve peut se généraliser au cas où $n \geq 1$, mais elle est alors très fastidieuse.

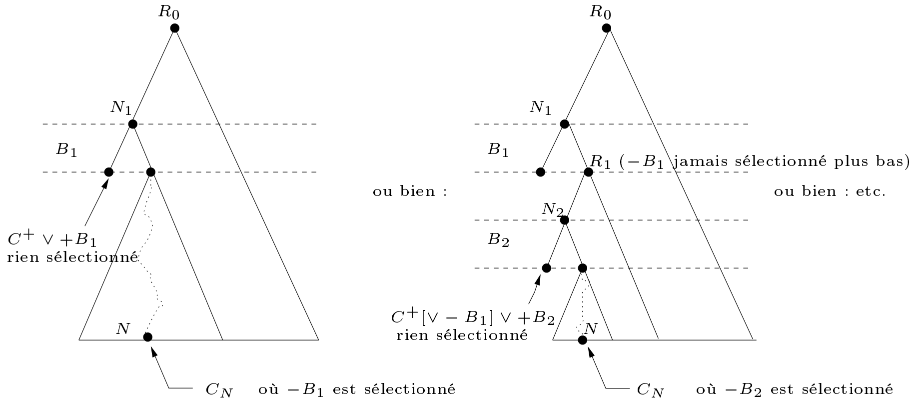


FIG. 3.2: La découverte d'un nœud d'échec N menant à la résolution ordonnée avec sélection

Réécrivons la règle de résolution ordonnée avec sélection dans le cas où $n = 1$:

$$\frac{C \vee +A_1 \vee \ldots \vee +A_n \quad C' \vee -A'}{(C \vee C')\sigma}$$

où :

(i) $n \geq 1$;

(ii) $\sigma \in \mathsf{csu}(A_1 \doteq A', \ldots, A_n \doteq A')$;

(iii) $\mathsf{sel}(C \vee +A_1 \vee \ldots \vee +A_n) = \emptyset$ et $A_1, \ldots, A_n$ sont maximaux dans $C \vee +A_1 \vee \ldots \vee +A_n$;

(iv) $-A' \in \mathsf{sel}(C' \vee -A')$, ou bien $\mathsf{sel}(C' \vee -A') = \emptyset$ et A' est maximal dans $C' \vee A'$.

Afin d'alléger les notations, nous écrirons $C\sigma$ pour $Subst_{fv(C),B}(\sigma)C$ et $B \subseteq \mathcal{V}$, en omettant ainsi l'indice B (on peut en effet voir une substitution comme la donnée de σ et de B, et par abus de notation ne garder que σ). On écrira de même $A\sigma$ pour un atome symbolique A.

La direction "si" est évidente. Réciproquement, fixons $A_1^0, \ldots, A_n^0$ l'ensemble fini des atomes symboliques clos figurant dans l'ensemble fini S_0 d'instances closes de S du théorème 3.3.1. On suppose les A_i^0 dans un ordre tel que si $A_i^0 \prec A_j^0$, alors $i < j$ (en d'autres termes, on étend $\succ$ à un ordre total sur les A_i^0). Appelons un arbre sémantique clos *adapté* si ses nœuds sont de la forme $\pm_1 A_1^0, \ldots, \pm_k A_k^0$ avec $k \leq n$. Par construction, il existe un arbre sémantique clos adapté, tel qu'à un nœud d'échec N est associé une clause C_N de S et une substitution θ_N telle que $C_N\theta_N$ est une clause symbolique qui est fausse en N.

Appelons *arbre décoré* pour un ensemble de clauses symboliques S' tout uplet $(T, C_\bullet, \theta_\bullet)$, où T est un arbre sémantique clos adapté, $C_\bullet$ associe à chaque nœud d'échec N (feuille de T) une clause symbolique C_N de S', et $\theta_\bullet$ associe à chaque nœud d'échec N une substitution θ_N telle que $C_N\theta_N$ est close et fausse en N. Il existe donc un arbre décoré pour S.

Étant donné un arbre décoré $(T, C_\bullet, \theta_\bullet)$ pour S', soit la racine ϵ est un nœud d'échec, et alors C_ϵ est nécessairement la clause vide, soit l'on trouve un nœud (qui n'est pas d'échec) dans T en calculant comme suit. Il sera utile de se guider sur la figure 3.2 pour comprendre la construction ; dans la figure, on n'a présenté que le cas de clauses symbolique closes, ce qui simplifie la construction.

Soit $R_0 = \epsilon$; pour tout $i \in \mathbb{N}$ pour lequel R_i est définie et n'est pas un nœud d'échec, soit T_i le sous-arbre de T de racine R_i, soit $N_i, -B_i$ le nœud d'échec de la branche gauche de T_i et $C_i \vee +B_i$ la clause symbolique $C_{N_i,-B_i}\theta_{N_i,-B_i}$, alors :

1. Si $N_i, +B_i$ est un nœud d'échec, stop.
2. Sinon :
   (a) S'il existe un nœud d'échec N dans le sous-arbre de T de racine $N_i, +B_i$ tel que $B_i = B\theta_N$ pour au moins un $-B \in \mathsf{sel}(C_N)$, alors stop.

(b) Sinon, poser $R_{i+1} = N_i, +B_i$, et itérer.

Cette définition s'arrête au bout d'un nombre fini d'itérations, car l'arbre décoré T est fini, et les R_i sont en ordre croissants (c'est-à-dire de plus en plus bas dans l'arbre). Soit k le plus grand entier pour lequel R_k, et donc N_k, est défini.

On note que $C_{N_k,-B_k}\theta_{N_k,-B_k}$ est une clause symbolique de la forme $C^+[\vee -B_1][\vee -B_2]\dots[\vee - B_{k-1}] \vee +B_k$, où C^+ est une clause symbolique positive, et la notation $[\vee - B_i]$ dénote la présence ou l'absence d'un littéral $-B_i$. Comme $N_k, -B_k$ est un nœud d'échec dans tous les arbres T_i, et que par construction pour toute clause symbolique $C_N\theta_N$ telle que N est un nœud d'échec dans T_i, B_i n'est pas de la forme $B\theta_N$ pour aucun $-B \in \mathsf{sel}(C_N)$, il s'ensuit qu'aucun littéral négatif ne peut être sélectionné dans $C_{N_k,-B_k}$: $\mathsf{sel}(C_{N_k,-B_k}) = \emptyset$. De plus, posons $C_{N_k,-B_k} = C \vee +A_1 \vee \dots \vee +A_n$, où les $+A_j$ sont les littéraux positifs de $C_{N_k,-B_k}$ tels que $A_j\theta_{N_k,-B_k} = B_k$. ($n \geq 1$, donc *(i)* est satisfaite.) Comme B_i est le littéral le plus grand de la clause symbolique $C_{N_k,-B_k}\theta_{N_k,-B_k}$, tous les $+A_j$, $1 \leq j \leq n$, sont maximaux dans $C_{N_k,-B_k}$ (s'il existait un littéral $\pm A$ plus grand, par stabilité on aurait $A\theta_{N_k,-B_k} \succ B_k$). Donc la condition *(iii)* est satisfaite.

Si l'itération ci-dessus s'arrête dans le cas 1, alors $N_k, +B_k$ est un nœud d'échec. Posons $N = N_k, +B_k$. Alors C_N est de la forme $C' \vee -A'$, où $-A'$ est l'un des littéraux tels que $A'\theta_N = B_k$. Par un argument similaire à celui pour $C_{N_k,-B_k}$ ci-dessus, on a $\mathsf{sel}(C' \vee -A') = \{-A'\}$ ou $\mathsf{sel}(C' \vee -A') = \emptyset$, et A' est maximal dans $C' \vee -A'$, c'est-à-dire que les deux possibilités de l'alternative *(iv)* sont vérifiées.

Si l'itération s'arrête dans le cas 2.(a), alors il existe un nœud d'échec N dans un sous-arbre de T de racine $N_k, +B_k$ tel que $B_k = B\theta_N$ pour au moins un $-B \in \mathsf{sel}(C_N)$. Renommons B en A'. La clause C_N est donc de la forme $C' \vee -A'$, avec $-A' \in \mathsf{sel}(C' \vee -A')$, et $B_k = A'\theta_N$. La première possibilité de l'alternative *(iv)* est donc vérifiée.

On suppose, grâce au corollaire 3.3.1, que $C \vee +A_1 \vee \dots \vee +A_n$ et $C' \vee -A'$ ont des ensembles de variables libres disjoints B et B'. L'union des substitutions $\sigma_0 : B \cup B' \to T_C$, pour un certain $C \subseteq \mathcal{V}$, de $\theta_{N_k,-B_k}$ et θ_N (que l'itération s'arrête en 1 ou en 2.(a)) est donc bien définie (par le lemme 3.3.1) et est un unificateur de $A_1, \dots, A_n, A'$. Donc il existe un unificateur σ : $B \cup B' \to T_C$ dans $\mathsf{csu}(A_1 \doteq A', \dots, A_n \doteq A')$ tel que pour tout $t \in B \cup B'$ on a $Subst_{B\cup B',C}(\sigma_0)(t) = Subst_{C,C}(\theta)(Subst_{B\cup B',C}(\theta)(t))$ pour une certaine substitution $\theta : C_1 \to T_C$. En particulier, la condition *(ii)* est satisfaite.

Les conditions *(i)*-*(iv)* étant satisfaites, la clause symbolique $C\sigma \vee C'\sigma$ est déductible par résolution abstraite ordonnée avec sélection.

Montrons que ce processus termine. Définissons le nouvel arbre décoré $(T', C'_\bullet, \theta'_\bullet)$ à partir de $(T, C_\bullet, \theta_\bullet)$ comme suit. Soit N' le nœud le plus haut de T où $(C\sigma \vee C'\sigma)\theta$ est faux. N' existe, car par construction N rend faux $(C\sigma \vee C'\sigma)\theta = C\sigma_0 \vee C'\sigma_0 = C\theta_{N_k,-B_k} \vee C'\theta_{N_k,+B_k}$. Alors :

(a) Si N' est strictement plus haut que N dans T, alors soit T' l'arbre sémantique clos dont les nœuds d'échec sont N', plus tous les nœuds d'échec de T qui ne sont pas au-dessous de N'. On pose $C'_{N'} = C\sigma \vee C'\sigma$, $\theta'_{N'} = \theta$, alors que $C'_{N''} = C_{N''}$ et $\theta'_{N''} = \theta_{N''}$ pour $N'' \neq N'$;

(b) Si $N = N'$, on pose $T' = T$, $C'_{N'} = C\sigma \vee C'\sigma$, $\theta'_N = \theta$ et $C'_{N''} = C_{N''}$ et $\theta'_{N''} = \theta_{N''}$ pour tout $N'' \neq N$.

Ce dernier cas ne peut se produire que si $-B_k$ est un littéral de $C'_N\theta'_N = C\theta_{N_k,-B_k} \vee C'\theta_{N_k,+B_k}$; ce littéral ne peut provenir que de $C'\theta_{N_k,+B_k}$, et l'on observe que le nombre m' de littéraux $-A'$ de C'_N tels que $A'\theta'_N = B_k$ est au plus un de moins que le nombre m de littéraux $-A'$ de C_N tels que $A'\theta_N = B_k$.

On définit donc la mesure $\mu_1(C_N, \theta_N)$ du nœud d'échec N d'un arbre décoré $(T, C_\bullet, \theta_\bullet)$ comme étant le multi-ensemble contenant autant de fois l'entier i qu'il y a de littéraux $\pm A'$ de C_N tels que $A'\theta_N = A_i^0$. Dans le cas *(b)*, où $N' = N$, on notera que ce multi-ensemble décroît lorsque l'on remplace $C_{N_k,+B_k}$ (avec $B_k = A_p^0$), qui est de mesure

$$\{ \underbrace{\dots}_{\text{tous } <p}, \underbrace{p, \dots, p}_{\text{en nombre } m} \}$$

par $C'_N = C\sigma \vee C'\sigma$, qui est de mesure

$$\{ \underbrace{\ldots}_{\text{tous } <p}, \underbrace{p, \ldots, p}_{\text{en nombre } m'<m} \}.$$

Notons que ce dernier multi-ensemble est plus petit que le précédent pour l'ordre multi-ensemble, et que cet ordre est bien fondé (voir annexe A).

On définit ensuite la mesure $\mu^-(T, C_\bullet, \theta_\bullet)$ comme étant le multi-ensemble des $\mu_1(C_N, \theta_N)$, N parcourant les nœuds d'échec de T. Dans le cas *(b)*, $\mu_1(C_N, \theta_N)$ décroît strictement, les autres $\mu_1(C_{N''}, \theta_{N''})$ restant inchangés. Donc la mesure μ^- décroît pour l'ordre multi-ensemble dans le cas *(b)*.

D'un autre côté, la taille $|T|$ décroît strictement dans le cas *(a)*, et est inchangée dans le cas *(b)*. Donc la mesure $\mu(T, C_\bullet, \theta_\bullet)$ définie comme le couple $(|T|, \mu^-(T, C_\bullet, \theta_\bullet))$ ordonné lexicographiquement sur les couples formés d'un entier et d'un multi-ensemble de multi-ensembles, décroît lors du passage de $(T, C_\bullet, \theta_\bullet)$ à $(T', C'_\bullet, \theta'_\bullet)$. L'ordre lexicographique sur les couples formés d'un entier et d'un multi-ensemble de multi-ensembles étant bien formé (voir annexe A), le processus termine, et l'on peut donc dériver la clause vide en un nombre fini d'étapes de résolution abstraite ordonnée avec sélection. □

3.4 Stratégies d'élimination de clauses symboliques

Comme dans le cas du premier ordre (section 1.4), on va combiner la règle de résolution abstraite avec des règles de simplification de l'ensemble de clauses courant.

Nous allons regarder quelles clauses symboliques peuvent être éliminées de façon sûre en regardant finement la preuve du théorème 1.3.1. L'idée sera de garantir que la mesure $\mu(T, C_\bullet, \theta_\bullet)$ (ou un raffinement) décroisse.

3.4.1 Élimination de tautologies

On rappelle qu'une clause est *tautologique* si et seulement si elle est de la forme $C \vee +A \vee -A$ pour un certain atome A.

Lemme – 3.4.1

> *La résolution abstraite ordonnée avec sélection et élimination des clauses tautologiques est complète.*

Démonstration. Une tautologie $C_1 \vee +A \vee -A$ n'a que des instances closes vraies. Donc aucune clause $C_N\theta_N$ n'est instance d'une telle clause.

Raffinons la mesure des arbres décorés en posant $\mu'(S, T, C_\bullet, \theta_\bullet) = (\mu(T, C_\bullet, \theta_\bullet), |S|)$ ordonnée lexicographiquement (on peut consulter l'annexe A pour des détails sur les ordres), où $|S|$ est la taille de l'ensemble de clauses S, et où l'on contraint $(T, C_\bullet, \theta_\bullet)$ à être un arbre décoré pour S. Comme aucune tautologie de S n'est une clause C_N, l'élimination de tautologies laisse $\mu(T, C_\bullet, \theta_\bullet)$ inchangée, mais fait décroître $|S|$. □

On en conclut que l'on peut intercaler entre toutes étapes de résolution abstraite ordonnée avec sélection n'importe quel nombre d'effacement de tautologies. Notamment, on peut décider d'éliminer systématiquement toutes les tautologies.

3.4.2 Élimination de clauses symboliques subsumées

Rappelons qu'une clause C *subsume* une clause C' si et seulement s'il existe σ telle que C' est de la forme $C\sigma \vee C_1$. Nous dirons qu'elle la subsume *linéairement* si σ n'unifie aucune paire de littéraux de C.

Lemme – 3.4.2

> *La résolution abstraite ordonnée avec sélection et élimination des clauses subsumées linéairement est complète.*

Démonstration. Supposons que S contient C et une clause subsumée $C\sigma \vee C_1$ (dans $C\sigma \vee C_1$, on suppose implicitement qu'il n'y a aucun littéral commun à $C\sigma$ et C_1). Supposons que N est un nœud d'échec rendant faux une instance close $(C\sigma \vee C_1)\theta$ de $C\sigma \vee C_1$. Comme N rend fausse la clause $C\sigma\theta \vee C_1\theta$, en particulier N rend fausse la clause $C\sigma\theta$, qui est une instance close de C. Donc il existe un nœud d'échec N' au-dessus de N.

Si N' est strictement au-dessus de N, comme en section 3.4.1, éliminer $C\sigma \vee C_1$ laisse $\mu(T, C_\bullet, \theta_\bullet)$ inchangé, mais fait décroître $|S|$.

Si $N = N'$ par contre, il faut être plus prudent. Soit C_N n'est pas la clause $C\sigma \vee C_1$, et le même argument s'applique, soit C_N est la clause $C\sigma \vee C_1$, et l'on peut remplacer la clause C_N au nœud d'échec N par C, à condition que $\mu_1(C, \sigma\theta)$ soit plus petit ou égal dans l'ordre multi-ensemble que $\mu_1(C\sigma \vee C_1, \theta)$ (le cas plus petit est clair, et si on a l'égalité $|S|$ décroît).

C'est notamment le cas si σ n'unifie aucun littéral dans C (autrement dit, ne calcule pas un facteur de C). En effet, dans ce cas $\mu_1(C\sigma \vee C_1, \theta) = \mu_1(C\sigma, \theta) \cup \mu_1(C_1, \theta) \geq \mu_1(C\sigma, \theta) = \mu_1(C, \sigma\theta)$, la dernière égalité étant due au fait que σ n'unifie aucun littéral dans C (en général $\mu_1(C\sigma, \theta) \leq \mu_1(C, \sigma\theta)$). □

La discussion ci-dessus montre que l'on peut éliminer toute clause subsumée linéairement, à tout moment de la recherche de preuve. En particulier, on peut supprimer les clauses subsumées linéairement le plus tôt possible.

Par exemple, la clause $+P(x) \vee +P(y)$ sur les termes du premier ordre subsume linéairement les clauses :

$+P(a) \vee +P(y)$ (σ instancie x par a)
$+P(a) \vee +P(y) \vee +Q(z)$
$+P(x) \vee +P(y) \vee +Q(z)$

et elle subsume, mais pas linéairement :

$+P(z)$ (σ instancie x et y par z)
$+P(a)$ (σ instancie x et y par a)

On remarque que, réciproquement, $+P(z)$ subsume linéairement $+P(x) \vee +P(y)$.

Remarque 4 *On peut en fait fournir des critères d'élimination de clauses subsumées plus libéraux, à condition de changer de mesure $\mu(T, C_\bullet, \theta_\bullet)$. Un cas simple est celui de la résolution ordonnée, c'est-à-dire où $\mathsf{sel}(C) = \emptyset$ pour toute clause C. La complétude de ce cas peut se montrer en prenant $\mu_1(C, \theta) = |T|$; on peut alors éliminer toute clause subsumée, même si elle ne l'est pas linéairement, à tout moment.*

Dans le cas de la résolution ordonnée avec sélection, on peut raffiner $\mu_1(C, \theta)$. Étendons l'ordre $\succ$ sur les atomes à un ordre sur les littéraux par le produit lexicographique de l'ordre $\succ$ sur l'atome sous-jacent avec l'ordre $- > +$ sur le signe. Ordonnons les clauses C par l'ordre multi-ensemble $\gg$ sur les littéraux, ordonnés par l'ordre précédent. La preuve du théorème 3.3.2 tient toujours en choisissant $\mu_1(C, \theta)$ égale à C, ordonnée par $\gg$. D'autre part, on a toujours $C\sigma \vee C_1 \gg C$, sauf si $C\sigma \vee C_1$ subsume elle aussi C. On en conclut que l'on peut éliminer toute clause C strictement subsumée par une autre clause C', c'est-à-dire toute clause C subsumée par C' telle que C' ne subsume pas C. D'autre part, on peut aussi éliminer tout résolvant abstrait (ordonné, avec sélection) qui est subsumé (pas nécessairement strictement) par une autre clause symbolique de S. C'est notamment de cette manière que l'élimination des clauses subsumées est traitée dans [BG01a].

3.4.3 Élimination des clauses symboliques pures

Rappelons qu'une clause symbolique est *pure* dans un ensemble S de clauses symboliques si et seulement si C est de la forme $C_1 \vee \pm A$, et il n'existe aucune clause symbolique de la forme $C_1' \vee \mp A'$ dans S telle que A et A' sont unifiables (on dit que $\pm A$ est *pur*).

Lemme – 3.4.3

La résolution abstraite ordonnée avec sélection et élimination des clauses symboliques pures est complète.

Démonstration. Un argument facile montrant qu'on peut éliminer les clauses symboliques pures est le suivant : tout résolvant d'une clause symbolique pure $C \vee \pm A$ (où A est pur, c'est-à-dire ne s'unifie avec aucun $\mp A'$ dans aucune clause) avec une clause symbolique C' est de la forme $C'' \vee \pm Subst_{fv(A),B}(\sigma)A$, où $\sigma : fv(A) \to T_B$, avec $B \subseteq \mathcal{V}$. Or $\pm Subst_{fv(A),B}(\sigma)A$ ne s'unifie toujours avec aucun $\mp A'$ dans aucune clause symbolique.

En effet, s'il existait un tel littéral $\mp A'$, il existerait $\sigma' : B \cup fv(A') \to T_{B'}$, $B \subseteq \mathcal{V}$, tel que $Subst_{B\cup fv(A'),B'}(\sigma')(A') = Subst_{B\cup fv(A'),B'}(\sigma')(Subst_{fv(A),B}(\sigma)(A))$.

Or, $Subst_{B\cup fv(A'),B'}(\sigma')(Subst_{fv(A),B}(\sigma)(A))$

$$\begin{aligned} &= Subst_{B\cup fv(A'),B'}(\sigma')(Subst_{fv(A),B\cup fv(A')}(\sigma)(A)) && \text{par l'équation (3.2)}\\ &= Subst_{fv(A),B'}(\lambda x \in fv(A).Subst_{B\cup fv(A'),B'}(\sigma')(\sigma(x)))(A) && \text{par l'équation (3.4)}\\ &= Subst_{fv(A)\cup fv(A'),B'}(\lambda x \in fv(A).Subst_{B\cup fv(A'),B'}(\sigma')(\sigma(x)))(A) && \text{par l'équation (3.1)}\end{aligned}$$

et $Subst_{B\cup fv(A'),B'}(\sigma')(A')$

$$\begin{aligned} &= Subst_{fv(A'),B'}(\sigma')(A') && \text{par l'équation (3.1)}\\ &= Subst_{fv(A)\cup fv(A'),B'}(\sigma')(A') && \text{par l'équation (3.1).}\end{aligned}$$

Grâce au lemme 3.3.2 et à son corollaire 3.3.1, nous pouvons supposer que $fv(A)$, $fv(A')$ et B ont une intersection vide, autrement dit que A, A' et $Subst_{fv(A),B}(\sigma)A$ n'ont aucune variable en commun.

Soit alors $\theta : fv(A) \cup fv(A') \to T_{B'}$ définie par :

$$\theta(x) = \begin{cases} \sigma'(x) & \text{si } x \in fv(A') \\ Subst_{fv(A)\cup fv(A'),B'}(\sigma')(\sigma(x)) & \text{si } x \in fv(A). \end{cases}$$

Ainsi, $Subst_{fv(A)\cup fv(A'),B'}(\lambda x \in fv(A).Subst_{B\cup fv(A'),B'}(\sigma')(\sigma(x)))(A)$
$= Subst_{fv(A)\cup fv(A'),B'}(\theta)(A)$,

donc $Subst_{fv(A)\cup fv(A'),B'}(\sigma')(A') = Subst_{fv(A)\cup fv(A'),B'}(\theta)(A')$.

Donc θ est un unificateur de A et A', ce qui contredit la pureté de A dans S.

Donc tout résolvant d'une clause symbolique pure est pur. Comme la clause vide n'est pas pure, aucune des étapes de résolution construites dans la preuve du théorème 3.3.2 ne peut utiliser de clause pure. On peut donc toutes les éliminer. □

On peut aussi choisir d'éliminer les clauses symboliques *fortement pures*, c'est-à-dire des clauses symboliques dans lesquelles le littéral pur est sélectionné par la fonction de sélection, ou à défaut est maximal. On peut alors donner un argument direct de complétude d'élimination de ces clauses pures particulières par la technique de preuve du théorème 3.3.2, en remarquant qu'aucune clause symbolique C_N de l'arbre sémantique ne peut être fortement pure.

3.5 Splitting

3.5.1 Splitting et tableaux

Nous avons défini en section 1.5.1 le splitting classique utilisant des notions de tableaux pour la logique du premier ordre. Nous allons ici présenter une technique similaire et montrer un résultat de complétude pour des ensembles de clauses symboliques.

Rappelons tout d'abord le principe du splitting, ainsi que quelques définitions similaires au cas de la logique du premier ordre.

Un *tableau* $\mathcal{T}$ est un multi-ensemble fini non vide d'ensembles de clauses symboliques. Le tableau formé des ensembles $S_1, \ldots, S_n$ de clauses symboliques sera noté $S_1|\ldots|S_n$, et l'union des multi-ensembles des tableaux $\mathcal{T}$ et $\mathcal{T}'$ sera noté $\mathcal{T}|\mathcal{T}'$. Chaque ensemble de clauses symboliques S_i est appelé une *branche* du tableau $\mathcal{T}$. On dira qu'une branche est *close* si et seulement si elle contient la clause vide $\Box$, et qu'un tableau $\mathcal{T}$ est *clos* si et seulement si toutes ses branches sont closes. Notons $\rhd$ la relation binaire de déduction définie par $S \rhd S'$ si et seulement si S' est obtenu par l'ajout à S d'un résolvant ordonné avec sélection ou par l'élimination d'une clause tautologique ou d'une clause subsumée linéairement par une autre clause de S. On peut alors formaliser le splitting à l'aide de la définition de la relation binaire $\rightsquigarrow$ entre tableaux par les règles suivantes : On dit qu'une interprétation I *satisfait* un tableau $\mathcal{T}$ si et seulement si I satisfait au

$$\frac{S \rhd S'}{S|\mathcal{T} \rightsquigarrow S'|\mathcal{T}}\,(\rhd) \qquad \frac{fv(C) \cap fv(C') = \emptyset}{S \cup \{C \vee C'\}|\mathcal{T} \rightsquigarrow S \cup \{C\}|S \cup \{C'\}|\mathcal{T}}\,(Split)$$

Fig. 3.3: Splitting et tableaux

moins une branche du tableau $\mathcal{T}$. En particulier, $\mathcal{T}$ est *insatisfiable* si et seulement si toutes ses branches sont insatisfiables.

Notons que $\rhd$ est correcte au sens où si $S \rhd S'$ et S est satisfiable, alors S' est satisfiable, et qu'elle est complète dans un sens analogue.

Lemme – 3.5.1

La relation $\rightsquigarrow$ est correcte, au sens où si $\mathcal{T} \rightsquigarrow^* \mathcal{T}'$ et $\mathcal{T}$ est satisfiable, alors $\mathcal{T}'$ est satisfiable.

En particulier, si $\mathcal{T} \rightsquigarrow^* \mathcal{T}'$ et $\mathcal{T}'$ est un tableau clos, alors $\mathcal{T}$ est insatisfiable.

Démonstration.

- Règle $(\rhd)$ Si $S \rhd S'$ et $S|\mathcal{T}$ est satisfiable, soit I une interprétation qui satisfaisant $S|\mathcal{T}$. Si I satisfait S, alors par hypothèse, il existe une interprétation I' qui satisfait S', donc I' satisfait $S'|\mathcal{T}$.
- Règle (*Split*) Si C et C' sont deux clauses symboliques n'ayant aucune variable en commun, soit I une interprétation satisfaisant $S \cup \{C \vee C'\}|\mathcal{T}$. Soit I satisfait $S \cup \{C \vee C'\}$, soit I satisfait $\mathcal{T}$. Si I satisfait $S \cup \{C \vee C'\}$, alors I satisfait $S \cup \{C\}$ ou $S \cup \{C'\}$, donc I satisfait $S \cup \{C\}|S \cup \{C'\}|\mathcal{T}$. Et trivialement, si I satisfait $\mathcal{T}$, alors I satisfait $S \cup \{C\}|S \cup \{C'\}|\mathcal{T}$.

$\Box$

Pour la complétude, on va supposer que $\rhd$ est l'union des relations $\rhd_r$ et $\rhd_{eff}$, où $S \rhd_r S'$ si et seulement si S' est obtenu par l'ajout à S d'un résolvant ordonné avec sélection, et $S \rhd_{eff} S'$ si et seulement si S' est obtenu par l'élimination d'une tautologie ou d'une clause symbolique subsumée par une clause symbolique de S.

On définit une *stratégie d'effacement et de splitting* comme une relation f qui à chaque tableau $\mathcal{T}$ associe soit le symbole spécial $*$, soit un tableau $\mathcal{T}'$ tel que soit $\mathcal{T} \rightsquigarrow \mathcal{T}'$ est déductible par la règle (*Split*), soit $\mathcal{T} \rhd_{eff} \mathcal{T}'$. Une f-dérivation est une dérivation

$$\mathcal{T}_0 \rightsquigarrow \mathcal{T}_1 \rightsquigarrow \ldots \rightsquigarrow \mathcal{T}_k$$

où pour tout i, $1 \leq i \leq k$, soit $f(\mathcal{T}_{i-1}) = \mathcal{T}_i$, soit $f(\mathcal{T}_{i-1}) = *$. On dit que $\rhd_r$ est f-*complète* si et seulement si, pour tout tableau $\mathcal{T}$ insatisfiable, il existe une dérivation finie comme ci-dessus telle que $\mathcal{T} = \mathcal{T}_0$ et $\mathcal{T}_k$ est clos. On a alors :

Théorème – 3.5.1

> *La résolution abstraite ordonnée avec sélection, élimination de tautologies et de clauses symboliques subsumées linéairement et utilisation de la règle de splitting (Split), est complète.*

Démonstration. On mesure chaque branche S d'un tableau par le couple $(\mu(T, C_\bullet, \theta_\bullet), |S|)$ comme en section 3.4, et un tableau $\mathcal{T}$ par le multi-ensemble de ces quantités lorsque S parcourt les branches de $\mathcal{T}$. Il est clair que toute application de (*Split*) ou de ($\rhd^{eff}$) va faire décroître cette mesure, et la construction de la preuve du théorème 3.3.2 fournit une étape de résolution abstraite ordonnée avec sélection $\rhd^r$ qui fait décroître cette mesure à partir de toute branche non close de tout tableau $\mathcal{T}$ si $f(\mathcal{T}) = *$. □

3.5.2 Splitting sans splitting

Nous avons vu en section 1.5.2 le splitting sans splitting pour le premier ordre. Le principe est ici similaire.

On définit de la même manière un ensemble de symboles de prédicats $\mathcal{Q}$ disjoint de $\mathcal{P}$ en bijection avec l'ensemble des clauses symboliques formées à l'aide de prédicats dans $\mathcal{P}$ uniquement. Il existe donc une application qui, à chaque clause symbolique formée à l'aide de prédicats dans $\mathcal{P}$ uniquement, associe un symbole $\ulcorner C \urcorner$ de $\mathcal{Q}$, telle que $\ulcorner C \urcorner = \ulcorner C' \urcorner$ si et seulement si il existe un renommage ρ tel que $C = C'\rho$. Pour éviter de parler de symboles de prédicats d'arité nulle, on note $q(*)$ l'atome formé avec $q \in \mathcal{Q}$, où $*$ est un élément quelconque de $T_\emptyset$ non utilisé par ailleurs. L'ensemble $T_\emptyset$ doit alors être non vide, ce que nous avons supposé en début de section 3.3 (p. 59).

Rappelons la règle de *Splitting sans splitting* décrite en figure 3.4.

$$\frac{C \vee C'}{C \vee -\ulcorner C' \urcorner(*) \qquad + \ulcorner C' \urcorner(*) \vee C'}$$

où :
- $fv(C) \cap fv(C') = \emptyset$;
- C' est formé à l'aide de symboles de prédicats de $\mathcal{P}$ uniquement ;
- C contient au moins un atome symbolique $P(t)$ avec $P \in \mathcal{P}$.

Fig. 3.4: Règle de splitting sans splitting

L'effet est de remplacer la clause symbolique en prémisse par les deux clauses symboliques en conclusion de la règle.

Définition – 3.5.1

> *On dira que C* est splittable en C_1 et C_2 *si C_1 et C_2 sont obtenus à partir de C par la règle de splitting sans splitting.*

Soit I une interprétation de Tarski, de domaine D, des symboles de prédicats dans $\mathcal{P}$. Son *extension standard* $I^{\mathcal{Q}}$ est telle que $I^{\mathcal{Q}}_P = I_P$ pour tout $P \in \mathcal{P}$, et $I^{\mathcal{Q}}{}_{\ulcorner C \urcorner}$ vaut D si $I \not\models C$, et $\emptyset$ si $I \models C$. On dira qu'une interprétation I est *standard* si et seulement si, pour toute clause symbolique C formée à l'aide de prédicats dans $\mathcal{P}$ uniquement, $I_{\ulcorner C \urcorner}$ est vide si et seulement si $I \models C$. Toute extension standard est clairement standard.

Lemme – 3.5.2

> *La résolution abstraite ordonnée avec sélection, l'élimination de tautologies et de clauses symboliques subsumées linéairement, et la règle de splitting sans splitting sont correctes au sens où, s'il existe une interprétation standard I telle que $I \models S$ et si S' est déduite de S par l'une de ces règles, alors $I \models S'$.*
>
> *En particulier, si on peut déduire la clause vide de S par ces règles, et si S ne contient que des clauses symboliques formées à l'aide de symboles de prédicats dans $\mathcal{P}$ uniquement, alors S est insatisfiable.*

Démonstration. Immédiat pour toutes les règles, sauf éventuellement celle du splitting sans splitting. Si $I \models C \vee C'$, alors on a deux cas :

- Si $I \models C'$, comme I est standard, $I \not\models \ulcorner C' \urcorner(*)$, donc $I \models C \vee -\ulcorner C' \urcorner(*)$; d'autre part, $I \models +\ulcorner C' \urcorner(*) \vee C'$, donc I est un modèle de toutes les clauses symboliques en conclusion de la règle.
- Si $I \not\models C'$, comme I est standard, $I \models \ulcorner C' \urcorner(*)$, donc $I \models +\ulcorner C' \urcorner(*) \vee C'$; d'autre part, comme $I \not\models C'$ et que $I \models C \vee C'$, C et C' ne partageant aucune variable, alors nécessairement $I \models C$, donc $I \models C \vee -\ulcorner C' \urcorner(*)$. Donc, encore une fois, I est un modèle de toutes les clauses symboliques en conclusion de la règle.

Finalement, si on peut déduire la clause vide à partir de S, S ne peut donc avoir aucun modèle standard. Mais toute extension standard d'un modèle de S est un modèle standard de S. Donc S n'a aucun modèle : S est insatisfiable. □

L'ajout de $\mathcal{Q}$ modifie la base de Herbrand, c'est-à-dire l'ensemble des atomes symboliques clos à considérer. On doit maintenant considérer tous les atomes symboliques clos de la forme $P(t_1, \ldots, t_n), P \in \mathcal{P}$, mais aussi tous les atomes symboliques clos de la forme $Q()$, où $Q \in \mathcal{Q}$. Pour tout ordre stable $\succ$, on étend $\succ$ de sorte que $P(t_1, \ldots, t_n) \succ Q$ pour tout terme symbolique clos $t_i \in T_\emptyset$, $1 \leq i \leq n$, pour tout $P \in \mathcal{P}$, pour tout $Q \in \mathcal{Q}$. Un tel ordre sera appelé *admissible*.

Théorème – 3.5.2

> *Soit $\succ$ un ordre admissible. La résolution abstraite ordonnée pour $\succ$, avec sélection, élimination de tautologies, de clauses symboliques subsumées linéairement et splitting sans splitting est complète.*

Démonstration. Il suffit de remarquer que si N est un nœud d'échec, et si C_N est de la forme $C \vee C'$, où ni C ni C' n'est vide, alors soit $\ulcorner C' \urcorner(*)$ est fausse en N et alors $+\ulcorner C' \urcorner(*) \vee C'\theta_N$ est fausse en N, soit $\ulcorner C' \urcorner(*)$ est vraie en N, et alors $C\theta_N \vee -\ulcorner C' \urcorner(*)$ est fausse en N.

Dans le deuxième cas, $\mu_1(C \vee -\ulcorner C' \urcorner(*), \theta_N)$ est plus petite que $\mu_1(C \vee C', \theta_N)$, car C' est non vide, construite uniquement à partir de symboles de $\mathcal{P}$, donc pour tout atome A de C', $A\theta_N \succ \ulcorner C' \urcorner(*)$, puisque $\succ$ est un ordre admissible.

Dans le premier cas, $\mu_1(+\ulcorner C' \urcorner(*) \vee C', \theta_N)$ est plus petite que $\mu_1(C \vee C', \theta_N)$, car C contient au moins un atome symbolique de la forme $P(t_1, \ldots, t_n)$ avec $P \in \mathcal{P}$, donc tel que $P(t_1, \ldots, t_n)\theta_N \succ \ulcorner C' \urcorner(*)$. □

Proposition – 3.5.1

> *Les règles d'élimination de tautologies, de clauses symboliques subsumées linéairement et de splitting sans splitting peuvent être appliquées à tout moment du processus de résolution abstraite ordonnée avec sélection.*

Démonstration. Nous allons procéder dans le même état d'esprit que pour la preuve de complétude du splitting classique (section 1.5.1).

Soit $\rhd$ l'union des relations $\rhd_r$, $\rhd_{eff}$ et $\rhd_{spl}$, où $S \rhd_r S'$ si et seulement si S' est obtenu par l'ajout d'un résolvant ordonné avec sélection, $S \rhd_{eff} S'$ si et seulement si S' est obtenu par l'élimination d'une clause subsumée ou d'une tautologie, et $S \rhd_{spl}$ si et seulement si S' est obtenu en remplaçant une clause C de S par les clauses résultantes du splitting sans splitting.

On définit une *stratégie d'effacement et de splitting sans splitting* comme étant une relation qui à chaque S associe soit S' tel que $S \rhd_{eff} S'$ ou $S \rhd_{spl} S'$, soit le symbole spécial $\star$.

Une *R-dérivation* est une dérivation :

$$S_0 \rightsquigarrow S_1 \rightsquigarrow \ldots \rightsquigarrow S_k$$

où pour tout j, $1 \leq j \leq k$, $R(S_{i-1}) = S_i$ ou $R(S_{i-1}) = \star$.

On dit que $\rhd_r$ est *R-complète* si et seulement si pour tout S insatisfiable, il existe une dérivation finie telle que $S = S_0$ et S_k insatisfiable.

Mesurons S par le couple $(\mu(T, C_\bullet, \theta_\bullet), |S|)$. Toute application de $\rhd_{eff}$ va clairement faire décroître la mesure. De plus, toute application de $\rhd_{spl}$ fait décroître la mesure $\mu_1(C_N, \theta_N)$ comme nous venons de le voir dans la démonstration précédente. Donc, par définition $\mu(T, C_\bullet, \theta_\bullet)$ décroît elle aussi, ainsi que $(\mu(T, C_\bullet, \theta_\bullet), |S|)$. Par ailleurs, la construction de la preuve du théorème 3.3.2 fournit une étape de résolution abstraite ordonnée avec sélection $\rhd^r$ qui fait décroître cette mesure à partir de tout ensemble S tel que $R(S) = \star$. $\square$

Nous avons ainsi défini la notion de résolution abstraite ordonnée avec sélection sur des clauses symboliques. Nous avons montré la correction et la complétude de ce raffinement de la résolution avec des stratégies de splitting et d'élimination de clauses.

Ceci va nous permettre de définir dans le chapitre suivant un prouveur modulaire dont le moteur de résolution sera basé sur la résolution abstraite ordonnée avec sélection.

Chapitre 4

MoP : un prouveur modulaire

Il existe de nombreuses implémentations de démonstrateurs automatiques. A l'instar de OTTER [Kal01], Vampire [RV01], Setheo [Ibe97], SPASS [Wei99a], Waldmeister [Hil03], ils sont tous spécialisés à la logique du premier ordre.

Il n'existe cependant pas à notre connaissance de démonstrateur de théorèmes pour des logiques clausales différentes. Or, en vérification de protocoles cryptographiques, l'approche qui consiste à modéliser les protocoles par des ensembles de clauses a l'avantage qu'on dispose ainsi de résultats sur la résolution qui nous offrent des techniques de preuves. Cependant tous ne sont pas décrits par les mêmes classes de clauses, et en particulier, comme dans le cas du protocole IKA.1, certains utilise des opérateurs définis modulo une théorie équationnelle. La résolution abstraite ordonnée avec sélection pour laquelle nous avons montré des résultats de correction et de complétude a l'avantage de pouvoir s'appliquer à toute famille normale de termes symboliques. Ainsi, il suffit, pour chaque famille de termes symboliques, de définir les notions qui leur sont propres (l'unification par exemple) pour disposer d'un prouveur sur ces termes.

Notre principal apport est donc d'avoir développé un démonstrateur automatique *modulaire* pour une logique clausale abstraite (chapitre 3), dont le noyau est basé sur une stratégie de résolution ordonnée avec sélection, où l'ordre sur les termes et littéraux, ainsi que la fonction de sélection sont des paramètres du système, donc définissables par l'utilisateur. Ce noyau est accompagné de modules de termes *concrets* par opposition aux termes *abstraits* correspondants aux termes symboliques définis en section 3.1. Nous verrons alors, dans le chapitre suivant, qu'il suffira de définir une substitution sur les termes concrets, un algorithme d'unification de deux termes et un ordre stable, ces notions devant être adéquates à celles du chapitre 3.

Alors que l'accent dans les projets cités plus haut porte sur l'efficacité pour les termes du premier ordre, celui de notre prouveur est mis sur la modularité et l'ouverture, l'efficacité passant au second plan.

Dans ce chapitre, nous exposerons l'architecture du prouveur modulaire MoP en section 4.1, en définissant les termes abstraits (section 4.1.1) puis le module de résolution (section 4.1.2), et enfin nous décrirons l'algorithme de résolution abstraite (section 4.2).

4.1 Architecture générale de MoP

Nous avons choisi le langage de programmation Caml pour réaliser le démonstrateur automatique, tout d'abord parce que c'est un langage élégant et fortement typé, ensuite parce qu'il offre une structure de modules très intéressante pour ce que nous voulons faire.

L'architecture générale du démonstrateur MoP utilise abondamment les structures de modules de Caml [CMP00], celles-ci se prêtant particulièrement bien à notre objectif de définir différentes entités dont certaines seront abstraites et d'autres concrètes, certaines seront des paramètres et d'autres seront paramétrées.

Nous allons voir comment nous implémentons ici notre notion de termes symboliques par des termes abstraits, et comment nous implémentons ainsi la résolution sur des clauses abstraites.

4.1.1 Les termes abstraits

Nous appellerons *termes abstraits* l'implémentation des termes symboliques (et *termes concrets* l'implémentation d'instances de termes symboliques, comme nous le verrons au chapitre 5).

Les termes abstraits sont définis par une *signature*, qui désigne en Caml l'interface d'un module. L'établissement d'une telle signature va nous permettre de définir, selon un certain type abstrait `term`, les constructions dont nous avons besoin pour identifier les termes abstraits. Nous en donnons une simplification en figure 4.1 (pour plus de détails, voir annexe B).

```
module type TERM =
sig
     type term
     type subst
     type ordering = Greater | Equal | NotGE
     val app-sub : subst -> term -> term
     val comp-sub : subst -> subst -> subst
     val unify : (term * term) list -> subst list
     val match-term : (term * term) list -> subst list
     val order : term * term -> ordering
     val order-litt : (string * term list) -> (string * term list)
            -> ordering
     val sel : (string * term list) list -> (string * term list) list ->
     (string * term list) list
end;;
```

Fig. 4.1: Signature définissant les termes abstraits (version simplifiée)

Les termes abstraits sont ainsi définis de manière cohérente avec la définition des termes symboliques de la section 3.1.1. On remarquera tout de même que les termes symboliques T_A sont définis sur un ensemble de variables libres A, ce qui n'est bien sûr pas le cas des termes abstraits où le type `term` n'est pas contraint à être un terme pris dans un certain ensemble de variables.

Revenons sur le type des substitutions symboliques tel que nous l'avions défini en section 3.1.1, c'est-à-dire $(A \to T_B) \to (T_A \to T_B)$. Les substitutions symboliques sont ainsi vues comme un prolongement de fonctions, celui qui étend une fonction de substitution des variables vers les termes à une fonction des termes vers les termes. Ici, le type `subst` des substitutions, tout comme le type `term` et pour les mêmes raisons, est un type abstrait. Ainsi, le choix de l'implémentation des termes concrets et de leur fonction de substitution est libre. L'application d'une substitution abstraite à un terme (`app-sub`) est quant à elle assez naturelle, puisqu'elle retourne un terme. De même, le type de la composition des substitutions (`comp-sub`) est définie de manière de manière attendue.

Les substitutions étant ainsi définies, nous pouvons alors parler d'unification abstraite. Ce qui nous intéresse pour procéder à la résolution est de trouver un `csu` de deux littéraux, c'est-à-dire, pour deux littéraux ayant le même symbole de prédicat, de trouver les `csu` qui unifieront simultanément les couples formés des termes qui ont la même position dans la liste des arguments des littéraux respectifs. C'est pourquoi la signature de la fonction d'unification `unify` fait apparaître une liste de couples de termes. Si la liste des substitutions retournée par la fonction `unify` est une liste vide, l'unification de la liste de couples de termes abstraits correspondant est un échec. Nous nous restreignons ici de fait à des ensembles finis d'unificateurs, puisque ceux-ci sont implémentés par des listes finies, et donc à des théories finitaires (voir section 3.1.4).

Revenons brièvement sur la notion de subsomption abstraite que nous avons présentée en section 3.4.2. Nous disons qu'une clause symbolique C subsume une clause symbolique C' si et seulement si C' est de la forme $C\sigma \vee D$ pour un certain σ. Autrement dit, C subsume C' si et seulement si C' est de la forme $D' \vee D$ et il existe σ tel que $D' = C\sigma$. On dit alors que C *matche* D', et la procédure permettant de calculer ces substitutions est appelée le *matching*. Tout comme l'unification, le matching sur les clauses symboliques est une extension du matching sur les termes symboliques, et on dit que t matche t' si et seulement s'il existe $\sigma : fv(t) \to T_{fv(t')}$ telle que $Subst_{fv(t),fv(t')}(\sigma)(t) = t'$.

Nous avons donc besoin de définir une signature pour le matching qui fait apparaître une liste de couples de termes pour des raisons semblables à celles évoquées pour l'unification, qui retourne la liste vide en cas d'échec du matching, et une liste de substitutions dans le cas contraire.

Pour ce qui est de l'ordre sur les termes abstraits, nous supposons qu'il en existe un qui est stable de la même manière que pour les termes symboliques (section 3.1.4). Mais faisons ici une parenthèse concernant le choix que nous avons fait de la stabilité de l'ordre. C'est une condition qui peut en réalité être omise, dans le cas de la logique du premier ordre, sans que cela ne remette en cause ni la correction ni la complétude du raffinement de la résolution que nous utilisons, à condition d'avoir la stabilité par renommage [dN94], mais qui modifierait tout de même la règle de résolution comme nous le montre la figure 4.2.

$$\frac{C \vee +A_1 \vee \ldots \vee +A_n \quad C' \vee -A'}{(C \vee C')\sigma}$$

où :

(i) $n \geq 1$;

(ii) $\sigma \in \mathsf{csu}(A_1 \doteq A', \ldots, A_n \doteq A')$;

(iii) $\mathsf{sel}(C \vee +A_1 \vee \ldots \vee +A_n) = \emptyset$ et $A_1\sigma, \ldots, A_n\sigma$ sont maximaux dans $(C \vee +A_1 \vee \ldots \vee +A_n)\sigma$;

(iv) $-A' \in \mathsf{sel}(C' \vee -A')$, ou bien $\mathsf{sel}(C' \vee -A') = \emptyset$ et $A'\sigma$ est maximal dans $(C' \vee A')\sigma$.

FIG. 4.2: Résolution abstraite ordonnée avec sélection pour un ordre non nécessairement stable

Le changement notable dans cette règle de résolution pour un ordre non nécessairement stable est le test de maximalité qui s'effectue cette fois-ci sur des clauses substituées par un unificateur, contrairement à un ordre stable qui, par définition, nous permet de vérifier la maximalité sur les clauses non substituées. L'inconvénient majeur est alors que l'on doit au préalable effectuer la règle de résolution pour pouvoir ensuite tester la maximalité, ce qui évidemment fait perdre en efficacité du point de vue de l'implémentation. Nous avons donc choisi notamment pour cette raison de supposer la stabilité de l'ordre, l'autre étant que l'on ne rencontre en fait pas beaucoup d'ordres non stables qui soient utiles dans les cas pratiques. Il a cependant été prouvé que pour les ordres non stables (ou *non-liftables*), la résolution ordonnée avec factorisation est complète pour la logique du premier ordre [dN94], et qu'elle termine sur une certaine classe E^+ de clauses du premier ordre [FLTZ93], prouvant ainsi sa décidabilité.

Ici l'ordre est donc supposé stable, mais il n'est pas forcément total, certains littéraux pouvant ainsi ne pas être comparables. Le type `ordering` est donc défini ainsi : étant donnés deux termes t_1 et t_2, soit $t_1 \succ t_2$ (`Greater`), soit $t_1 = t_2$ (`Equal`), soit t_1 n'est ni supérieur ni égal à t_2 (`NotGE`).

La fonction `order-litt` sert à définir un ordre sur les littéraux, comme une extension de l'ordre sur les termes. L'ordre sur les littéraux est défini dans les modules de termes, et non dans le module de résolution où il serait défini comme une extension générique d'un ordre sur les termes. Ainsi, une plus grande liberté est laissée pour le choix de l'ordre (par exemple, étant donnés deux prédicats unaires $P(t)$ et $Q(t')$, l'ordre entre ces deux prédicats sera l'ordre entre les deux termes sauf si t et

t' sont incomparables ; dans ce cas l'ordre entre les deux prédicats est un ordre (lexicographique typiquement) sur les symboles de prédicats).

La fonction `sel` prend deux listes de type `(string * term list)` et renvoie une liste du même type. Ce type, comme nous le verrons en section suivante, est en fait le type des littéraux et la fonction `sel` sert à sélectionner les littéraux négatifs d'une clause. Nous avons donné quelques exemples assez simples de fonctions de sélection en section 1.3, celle qui retourne l'ensemble vide, celle qui retourne l'ensemble de tous les littéraux négatifs. Cependant, on peut avoir besoin de définir des sélections plus fines qui demandent par exemple d'interdire la sélection sur des littéraux dans lesquels apparaissent un symbole de fonction donné. La fonction de sélection est donc définie dans le module des termes. De plus, elle sera appelée avec deux listes : celle des littéraux négatifs et celle des littéraux positifs. Ceci sera utile dans le cas où l'on veuille sélectionner, parmi les littéraux négatifs, ceux qui sont maximaux dans l'ensemble de la clause. Ainsi, pour chaque module de termes concrets, on dispose d'une fonction de sélection.

4.1.2 La résolution abstraite

Comme nous l'avons décrit en section 3.3, la résolution abstraite opère sur des clauses formées avec des termes symboliques. Dans le prouveur modulaire, elle opère donc sur des clauses formées avec les termes abstraits tels que nous les avons défini à la section précédente, et le noyau la mettant en œuvre est représenté comme un *foncteur* de Caml, c'est-à-dire comme un module paramétré par un autre module.

Avant de décrire la signature de ce module, faisons une remarque : nous avons présenté en section 3.3 une règle de résolution abstraite qui est plus générale que celle que nous utilisons ici. En effet, nous utilisons ici le cas où $n = 1$, c'est-à-dire le cas où il n'y a que deux clauses en prémisse (nous allons voir un peu plus loin qu'en réalité nous n'utilisons même pas celle-ci, mais deux règles équivalentes). Cette règle est décrite en figure 4.3.

$$\frac{C \vee +A_1 \vee \ldots \vee +A_n \quad C' \vee -A'}{(C \vee C')\sigma}$$

où :

(i) $n \geq 1$;

(ii) $\sigma \in \mathsf{csu}(A_1 \doteq A', \ldots, A_n \doteq A')$;

(iii) $\mathsf{sel}(C \vee +A_1 \vee \ldots \vee +A_n) = \emptyset$ et $A_1, \ldots, A_n$ sont maximaux dans $C \vee +A_1 \vee \ldots \vee +A_n$;

(iv) $-A' \in \mathsf{sel}(C' \vee -A')$, ou bien $\mathsf{sel}(C' \vee -A') = \emptyset$ et A' est maximal dans $C' \vee A'$.

Fig. 4.3: Résolution abstraite ordonnée avec sélection réduite à deux prémisses

Cette règle nous suffira dans tous les cas pratiques que nous présenterons au chapitre 5. Cependant, nous verrons dans ce même chapitre à la section 5.3, que la règle de résolution abstraite ordonnée avec sélection dans le cas où $n > 1$ nous est utile dans une démonstration. En effet, l'application à la vérification du protocole cryptographique IKA.1 (section 6.1) nécessite de réaliser une approximation de la procédure de résolution. C'est la preuve de complétude de cette approximation qui utilise la règle de résolution ordonnée avec sélection pour $n > 1$, mais les moyens mis en œuvre dans l'approximation elle-même ne l'utilisent pas.

Revenons à la signature du module de résolution décrite en figure 4.4.

Le module de type RESOLUTION est donc défini comme un foncteur prenant en paramètre un module de type TERM.

La signature du module est somme toute sans surprise puisqu'elle contient les constructions que nous avions dégagées au cours de la description de la résolution abstraite.

En effet, un littéral est représenté comme un couple composé d'un symbole de prédicat (le type string), et d'une liste de termes abstraits qui sont les arguments, une clause est défini par un type

```
module type RESOLUTION =
     functor (Term : Term.TERM) ->
sig
     type litt = string * (Term.term list)
     type clause = { neg : litt list ;
                     pos : litt list }
     type type-res = Satis | Insatis
     val app-sub-to-clause : Term.subst -> clause -> clause
     val selec : clause -> litt list
     val facteurs : clause -> clause list
     val res-bin : (clause -> litt list) -> clause -> clause -> clause list
     val resolution : (clause -> litt list) -> clause list -> type-res
end ; ;
```

Fig. 4.4: Signature du module de résolution

enregistrement de Caml à deux champs : une liste de littéraux négatifs et une liste de littéraux positifs.

L'application d'une substitution à une clause est définie comme une extension de l'application d'une substitution à un terme abstrait, de la même manière que la substitution sur les termes symboliques.

Le type type-res est un type somme renvoyé par la fonction générale de résolution, c'est-à-dire la fonction implémentant l'algorithme général tel que nous allons le décrire en section 4.2.2, composé des types Satis et Insatis correspondant respectivement, comme leur nom l'indique, à la satisfiabilité et l'insatisfiabilité de l'ensemble de clauses considérée.

La fonction de sélection selec correspond exactement à la définition que nous en avions donnée : c'est une fonction qui étant donnée une clause retourne une liste de littéraux qu'il suffira de choisir négatifs. Précisons au passage que le type de la fonction resolution fait apparaître le type clause -> litt list qui est celui de la fonction de sélection. Cette fonction fera en fait appel à la fonction `sel` présente dans le module des termes abstraits, avec en argument la liste des littéraux négatifs de la clause, et celle des littéraux positifs.

Tout ceci étant précisé, nous pouvons remarquer que la signature fait apparaître deux fonctions facteurs et res-bin.

En effet, pour des soucis d'efficacité dans l'implémentation, nous avons fait le choix de décomposer la règle de résolution abstraite (section 3.3, figure 3.1) en deux règles : la *résolution binaire abstraite ordonnée avec sélection* et la *factorisation positive abstraite ordonnée* (figures 4.5 et 4.6), la fonction de sélection d'une clause symbolique devant alors être définie comme suit :

Définition – 4.1.1

Une fonction de sélection sel *est* stable *si et seulement si* $sel(C\sigma) = \{A\sigma \mid A \in sel(C)\}$.

En effet, la règle telle qu'elle est définie en section 3.3 nous contraint à trouver, étant donnée deux clauses $C = +A_1 \vee \ldots \vee +A_n \vee D$ et $C' = -A' \vee D'$ les ensembles possibles d'éléments pris dans les A_i, $1 \leq i \leq n$, tels que C et C' répondent aux conditions de la règle de résolution. Il nous est donc apparu beaucoup plus simple de la décomposer, d'autant plus, qu'en pratique les fonctions de sélection sont souvent stables.

Lemme – 4.1.1

L'utilisation de la règle de résolution abstraite ordonnée avec sélection stable est équivalente à l'utilisation des deux règles de résolution binaire abstraite ordonnée avec sélection stable et de factorisation positive abstraite ordonnée. Autrement dit, on peut dériver la clause vide à partir de S avec la seule règle de résolution abstraite ordonnée avec sélection stable si et

$$\frac{C \vee +A \quad C' \vee -A'}{(C \vee C')\sigma}$$

où :

(i) $\sigma \in \mathsf{csu}(A \doteq A')$;

(ii) $\mathsf{sel}(C \vee +A) = \emptyset$ et A est maximal dans $C \vee +A$;

(iii) $-A' \in \mathsf{sel}(C' \vee -A')$, ou bien $\mathsf{sel}(C' \vee -A') = \emptyset$ et A' est maximal dans $C' \vee A'$.

Fig. 4.5: Résolution binaire abstraite ordonnée avec sélection

$$\frac{C \vee +A \vee +A'}{(C \vee +A)\sigma}$$

où

(i) $\sigma \in \mathsf{csu}(A \doteq A')$;

(ii) $\mathsf{sel}(C \vee +A \vee +A') = \emptyset$;

(iii) A et A' sont maximaux dans $C \vee +A \vee +A'$.

Fig. 4.6: Factorisation positive abstraite ordonnée avec sélection

> *seulement si on le peut avec les règles de résolution binaire abstraite ordonnée avec sélection stable et de factorisation positive abstraite ordonnée avec sélection.*

Démonstration. Il suffit de remarquer que la clause $C \vee +A_1 \vee \ldots \vee +A_n$ peut se ramener en un nombre finie d'étapes de factorisation positive à la clause $(C \vee +A_1)\sigma$ (prémisse de la résolution binaire) pour un certain σ. La fonction de sélection étant stable, cela nous assure que les littéraux qui étaient sélectionnés dans C le sont toujours dans $C\sigma$. □

4.2 Algorithme de résolution

Notre démonstrateur utilise une stratégie de résolution abstraite ordonnée avec sélection, avec splitting et élimination de clauses redondantes. Nous avons montré que cette stratégie est correcte et complète (section 3.3), et nous allons utiliser ce résultat pour définir un algorithme.

Pour ce faire, nous nous sommes inspirés du prouveur par résolution sur des clauses du premier ordre présenté par L. Bachmair et H. Ganzinger [BG01b], qui utilise aussi la résolution ordonnée avec sélection, et qui procède par une recherche en largeur, ce qui est préférable puisqu'une recherche en profondeur est susceptible de ne pas terminer là où une recherche en largeur le ferait. C'est par exemple en profondeur que le moteur de résolution Prolog fonctionne. Certaines implémentations forcent la terminaison en fixant une borne sur la profondeur de l'arbre de recherche. Une autre approche intéressante est aussi celle suivie dans PTTP (Prolog Technology Theorem Prover) [Sti87], une extension de Prolog complète pour le calcul des prédicats du premier ordre, qui effectue une recherche en profondeur d'abord limitée par une borne itérative.

4.2.1 Remarques sur la subsomption

Il y a en fait deux façons d'utiliser la subsomption : la *subsomption avant* consiste à ne garder que les résolvants construits à partir d'un ensemble courant de clauses S qui ne sont subsumés par aucune clause de S ; la *subsomption arrière* consiste à effacer de l'ensemble courant S toutes les clauses subsumées par un résolvant C déduit à partir de clauses de S. On peut utiliser une combinaison des deux, à savoir la subsomption avant, puis la subsomption arrière si le nouveau résolvant a survécu. C'est ce qui est fait dans [BG01b], ainsi que dans notre prouveur.

Nous avions fait noter que les critères d'élimination de clauses subsumés peuvent être rendus plus libéraux (remarque 4). En effet, en modifiant la mesure utilisée dans la démonstration, nous avions remarqué que l'on peut éliminer d'un ensemble de clauses S toute clause C strictement subsumée par une autre clause C', ainsi que tout résolvant abstrait (ordonné, avec sélection) qui est subsumé par une autre clause symbolique de S. C'est cette approche que l'on trouve dans [BG01b].

Le prouveur opère sur un tuple $(\mathcal{N}_R, \mathcal{N}_F, \mathcal{P}r, \mathcal{O})$ de listes de clauses $\mathcal{N}_R$, $\mathcal{N}_F$, $\mathcal{P}r$ et $\mathcal{O}$, qui représente un état du système (voir section 4.2.2). Notons que [BG01b] utilise un ensemble $\mathcal{N}$ en lieu et place de nos deux listes $\mathcal{N}_R$ et $\mathcal{N}_F$. En effet, comme nous utilisons d'une part une règle de résolution binaire et d'autre part une règle de factorisation, nous avons différencié les clauses selon qu'elles sont déduites de l'une ou de l'autre de ces deux règles. Cela a une importance dans le cas de l'élimination des clauses subsumées, puisqu'une clause subsume toujours ses facteurs (par définition de la subsomption et de la règle de factorisation, la clause $C \vee +A \vee +A'$ en prémisse subsume la clause $C\sigma \vee +A\sigma$ en conclusion). Donc si on ne faisait pas cette distinction, ceci conduirait, dans le cas de la subsomption avant à éliminer tous les facteurs, et la complétude ne serait bien sûr plus assurée.

Dans le cas de la subsomption linéaire qui est celle que nous utilisons, la distinction entre les listes $\mathcal{N}_R$ et $\mathcal{N}_F$ n'est alors plus nécessaire puisque, par définition, une clause ne subsume pas linéairement ses facteurs. Cependant, toujours afin de laisser une ouverture à l'utilisateur qui souhaiterait utiliser la notion de subsomption stricte plutôt que celle de subsomption linéaire, nous avons tout de même fait le choix d'avoir deux listes distinctes pour les résolvants abstraits binaires (ordonnés, avec sélection) et pour les facteurs.

Nous pouvons remarquer au passage que le problème de la subsomption est un problème dur. En effet, dans le cas de termes du premier ordre, décider si une clause C est subsumée par une clause C' est un problème NP-complet [com79]. C'est d'autant plus gênant que nous ne sommes en fait pas intéressé par le problème de savoir si C est subsumée par une clause C' donnée, mais par une clause quelconque prise dans un ensemble qui peut être grand. Ce problème reste NP-complet, mais est plus long à résoudre en pratique. Il existe pour certaines logiques des techniques algorithmiques dédiées, notamment [RSV01] pour la logique du premier ordre, qui sont extrêmement efficaces. Cependant, nous n'utilisons pas de telles techniques puisque notre prouveur est modulaire et n'est donc pas dédié à une logique particulière. Nous verrons que nous obtenons tout de même des résultats satisfaisants (voir chapitre 6).

4.2.2 Algorithme

Le prouveur opère sur un tuple de listes $(\mathcal{N}_R, \mathcal{N}_F, \mathcal{P}r, \mathcal{O})$ qui représente un état du système de résolution. La liste $\mathcal{N}_R$ contient les clauses nouvellement déduites par résolution binaire, la liste $\mathcal{N}_F$ celles nouvellement déduites par factorisation positive, $\mathcal{P}r$ contient les clauses qui sont prêtes pour être résolues et factorisées, et la liste $\mathcal{O}$ contient les vieilles clauses qui ont déjà été résolues et factorisées. L'état initial du système est $(\mathcal{S}, \emptyset, \emptyset, \emptyset)$ pour un ensemble S fini de clauses.

Nous décrivons à la figure 4.7 la relation binaire $\Longrightarrow$ qui formalise notre prouveur par résolution, avec élimination des tautologies, élimination des clauses subsumées et splitting sans splitting. Bien que le prouveur manipule des listes, nous conserverons tout de même une notation ensembliste puisque ce sont des ensembles qu'elles décrivent.

Notons $Res^{\mathsf{sel}}_{\succ}(C, \mathcal{O})$ l'ensemble des résolvants binaires abstraits, ordonnés avec $\succ$ et sélectionnés avec sel, de C avec toutes les clauses de $\mathcal{O}$. Notons aussi $Fac^{\mathsf{sel}}_{\succ}(C)$ l'ensemble des facteurs abstraits positifs de C.

Les clauses nouvellement dérivées, c'est-à-dire celles des listes $\mathcal{N}_R$ et $\mathcal{N}_F$ sont examinées séparément, une liste après l'autre. Ainsi, si une clause de $\mathcal{N}_R$ (respectivement $\mathcal{N}_F$) est éliminée par *Élimination des tautologies* ou *subsomption avant*, on passe à l'examen de la clause suivante dans

Élimination des tautologies

$\mathcal{N}_R \cup \{C\} \mid \mathcal{N}_F \mid \mathcal{P}r \mid \mathcal{O} \Longrightarrow \mathcal{N}_R \mid \mathcal{N}_F \mid \mathcal{P}r \mid \mathcal{O}$

$\mathcal{N}_R \mid \mathcal{N}_F \cup \{C\} \mid \mathcal{P}r \mid \mathcal{O} \Longrightarrow \mathcal{N}_R \mid \mathcal{N}_F \mid \mathcal{P}r \mid \mathcal{O}$

si C est une tautologie

Subsomption avant

$\mathcal{N}_R \cup \{C\} \mid \mathcal{N}_F \mid \mathcal{P}r \mid \mathcal{O} \Longrightarrow \mathcal{N}_R \mid \mathcal{N}_F \mid \mathcal{P}r \mid \mathcal{O}$

$\mathcal{N}_R \mid \mathcal{N}_F \cup \{C\} \mid \mathcal{P}r \mid \mathcal{O} \Longrightarrow \mathcal{N}_R \mid \mathcal{N}_F \mid \mathcal{P}r \mid \mathcal{O}$

si une clause de $\mathcal{P}r \cup \mathcal{O}$ subsume linéairement C

Subsomption Arrière

$\mathcal{N}_R \mid \mathcal{N}_F \mid \mathcal{P}r \cup \{C\} \mid \mathcal{O} \Longrightarrow \mathcal{N}_R \mid \mathcal{N}_F \mid \mathcal{P}r \mid \mathcal{O}$

$\mathcal{N}_R \mid \mathcal{N}_F \mid \mathcal{P}r \mid \mathcal{O} \cup \{C\} \Longrightarrow \mathcal{N}_R \mid \mathcal{N}_F \mid \mathcal{P}r \mid \mathcal{O}$

si une clause de $\mathcal{N}_R \cup \mathcal{N}_F$ subsume linéairement C

Splitting sans splitting

$\mathcal{N}_R \cup \{C\} \mid \mathcal{N}_F \mid \mathcal{P}r \mid \mathcal{O} \Longrightarrow \mathcal{N}_R \cup \{C'_1, C'_2\} \mid \mathcal{N}_F \mid \mathcal{P}r \mid \mathcal{O}$

$\mathcal{N}_R \mid \mathcal{N}_F \cup \{C\} \mid \mathcal{P}r \mid \mathcal{O} \Longrightarrow \mathcal{N}_R \mid \mathcal{N}_F \cup \{C'_1, C'_2\} \mid \mathcal{P}r \mid \mathcal{O}$

si C est splittable C'_1 et C'_2

Traitement

$\mathcal{N}_R \cup \{C\} \mid \mathcal{N}_F \mid \mathcal{P}r \mid \mathcal{O} \Longrightarrow \mathcal{N}_R \mid \mathcal{N}_F \mid \mathcal{P}r \cup \{C\} \mid \mathcal{O}$

$\emptyset \mid \mathcal{N}_F \cup \{C\} \mid \mathcal{P}r \mid \mathcal{O} \Longrightarrow \emptyset \mid \mathcal{N}_F \mid \mathcal{P}r \cup \{C\} \mid \mathcal{O}$

si C n'est ni une tautologie, ni subsumée linéairement par une clause de $\mathcal{P}r \cup \mathcal{O}$

Déduction

$\emptyset \mid \emptyset \mid \mathcal{P}r \cup \{C\} \mid \mathcal{O} \Longrightarrow \mathcal{N}_R \mid \mathcal{N}_F \mid \mathcal{P}r \mid \mathcal{O} \cup \{C\}$

avec $\mathcal{N}_R = Res^{\mathsf{sel}}_{\succ}(C, \mathcal{O})$ et $\mathcal{N}_F = Fac^{\mathsf{sel}}_{\succ}(C)$

FIG. 4.7: Règles du démonstrateur

la liste $\mathcal{N}_R$ (respectivement $\mathcal{N}_F$), tant que celles-ci ne sont pas vides. Si au contraire elle n'est pas éliminée, on peut appliquer la règle (*Splitting sans splitting*). Rappelons que C est splittable en C'_1 et C'_2 par splitting sans splitting si $C'_1x = C_1 \vee +q$ et $C'_2 = -q \vee C_2$ pour un certain q est dérivable à partir de $C = C_1 \vee C_2$ par la règle d'inférence décrite en section 3.5.2 à la figure 3.4. La clause C est alors remplacée par les clauses C'_1 et C'_2 qui sont placées dans la liste $\mathcal{P}r$. En revanche, si elle n'est ni une redondance, ni splittable (dans le cas où on souhaite faire du splitting), elle est placée dans la liste $\mathcal{P}r$ (*Traitement*), où elle pourra encore se faire éliminer (*Subsomption arrière*).

Une fois ce pré-traitement effectué, les listes $\mathcal{N}_R$ et $\mathcal{N}_F$ étant vides, deux cas peuvent se présenter. Soit la liste $\mathcal{P}r$ est vide, l'ensemble S de clauses de départ est alors saturé, c'est-à-dire qu'aucune nouvelle clause non redondante ne pourra plus être déduite ; on peut donc conclure à la satisfiabilité de S. Soit $\mathcal{P}r$ est non vide, on procède alors à la phase de résolution proprement dite, qui consiste à déduire tous les résolvants binaires possibles d'une clause C prise dans $\mathcal{P}r$ avec les clauses de $\mathcal{O}$ (c'est-à-dire à générer la liste $Res^{\mathsf{sel}}_{\succ}(C, \mathcal{O})$), et tous les facteurs de C (c'est-à-dire la liste $Fac^{\mathsf{sel}}_{\succ}(C)$) (*Déduction*). Si la liste $Res^{\mathsf{sel}}_{\succ}(C, \mathcal{O})$ contient la clause vide, l'ensemble S des clauses de départ est insatisfiable, et le processus se termine. Dans le cas contraire, on réitère le processus, en vérifiant que C n'est pas devenue une clause redondante (*Subsomption arrière*).

La distinction entre les listes $\mathcal{P}r$ et $\mathcal{O}$ sert à garantir une équité dans le processus de résolution : ainsi aucune clause ne sera oublié par la règle de *Déduction*.

Cette procédure est ainsi définie de manière récursive sur les quatre listes de clauses décrites ci-dessus. Nous pouvons à ce stade faire quelques remarques.

Par la règle de *Subsomption avant*, toute clause de $\mathcal{N}_R$ linéairement subsumée par une clause de $\mathcal{P}r$ ou par une vieille clause est éliminée.

Les règles de *Subsomption arrière* présentent deux aspects : la première élimine toute clause de $\mathcal{P}r$ linéairement subsumée par une clause de $\mathcal{N}_R \cup \mathcal{N}_F$. Ceci revient à dire qu'elle élimine toute clause de $\mathcal{N}_R \cup \mathcal{N}_F$ qui n'est pas éliminée par les règles *Élimination des tautologies* ou *Subsomption avant*, mais qui est subsumée linéairement par une clause C de $\mathcal{N}_R \cup \mathcal{N}_F$. La deuxième élimine de l'ensemble des vieilles clauses toutes clauses linéairement subsumée par une clause nouvellement créée.

Remarque 5 *Nous avions défini en section 1.4 la notion de clauses symboliques pures. Nous avons cependant fait le choix de ne pas procéder à l'élimination de ces clauses dans le prouveur car il nous semble qu'en pratique, même si nous ne l'avons pas mesuré, il y en a peu, et ce serait certainement équivalent en efficacité, voire plus coûteux, de les éliminer plutôt que de les garder.*

Remarque 6 *Nous avons aussi défini en section 3.5.1 le splitting par tableau, bien que celui-ci n'apparaisse pas dans notre prouveur. C'est en fait une amélioration que nous aimerions apporter, mais nous y reviendrons en section 6.2.2.*

Nous avons donc présenté un prouveur modulaire basé sur la notion de résolution abstraite ordonnée avec sélection que nous avions établi au chapitre précédent. Il existe cependant une différence importante en ce qui concerne les termes symboliques. En effet, comme nous l'avions déjà fait remarquer en section 4.1.1, la représentation des termes symboliques ne fait pas apparaître d'ensembles de variables, puisque le type `term` est un type abstrait, le choix étant fait de définir ces variables dans l'implémentation des modules de langages de termes, laissant alors une plus grande ouverture et respectant le principe de modularité du prouveur.

Nous avons montré que la résolution abstraite ordonnée avec fonction de sélection et application à tout moment du processus de résolution des règles d'élimination de tautologies et de clauses subsumées linéairement, et de la règle de splitting sans splitting est correcte et complète. Ainsi, toute stratégie qui utilise la règle $\Longrightarrow$ définit un algorithme de résolution correct et complet.

Nous disposons ainsi d'un démonstrateur dont l'algorithme est correct et complet sur des termes abstraits. Nous allons voir en pratique dans le chapitre suivant quels sont les modules de termes que nous avons implémentés, modules que nous appellerons *concrets* par opposition claire au module des termes abstraits.

Chapitre 5

Les différents modules

Nous allons ici présenter nos différents modules, c'est-à-dire nos termes concrets, et veiller à ce que les conditions données en section 3.3 soient satisfaites.

5.1 Termes finis du premier ordre

Nous savons bien entendu que la logique du premier ordre n'est pas décidable dans le cas général, mais il existe beaucoup de fragments de cette logique pour lesquels la décidabilité a été montrée, dont certains ont été décrits en section 1.6.

Comme nous l'avons déjà dit, il existe de nombreux prouveurs dédiés à la logique du ordre qui sont très efficaces, alors que notre prouveur met l'accent sur l'ouverture et la modularité. Nous avons fait le choix d'implémenter les termes finis du premier ordre, non pas pour rivaliser avec ceux qui existent, mais pour nous permettre d'utiliser ce cas bien connu pour lequel il existe beaucoup de problèmes établis, comme les fameux *Seventy-five problems for testing automatic theorem provers* de F.J. Pelletier [Pel86] dont nous nous sommes amplement servis.

Nous avons montré en section 3.1.2.1 que les termes finis du premier ordre sont une famille normale de termes symboliques. Ainsi, les différents résultats que nous avons établi dans le cadre général des termes symboliques sont utilisables pour les termes du premier ordre qui constituent un module du prouveur.

Nous allons rappeler brièvement la définition de ces termes et des substitutions, beaucoup de notions ayant déjà été introduites au cours du chapitre 1 consacré aux termes du premier ordre, puis nous présenterons l'algorithme d'unification que nous avons implémenter.

Fixons un ensemble $\mathcal{V}$ infini dénombrable de symboles de variables, un ensemble $\mathcal{F}$ infini dénombrable de symboles de fonctions d'arité fixée, et un ensemble $\mathcal{P}$ infini dénombrable de symboles de prédicats. Les éléments de $\mathcal{V}$, $\mathcal{F}$ et $\mathcal{P}$ seront notés respectivement x, y, z et f, g, h et P, Q, R.

Rappelons qu'un *terme du premier ordre* est défini par :

- x est un terme ;
- si $t_1, \ldots, t_n$ sont des termes, alors pour tout $f \in \mathcal{F}$ d'arité n, $f(t_1, \ldots, t_n)$ est un terme.

Nous allons maintenant présenter l'algorithme d'unification utilisé sur les termes du premier ordre ([GLM97]) en figure 5.1

Initialement, pour unifier deux termes s et t, on démarre l'algorithme avec $(\{s \doteq t\}, [])$. Ensuite, on choisit une règle à appliquer selon une certaine stratégie, jusqu'à ce qu'aucune règle ne puisse plus être appliquée. La stratégie est la suivante : si F est vide, alors σ est le mgu désiré ; sinon, on choisit une équation $t \doteq t'$ dans F (où les équations $s \doteq t$ et $t \doteq s$ sont identifiées) : si $t = t'$ (règle **(Effacement)**), alors l'équation est déjà résolue, et on peut à nouveau en choisir une autre à résoudre. Si t et t' sont tout deux des applications du même symbole de fonction, alors seule la règle **(Decomposition)** s'applique, et on remplace $t \doteq t'$ par les équations indiquées. Si t ou t' est une variable, disons t, alors si t n'est pas dans le domaine de σ, on applique la règle **(Liaison)**

(Effacement)	$(F \cup \{t \doteq t'\}, \sigma) \rightarrow (F, \sigma)$ si $t = t'$
(Decomposition)	$(F \cup \{t \doteq t'\}, \sigma) \rightarrow (F \cup \{t_1 \doteq t'_1, \ldots, t_m \doteq t'_{m'}\}, \sigma)$ si $t \neq t'$ et $t = f(t_1, \ldots, t_m)$, $t' = f'(t'_1, \ldots, t'_{m'})$ avec $f = f'$ et $m = m'$
(Liaison)	$(F \cup \{x \doteq t\}, \sigma) \rightarrow (F, [x/t] \cup \sigma)$ si $x \notin \mathsf{dom}(\sigma) \cup \mathsf{fv}(\sigma(t))$
(Fusion)	$(F \cup \{x \doteq t\}, \sigma) \rightarrow (F \cup \{\sigma(x) \doteq t\}, \sigma$ si $\sigma(x) \neq t$, $x \in \mathsf{dom}(\sigma)$

Fig. 5.1: Règles pour l'unification des termes finis du premier ordre

pour lier t à t' ; si t est dans le domaine de σ, on applique la règle **(Fusion)**. Si aucune règle ne s'applique et que F n'est pas vide, alors s et t ne sont pas unifiables.

5.2 Termes cycliques du premier ordre

Les termes cycliques du premier ordre, aussi appelés termes infinis rationnels, sont des termes du premier ordre possiblement infinis, mais réguliers dans leur infinité. Nous en donnons quelques exemples en figure 5.2.

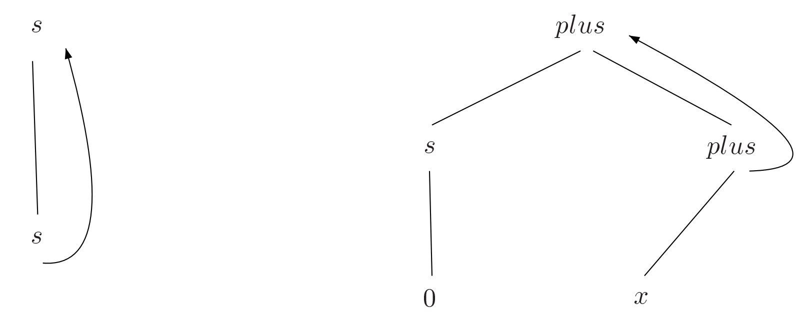


Fig. 5.2: Exemples de termes rationnels

L'algorithme d'unification est le même algorithme que pour les termes finis, excepté qu'on n'a pas besoin de vérifier qu'une variable n'apparaît pas dans un terme dans la règle **(Bind)** (cf figure 5.1).

Par exemple, soient $t_1 = f(x, g(x))$ et $t_2 = f(h(y, z), y)$. Ici, l'unification sur les termes finis échoue après avoir unifié les premiers arguments de chaque terme, on a une substitution qui à x associe $h(y, z)$, et pour unifier les deuxièmes arguments il faudrait associer à y le terme $g(x)$ ce qui n'est pas possible car y apparaît déjà dans le co-domaine de la substitution. Par contre, l'unification sur les termes rationnels réussit puisque nous n'avons pas besoin de cette dernière contrainte. La représentation graphique des termes t_1 et t_2 unifiés est donnée en figure 5.3.

Ces termes sont bien connus, mais il n'existe pas à notre connaissance de travaux relatant la manière de les implémenter.

Nous avons donné en section 3.1.2.3 une façon de définir les termes infinis possiblement rationnels, et nous avons montré que l'ensemble de ces termes est une famille de termes symboliques. Nous allons ici en donner une autre définition (section 5.2.1) qui sera intuitivement plus proche

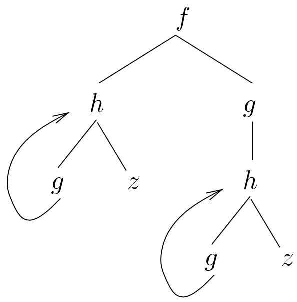


FIG. 5.3: Représentation graphique de $t_1\sigma = t_2\sigma$

d'une représentation graphique (voir figure 5.2), et mieux représentable d'un point de vue implémentation. Nous présenterons ensuite l'algorithme d'unification sur ces termes ainsi définis.

5.2.1 Une définition des termes rationnels

La définition des termes infinis que nous avons donnée en section 3.1.2.3 est de manière évidente peu utilisable pour une implémentation.

Nous avons donc cherché à donner une définition des termes cycliques (qui sont des termes infinis rationnels) qui soit plus exploitable. Nous aurions pu les définir en ajoutant un symbole de fonction spécial d'arité 1 (link) qui prendrait en argument un entier naturel pour représenter les cycles d'un terme, jouant ainsi le rôle d'un pointeur. Pour donner une idée, les deux termes donnés en figure 5.2 peuvent s'écrire $s(\mathsf{link}\ 1)$ et $plus(s(0), plus(x, \mathsf{link}\ 1))$. Mais cette définition poserait des différents problèmes tant d'un point de vue formel (il faudrait ajouter des contraintes sur l'argument des link pour que les termes soient bien formés par exemple), que du point de vue implémentation (les termes de la figure 5.2 peuvent aussi s'écrire $s(s(\mathsf{link}\ 1))$ et $plus(s(0), plus(x, plus(s(0), plus(x, \mathsf{link}\ 3))))$, ce qui soulève d'ailleurs un autre problème d'ordre formel, celui de la définition d'une théorie équationnelle assez complexe tant sur les termes que sur les substitutions).

Nous avons fait le choix de définir les termes cycliques comme des graphes orienté représentés par des listes d'adjacence.

DÉFINITION – 5.2.1

> *Un* graphe *est défini comme étant la donnée :*
> - *d'un entier positif que nous appellerons la* racine*,*
> - *d'une structure G qui à un entier positif, que nous appellerons une clef, associe :*
>   * *soit un couple composé d'un symbole de fonction de $\mathcal{F}$ et d'une liste d'entiers qui représentent la liste des arguments de la fonction (possiblement vide) pour lesquelles il existe une clef,*
>   * *soit une variable.*
>
> *Un graphe de racine r et de structure G sera noté (r, G).*

Nous appellerons *domaine de G* et noterons $\mathsf{dom}(G)$ l'ensemble des clefs accessibles à partir de la racine.

Nous allons définir une traduction des graphes vers les termes rationnels tels qu'ils sont décrits en section 3.1.2.3. Pour cela nous définissons un ensemble de position, ainsi qu'une application qui à une position donnée associe un symbole de fonction ou une variable.

Pour un graphe (r, G) donné, définissons l'ensemble $Pos_G(r)$ comme le plus petit ensemble de mots tel que :
- $Pos_G(r) = \{\epsilon\}$ si $G(r) = x$;
- $Pos_G(r) = \{\epsilon\} \cup \{i.p \mid 1 \leq i \leq n,\ p \in Pos_G(r_i)\}$ si $G(r) = (f, [r_1, \ldots, r_n])$.

Soit $r.w$ tel que $w \in Pos_G(r)$, on défini le *sommet* par récurrence sur w comme suit :

– $r.\epsilon = r$;
– si $w \neq \epsilon$ et $w = i.p$, posons $r.w = r_i.p$, alors $G(r.w) = (f, [r_1, \ldots, r_n])$, $1 \leq i \leq n$.

Posons alors $lab_{G,r}(w) = \begin{cases} f \text{si } G(r.w) = (f, [r_1, \ldots, r_n]) \\ x \text{si } G(r.w) = x \end{cases}$

Ainsi, étant donné un graphe (r, G), nous disposons d'un ensemble de positions $Pos_G(r)$ et d'une fonction $lab_{G,r}$. Posons alors $Rat(r, G) = (Pos_G(r), lab_{G,r})$. Ainsi, un graphe peut être vu comme un terme rationnel qui est un terme symbolique.

Définition – 5.2.2

> *Un terme cyclique t est une classe d'équivalence du graphe (r, G) modulo la relation $\equiv$ définie par $(r, G) \equiv (r', G')$ si et seulement si $Rat(r, G) = Rat(r', G')$.*

Les termes cycliques ainsi définis sont donc une famille normale de termes symboliques, puisque nous avons vu en section 3.1.2.2 qu'une famille de termes symboliques modulo une théorie équationnelle est une famille de termes symboliques.

L'*ensemble des sous-termes* d'un terme $t = (r, G)$ est l'ensemble $\{(k, G) \mid k \in dom(G)\}$.

Exemple 1 *: Représentation des termes.*
Soit t le défini par :

$$\begin{array}{lll} (\ 2, & 2 \mapsto (g, [3]) & \\ & 3 \mapsto (f, [4; 3]) & \\ & 4 \mapsto x &) \end{array}$$

t peut aussi s'écrire de la façon suivante :

$$\begin{array}{lll} (\ 2, & 1 \mapsto (h, [2]) & \\ & 2 \mapsto (g, [3]) & \\ & 3 \mapsto (f, [4; 3]) & \\ & 4 \mapsto x &) \end{array}$$

la clef 1 n'étant de toute façon pas accessible. Ces deux graphes sont égaux modulo la relation $\equiv$. On peut aussi remarquer que le terme :

$$\begin{array}{lll} (\ 3, & 2 \mapsto (g, [3]) & \\ & 3 \mapsto (f, [4; 3]) & \\ & 4 \mapsto x &) \end{array}$$

est un sous-terme de t qui est égal modulo $\equiv$ au terme :

$$\begin{array}{lll} (\ 3, & 3 \mapsto (f, [4; 3]) & \\ & 4 \mapsto x &) \end{array}$$

5.2.2 Unification

L'algorithme d'unification est décrit en figure 5.4. C'est une variante d'algorithmes d'unification existants [JK90]. Soient $t = (r, G)$ et $t' = (r', G')$ deux termes cycliques à unifier avec $dom(G) \cap dom(G') = \emptyset$, et soit H un ensemble de couples de clefs initialement vide représentant l'historique des couples examinés. L'idée de l'algorithme est de parcourir simultanément les deux graphes en unifiant les sous-termes dès que cela est possible. Les termes étant cyclique, on pourrait parcourir indéfiniment les même sous-termes, c'est pour cette raison que nous nous servons de l'ensemble H comme d'un historique des couples de clefs qui ont déjà été examinées. Ainsi, dès qu'un couple à examiner est dans l'ensemble H, on peut l'ignorer. C'est ce que décrit la règle **(Cycle)**. De plus, toutes les autres règles ont pour effet de rajouter le couple courant de clefs à l'ensemble H. Les autres règles sont très similaires à celles de l'algorithme d'unification des termes finis du premier ordre.

(Cycle)	$(F \cup \{(k,G)\doteq(k',G')\}, H, \sigma) \longrightarrow (F, H, \sigma)$
	si $(k,k') \in H$
(Effacement)	$(F \cup \{(k,G)\doteq(k',G')\}, H, \sigma) \longrightarrow (F, H \cup \{(k,k')\}, \sigma)$
	si $G(k) = G'(k') = x$
(Decomposition)	$(F \cup \{(k,G)\doteq(k',G')\}, H, \sigma)$
	$\longrightarrow (F \cup \{(k_1,G)\doteq(k'_1,G'), \ldots, (k_m,G)\doteq(k'_m,G')\}, H \cup \{(k,k')\}, \sigma)$
	si $G(k) = (f, [k_1; \ldots; k_m])$ et $G'(k') = (f', [k'_1; \ldots; k'_{m'}])$
	avec $f = f'$ et $m = m'$
(Liaison)	$(F \cup \{(k,G)\doteq(k',G')\}, H, \sigma) \longrightarrow (F, H \cup \{(k,k')\}, [x/(k',G')] \circ \sigma)$
	si $G(k) = x$, $G'(k') \neq x$, et $x \notin \mathsf{dom}(\sigma)$
(Fusion)	$(F \cup \{(k,G)\doteq(k',G')\}, H, \sigma) \longrightarrow (F \cup \{\sigma(k',G')\}, H \cup \{(k,k')\}, \sigma)$
	si $G(k) = x$, $G'(k') \neq x$, et $x \in \mathsf{dom}(\sigma)$

FIG. 5.4: Règles pour l'unification des termes rationnels du premier ordre

5.3 Automates d'arbres bidirectionnels modulo AC

Notre motivation dans cette section, comme nous le verrons au chapitre 6, est de modéliser le protocole cryptographique IKA.1 par un ensemble de clauses de Horn. Les résultats présentés dans [GLV02] sur les automates d'arbres bidirectionnels modulo associativité et commutativité prennent ce protocole pour exemple, en montrant qu'il peut se modéliser sous la forme d'un tel automate d'arbre.

Nous avons vu en section 1.7 comment représenter des automates d'arbres par un ensemble de clauses de Horn, et plus particulièrement les automates d'arbres bidirectionnels. Nous allons ici nous servir du résultat [GLV02] présenté en section 2.4.3, à savoir la décidabilité des ensembles de clauses de Horn représentant des AC-automates non alternants bidirectionnels avec clauses intersection finales.

Cependant, l'approche qui consisterait à utiliser la résolution ordonnée avec sélection utilisant l'unification modulo associativité et commutativité est en pratique beaucoup trop coûteuse pour être exploitable.

Nous allons tout d'abord présenter le langage des termes qui sont des termes finis du premier ordre avec opérateurs associatif et commutatif. Les résultats de la section 3.1.2.2 sur les familles de termes symboliques modulo une théorie équationnelle nous assure que cet ensemble de termes est bien une famille de termes symboliques. Puis nous rappellerons les formes de clauses représentant les automates d'arbres (section 5.3.1), et enfin nous présenterons ensuite en section 5.3.2 une approximation que nous avons effectuée pour la résolution de tels ensembles de clauses.

5.3.1 Termes et clauses avec opérateur associatif et commutatif

L'ensemble des termes que nous allons ici considérer est un ensemble de termes finis du premier ordre modulo associativité et commutativité ; ces termes sont appelés *termes plats* :

$$\begin{array}{rcl} t & ::= & x \\ & \mid & f(x_1, \ldots, x_n) \\ & \mid & x_1 \oplus \ldots \oplus x_m \end{array}$$

où $x, y, x_1, \ldots, x_n$ désignent des variables, f est un symbole de fonction, $\oplus$ est un opérateur binaire associatif et commutatif avec un élément neutre noté 0, défini par la théorie équationnelle E suivante :

$$E : \quad \begin{array}{rcl} x \oplus y & = & y \oplus x \\ (x \oplus y) \oplus z & = & x \oplus (y \oplus z) \\ x \oplus 0 & = & x \end{array}$$

Les symboles de prédicat utilisés sont tous des prédicats unaires de la forme $P(t)$ où t est un terme plat.

Afin d'illustrer le type de clauses dont nous aurons besoin, plaçons nous dans le cadre des protocoles cryptographiques en prenant l'exemple suivant :

Exemple 2 *Supposons qu'un intrus de Dolev-Yao a réussi à récupérer un ensemble de messages définis par un prédicat E – autrement dit, le message t est récupéré si et seulement si $E(t)$ est dans le plus petit modèle de Herbrand d'un ensemble de clauses donné –, alors le prédicat I défini par les clauses additionnelles :*

$$\begin{aligned} I(X) &\Leftarrow E(X) \\ I(c(X,Y)) &\Leftarrow I(X), I(Y) \\ I(X) &\Leftarrow I(c(X,Y)), I(Y) \\ I(p(X,Y)) &\Leftarrow I(X), I(Y) \\ I(X) &\Leftarrow I(p(X,Y)) \\ I(Y) &\Leftarrow I(p(X,Y)) \\ I(X \oplus Y) &\Leftarrow I(X), I(Y) \end{aligned}$$

est tel que $I(t)$ est vrai dans le plus petit modèle de Herbrand si et seulement si t est déductible par l'intrus de Dolev-Yao, en lisant $c(u,v)$ "u chiffré avec la clef symétrique v" et $p(u,v)$ "paire u,v". La première clause exprime que tout message récupéré est déductible, et c'est une ϵ-transition. La deuxième exprime que l'intrus peut déduire le chiffrement de deux messages déjà déduits, et c'est une transition d'automate d'arbres (2.1). La troisième exprime que l'intrus peut déchiffrer un message s'il a déjà déduit le message chiffré correspondant et la clef symétrique ayant servie au chiffrement ; et c'est une clause de la forme (2.4). La quatrième clause exprime que l'intrus peut former une paire de messages s'il les a déjà déduit ; c'est une clause de la forme (2.1). Les deux suivantes expriment que l'intrus peut déduire un message s'il a déjà déduit une paire le contenant ; ce sont des clauses de la forme (2.4). La dernière implique que l'intrus peut additionner des messages des messages qu'il a déjà déduit, si l'on considère le symbole $\oplus$ comme l'addition usuelle

Nous rappelons en figure 5.5, comme nous l'avions vu en section 2.4.3, le format général des clauses que nous utiliserons.

$P(f(x_1,\ldots,x_n)) \Leftarrow P_1(x_1),\ldots,P_n(x_n)$	clause pop	(2.1)
$P_i(x_i) \Leftarrow P(f(x_1,\ldots,x_n)), P_{i_1}(x_{i_1}),\ldots,P_{i_k}(x_{i_k})$ $i \notin \{i_1,\ldots,i_k\}$	clause push	(2.4)
$P(x_1 \oplus \ldots \oplus x_n) \Leftarrow P_1(x_1),\ldots,P_n(x_n)$	clause $\oplus$-pop	(2.7)
$P(x_i) \Leftarrow P_1(x_1 \oplus \ldots \oplus x_n)$	clause $\oplus$-push	(2.8)
$\perp \Leftarrow P_1(x),\ldots,P_k(x)$	clause intersection finale	(2.9)

où les symboles de fonction f sont libres, c'est-à-dire qu'ils sont différents du symbole $\oplus$.

Fig. 5.5: Format des clauses

5.3.2 Approximation sur $\mathbb{Z}/n\mathbb{Z}$

L'intérêt théorique du format des clauses présenté en figure 5.5 est que la satisfiabilité de tels ensembles de clauses est décidable [GLV02]. De plus, on verra au chapitre 6 que l'on peut coder au moins une application de vérification de protocole cryptographique, utilisant de manière non triviale un opérateur associatif et commutatif $\oplus$, dans ce format.

Il serait donc en principe intéressant de réaliser une implémentation de l'algorithme de décision de [GLV02] pour décider notamment de la sécurité de tels protocoles cryptographiques. En pratique, ceci se heurte cependant à de nombreux obstacles. En premier, l'algorithme de [GLV02] est largement théorique, et se fonde à plusieurs endroits sur des énumérations de candidats solutions, en nombre fini certes mais gigantesque. Notamment, l'algorithme de [GLV02] demande à énumérer répétitivement toutes les clauses de Horn de la forme :

$$B_1(x_1) \vee B_2(x_2) \vee \ldots \vee B_n(x_n) \tag{5.1}$$

où $x_1, x_2, \ldots, x_n$ sont des variables distinctes, où les $B_i(x_i)$ sont des blocs, et où de plus $B_i(x) \neq B_j(x)$ pour tous $i \neq j$. Rappelons qu'un bloc $B(x)$ est une disjonction

$$\pm_1 P_1(x) \vee \ldots \vee \pm_k P_k(x)$$

En supposant que nous ayons p symboles de prédicats dans l'ensemble de clauses de départ, il existe 2^p blocs négatifs (de Horn, de tête vide), $p2^p$ blocs définis (de Horn, de tête non vide), et donc 2^{2^p} clauses négatives de la forme (5.1) et $p2^p2^{2^p}$ clauses définies de la forme (5.1), pour un total de $(p2^p+1)2^{2^p}$ clauses de Horn possibles de la forme (5.1).

L'algorithme de [GLV02], de plus, demande à vérifier pour chacune des clauses de Horn C de la forme (5.1) que $S' \models^{std}_{\mathrm{AC}} C$, où, S étant l'ensemble courant de clauses, S' est un ensemble de clauses déduites de S, et $S' \models^{std}_{\mathrm{AC}} C$ signifie que C est valide dans tout modèle dit standard (voir [GLV02]) de S' qui interprète $\oplus$ comme un opérateur associatif et commutatif. Il est à noter qu'ici C n'est *pas* restreinte à être une clause de l'ensemble courant S, et doit être devinée parmi l'ensemble de taille doublement exponentielle des clauses de Horn de la forme (5.1). La vérification de la condition $S' \models^{std}_{\mathrm{AC}} C$ est elle-même très complexe, puisqu'elle fait appel notamment à des techniques de résolution de formules de l'arithmétique de Presburger et à une généralisation de la construction des arbres de couverture de Karp et Miller, tels qu'utilisés dans la théorie des réseaux de Petri.

Il est donc nécessaire, si l'on souhaite attaquer la vérification d'ensembles de clauses du format décrit en figure 5.5, de définir des procédures d'approximation plus efficaces.

C'est ce que nous allons faire. Examinons les différentes facettes du problème :

– En premier, la résolution ordonnée avec fonction de sélection portant sur des termes modulo la théorie AC est complète ; c'est une conséquence du théorème de complétude 3.3.2 sur les clauses symboliques, et du fait que l'ensemble des termes du premier ordre modulo AC définit une famille de termes symboliques normale (proposition 3.1.4).
– Mais la résolution ordonnée avec sélection ne termine pas en général sur les clauses de la figure 5.5. La raison en est purement l'interaction entre les clauses dont les termes ne sont construits qu'avec le symbole $\oplus$ — clauses que l'on appellera les $\oplus$-*clauses* — qui construisent par résolution des clauses de plus en plus grosses, sans que l'on puisse borner leur taille. La terminaison n'est donc pas garantie.

  À titre d'exemple, considérons les deux clauses :

$$P(x') \Leftarrow Q'(x'+y') \qquad Q'(x+y) \Leftarrow Q(x), I(y)$$

  Un ensemble complet d'unificateurs $\mathsf{csu}(x'+y' \doteq x+y)$, de cardinal minimal, consiste en les sept substitutions suivantes, où x_1, x_2, y_1, y_2 sont des variables fraîches et deux à deux distinctes :

  1. $[x := x_1 + y_1, y := x_2 + y_2, x' := x_1 + x_2, y' := y_1 + y_2]$;
  2. $[y := x' + y_2, y' := x + y_2]$;
  3. $[y := y' + x_2, x' := x + x_2]$;
  4. $[x := x' + y_1, y' := y + y_1]$;
  5. $[x := y' + x_1, x' := y + x_1]$;
  6. $[x' := y, y' := x]$;
  7. $[x' := x, y' := y]$.

  On obtient ainsi sept résolvants des deux clauses ci-dessus :

  1. $P(x_1 + x_2) \Leftarrow Q(x_1 + y_1), I(x_2 + y_2)$

2. $P(x') \Leftarrow Q(x), I(x' + y_2)$
3. $P(x + x_2) \Leftarrow Q(x), I(y' + x_2)$
4. $P(x') \Leftarrow Q(x' + y_1), I(y)$
5. $P(y + x_1) \Leftarrow Q(y' + x_1), I(y)$
6. $P(y) \Leftarrow Q(x), I(y)$
7. $P(x) \Leftarrow Q(x), I(y)$

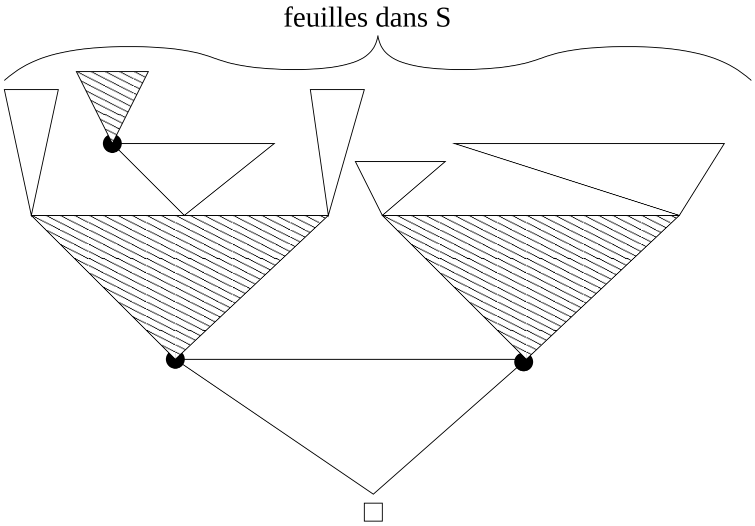
On peut splitter les clauses 2, 4, 6, et 7, ce qui mène à :

$$\begin{array}{rl} P(x') \Leftarrow I(x' + y_2), q_1(*) & \qquad q_1(*) \Leftarrow Q(x) \\ P(x') \Leftarrow Q(x' + y_1), q_2(*) & \qquad q_2(*) \Leftarrow I(y) \\ P(y) \Leftarrow I(y), q_1(*) & \\ P(x) \Leftarrow Q(x), q_2(*) & \end{array}$$

par splitting sans splitting, où $q_1 = \ulcorner -Q(x) \urcorner$, $q_2 = \ulcorner -I(y) \urcorner$. Cependant, les clauses 1, 3, et 5 ne sont pas du format de la figure 5.5, et la fabrication de nouveaux résolvants à partir de ces dernières clauses produit des clauses de plus en plus volumineuses, et impliquant des nombres de plus en plus grands de variables.
- La résolution ordonnée avec fonction de sélection entre clauses qui ne sont pas des $\oplus$-clauses ne souffre pas de ce problème : à partir de clauses simples ou complexes ne faisant pas intervenir $\oplus$, on n'obtient encore que des clauses simples ou complexes, modulo splitting. De plus, les clauses complexes ne faisant pas figurer $\oplus$ ne se résolvent avec aucune $\oplus$-clause. On peut donc séparer les traitements des $\oplus$-clauses de celui des autres clauses, à ceci près que les blocs doivent être utilisés à la fois dans les deux traitements.
- Pour traiter du cas des $\oplus$-clauses, nous allons concevoir une procédure qui sera en général incorrecte mais complète. Par cela, nous entendons que si C est une clause (dans un ensemble à définir) qui est déductible des $\oplus$-clauses de l'ensemble courant S, alors C devra être déductible par notre règle d'approximation. Nous interdirons ensuite la règle usuelle de résolution ordonnée avec sélection sur les $\oplus$-clauses. Ainsi, toute clause déductible par résolution ordonnée avec sélection le sera par notre procédure, ce qui implique qu'elle pourra déduire la clause vide de tout ensemble insatisfiable. De plus, si nous pouvons montrer que la nouvelle règle d'approximation ne produit qu'un nombre fini de clauses, alors notre procédure terminera.

L'essentiel dans cette approche est de définir la nouvelle règle d'approximation s'appliquant aux $\oplus$-clauses. Or l'esprit des résultats de [GLV02] est que, partant d'un ensemble de clauses S qui sont soit des clauses simples soit des clauses complexes (figurant possiblement $\oplus$), la seule utilité de la résolution ordonnée avec sélection lorsqu'on l'applique à des $\oplus$-clauses est de finir par produire, après un nombre fini (possiblement grand) d'étapes, une disjonction de blocs telle qu'en (5.1).

Illustrons ceci à l'aide d'arbres de preuve. En simplifiant abusivement (l'argument qui suit est censé apporter une certaine intuition, mais est erroné), considérons pour l'exemple que la seule règle que nous utilisons est la règle de résolution binaire ordonnée avec sélection. On peut définir un arbre de preuve comme un arbre dont les nœuds sont des clauses C, les feuilles sont des clauses de l'ensemble de clauses de départ S, et les autres nœuds C ont deux parents C_1 et C_2, tels que C se déduise de C_1 et C_2 par résolution binaire ordonnée avec sélection. Regroupons les nœuds de l'arbre de preuve qui sont des $\oplus$-clauses non triviales (avec au moins une occurrence de $\oplus$). Une dérivation de la clause vide par résolution binaire ordonnée avec sélection à partir de S aura donc typiquement la forme :



où les feuilles (les clauses de S) sont par convention en haut, la conclusion (la clause vide) est la racine de l'arbre, en bas, les zones hachurées représentent des résolutions binaires ordonnées avec sélection entre $\oplus$-clauses (incluant possiblement des disjonctions de blocs), et les zones claires représentant des résolutions binaires ordonnées avec sélection entre clauses ne contenant pas le symbole $\oplus$. On rappelle qu'une $\oplus$-clause contenant au moins un symbole $\oplus$ ne se résoudra pas avec une clause sans symbole $\oplus$ — ceci modulo des détails techniques que nous traiterons plus loin, et qui nécessitent notamment de bien choisir la fonction de sélection. On voit sur le graphique ci-dessus que, pour qu'une zone hachurée puisse effectivement participer d'une preuve, elle doit avoir pour conclusion (racine) une clause qui ne contient aucun symbole $\oplus$, c'est-à-dire une disjonction de blocs (5.1) ; dans le graphique ci-dessus, ces conclusions sont matérialisées par des points noirs. Le principe de notre approximation sera d'imaginer une règle calculable qui produit au moins toutes les conclusions de zones hachurées dérivables par résolution binaire ordonnée avec sélection à partir des $\oplus$-clauses de S.

Avant de formaliser la construction, décrivons comme une telle règle calculable peut être définie. A l'intérieur d'une zone hachurée, on n'a que des $\oplus$-clauses, en particulier, aucun symbole de fonction autre que $\oplus$ n'intervient. Les déductions effectuées à l'intérieur d'une zone hachurée sont donc valides dans n'importe quel modèle du seul symbole associatif commutatif $\oplus$. En particulier, nous choisirons comme modèle $\mathbb{Z}/n\mathbb{Z}$, $\oplus$ dénotant l'addition modulo n. Nous calculons ensuite toutes les instances closes dans ce modèle des clauses données en entrée de la zone hachurée, en calculons toutes les conséquences par la règle de résolution (par exemple, les instances closes de $P(x) \Leftarrow P(x \oplus y)$ sur $\mathbb{Z}/n\mathbb{Z}$ sont $P(0) \Leftarrow P(0)$, $P(0) \Leftarrow P(1)$, $P(1) \Leftarrow P(1)$, $P(1) \Leftarrow P(0)$). Ceci termine, car il n'existe qu'un nombre fini de telles clauses closes. Une fois ceci fait, nous calculons toutes les disjonctions de blocs (5.1) dont toutes les instances closes ont été produites par les étapes précédentes. L'étape de résolution sur les instances closes produisant au moins toutes les instances closes de clauses obtenues par résolution binaire avec sélection des zones hachurées, cette procédure trouvera au moins toutes les disjonctions de blocs qui auraient été déductibles par résolution binaire avec sélection. La finitude du modèle $\mathbb{Z}/n\mathbb{Z}$ garantit que cette procédure termine.

Formalisons maintenant cette approche, à un niveau suffisant pour faire apparaître les difficultés techniques.

5.3.2.1 Choix de la fonction de sélection

Choisissons une fonction de sélection garantissant qu'une $\oplus$-clause non triviale ne se résolve pas avec une clause complexe ne mentionnant pas $\oplus$. L'idée est de sélectionner un des littéraux négatifs de la forme $-P(t)$ où t est une somme non-triviale (c'est-à-dire un terme de la forme $t_1 \oplus t_2$), s'il y en a. Le seul cas où cette fonction de sélection ne réussit pas à garantir qu'une

$\oplus$-clause non triviale ne se résoudra pas avec une clause complexe ne mentionnant pas $\oplus$ est celui d'une $\oplus$-clause de la forme

$$+P(x_1 \oplus \ldots \oplus x_n) \vee B_1(x_1) \vee \ldots \vee B_n(x_n) \vee B_{n+1}(x_{n+1}) \vee \ldots \vee B_m(x_m)$$

avec $m > n$, et où les $B_i(x_i)$ désignent des blocs négatifs : aucun littéral négatif n'est sélectionné, mais on peut résoudre sur $B_i(x_i)$, $n < i \leq m$, avec n'importe quelle clause dans laquelle aucun littéral n'est sélectionné, par exemple une clause complexe sans symbole $\oplus$. (Les littéraux de $B_i(x_i)$, $i \leq n$ ne peuvent pas être résolus : ils ne sont pas maximaux dans la clause, car sous-termes de la tête. Mais ceux avec $i > n$ sont maximaux pour l'ordre sous-terme, précisément parce que ces x_i ne sont pas libres dans la tête.) Typiquement, ce choix de fonction de sélection permet la résolution :

$$\frac{+P(x \oplus y) \vee -Q(z) \vee -Q'(z) \qquad +Q(f(x')) \vee -R(x')}{+P(x \oplus y) \vee -R(x') \vee -Q'(f(x'))}$$

Nous aimerions interdire ce genre d'étape de résolution, qui produit des clauses dans lesquelles on mélange le symbole associatif commutatif $\oplus$ et des symboles libres f.

Définissons donc la fonction de sélection comme suit :

- Si C est une $\oplus$-clause non triviale, possiblement en disjonction avec $+q$, alors :
  - si C contient un littéral négatif $-P(t)$ avec t une somme non triviale, alors $\mathsf{sel}(C)$ est un tel singleton $\{-P(t)\}$;
  - sinon, s'il existe une variable x non libre dans la tête de C mais libre dans son corps, alors $\mathsf{sel}(C)$ est l'ensemble des littéraux de la forme $-P(x)$ dans C ;
  - sinon, $\mathsf{sel}(C)$ est vide.
- Sinon, soit $\mathsf{sel}(C) = \{-q\}$, pour un littéral $-q$ quelconque de C s'il en existe.
- Sinon, soit $\mathsf{sel}(C) = \{-A\}$, où $-A$ est un littéral négatif de C, et en même temps un littéral maximal de C, s'il en existe.
- Sinon, soit $\mathsf{sel}(C) = \emptyset$.

Nous prenons comme ordre $\succ$ l'ordre sous-terme.

Nous utilisons maintenant la stratégie consistant à n'appliquer la règle de splitting sans splitting :

$$\frac{C \vee C'}{C \vee -q(*) \qquad +q(*) \vee C'}$$

que si, de plus, C' est un bloc négatif $B(x)$ non vide, $q = \ulcorner B(x) \urcorner$, et $C \vee C'$ n'est pas une $\oplus$-clause non triviale. On posera dans la suite $\mathcal{Q}_0$ l'ensemble des $\ulcorner B(x) \urcorner$, $B(x)$ variant parmi les blocs négatifs non vides, et on appellera la règle ci-dessus le *splitting restreint*.

La première restriction permet d'assurer qu'il n'y aura qu'un nombre fini de symboles spéciaux q. La seconde restriction permet de considérer que, dans les zones hachurées du graphique évoqué plus haut, aucun splitting n'est utilisé. Ceci nous permet de supposer que toute clause dans la zone hachurée est effectivement la conclusion d'un arbre de preuve. (L'utilisation de splitting sans splitting ici nous forcerait à manipuler des *graphes* de preuves, branchant tant vers le bas que vers le haut, ce qui serait d'un maniement malaisé.)

Montrons maintenant que cette stratégie de résolution fournit des clauses d'un certain format uniquement. On notera $C[\vee \pm A]$ une clause s'écrivant soit $C \vee \pm A$ soit C ;

Lemme – 5.3.1

Soit S un ensemble de clauses de Horn qui sont de l'un des types suivants :

1. *$C[\vee + q]$, où C est un bloc et $q \in \mathcal{Q}_0$;*
2. *$C \vee -q_1 \vee \ldots \vee -q_m[\vee + q]$, où C est une clause complexe où $\oplus$ n'apparaît pas, et $q_1, \ldots, q_m, q \in \mathcal{Q}_0$;*
3. *$C[\vee + q]$, où C est une $\oplus$-clause non triviale.*

Alors toute clause obtenue par résolution ordonnée avec sélection, et application immédiate du splitting restreint, est encore de l'un des types 1, 2, ou 3.

Démonstration. C'est un exercice légèrement fastidieux que de montrer que tout résolvant ainsi obtenu à partir de clauses de type 1 ou 2 est une disjonction formée d'une clause au plus de type 1 ou 2, et de blocs négatifs ; par splitting restreint, on obtient ainsi encore des clauses de type 1 ou 2.

Examinons maintenant les étapes de résolution ordonnée avec sélection :

$$\frac{C \vee -A_1 \vee \ldots \vee -A_m \quad + A'_i \vee C_i \ (1 \leq i \leq m)}{(C \vee C_1 \vee \ldots \vee C_m)\sigma}$$

où au moins une des prémisses est de type 3. Rappelons que $-A_1, \ldots, -A_m$ sont sélectionnés dans la prémisse de gauche, ou bien aucun littéral n'est sélectionné et $-A_1, \ldots, -A_m$ y sont maximaux. Rappelons aussi qu'aucun littéral n'est sélectionné dans $+A'_i \vee C_i$, et que σ unifie A_i avec A'_i pour tout i modulo AC.

- Si $C \vee -A_1 \vee \ldots \vee -A_m$ est de type 3, nous reprenons les trois cas de la définition de la fonction de sélection sur une clause de type 3 :
  - Premier cas : $m = 1$, $A_1 = P(t)$ avec t une somme non triviale. Si $+A'_1 \vee C_1$ est de type 1 ou 3, le résolvant est de type 3, ou bien une disjonction de blocs (et possiblement d'un $+q$), qui se décompose par splitting restreint en clauses de type 1. Si $+A'_1 \vee C_1$ est de type 2, C_1 ne peut pas contenir de littéral de la forme $-Q(f(x_1, \ldots, x_n))$ sinon il serait sélectionné. Mais A'_1 ne peut pas être de la forme $P(f(x_1, \ldots, x_n))$, sinon il ne s'unifierait pas avec $A_1 = P(t)$. On en déduit que $+A'_1 \vee C_1$ ne contient aucun littéral $\pm Q(f(x_1, \ldots, x_n))$, ce qui contredit qu'elle est de type 2.
  - Deuxième cas : $m \geq 1$, $A_1 = P_1(x)$, $\ldots$, $A_m = P_m(x)$, où x n'est pas libre dans C. Comme σ unifie A_i avec A'_i, il unifie tous les A'_i entre eux. Si l'un des A'_i est une somme non-triviale, alors aucune des clauses $+A'_i \vee C_i$ n'est de type 2, et par des arguments similaires à ci-dessus, le résolvant est soit de type 3, soit se décompose en clauses de type 1.

    Sinon, supposons sans perte de généralité que les clauses de type 1 parmi les $+A'_i \vee C_i$ sont celles où i varie entre 1 et k, et que celles de type 2 sont celles où i varie entre $k+1$ et m. Comme aucun littéral n'est sélectionné dans $+A'_i \vee C_i$, si $1 \leq i \leq k$, alors C_i est vide et $+A'_i$ de la forme $+P_i(x)$. (Nous choisissons la même variable x dans un souci de lisibilité du résolvant ; ceci se fait sans perte de généralité.) De même, si $k+1 \leq i \leq m$, la clause de type 2 $+A'_i \vee C_i$ est nécessairement de la forme $+P_i(f(x_1, \ldots, x_n)) \vee B_{i1}(x_1), \ldots, B_{in}(x_n)$, encore une fois car aucun littéral n'y est sélectionné.

    En particulier, si $k = m$, alors le résolvant est juste C, qui est de type 3 ou se décompose en clauses de type 1 par splitting restreint. Si $k < m$, alors σ envoie x vers $f(x_1, \ldots, x_n)$, et en se rappelant que x n'est pas libre dans C, le résolvant est :

    $$C \vee \bigvee_{i=k+1}^{m} \bigvee_{j=1}^{n} B_{ij}(x_j)$$

    qui est de type 3, ou se décompose en clauses de type 1 par splitting restreint.
  - Troisième cas : $C \vee -A_1 \vee \ldots \vee -A_m$ ne contient aucun littéral négatif où $\oplus$ apparaît, et toutes les variables libres dans son corps sont libres dans sa tête. Donc cette clause est de la forme :

    $$+P(x_1 \oplus \ldots \oplus x_n) \vee B_1(x_1) \vee \ldots \vee B_n(x_n)$$

    mais alors non seulement rien n'y est sélectionné, mais encore aucun littéral négatif n'y est maximal, donc $m = 0$, contradiction.
- Si une des clauses $+A'_i \vee C_i$ est de type 3, comme aucun littéral n'y est sélectionné, c'est qu'elle est de la forme

  $$+P(x_1 \oplus \ldots \oplus x_n) \vee B_1(x_1) \vee \ldots \vee B_n(x_n)$$

  avec $n \geq 2$. Ceci implique en particulier que $C \vee -A_1 \vee \ldots \vee -A_m$ n'est pas de type 2, sinon A_i serait nécessairement de la forme $P(x)$ avec x une variable, mais un tel littéral n'est jamais sélectionné ni maximal dans une clause de type 2. Ceci implique aussi qu'aucune des autres clauses $+A'_j \vee C_j$, $j \neq i$ n'est de type 2, car sinon elle serait de la forme

$$+P(f(x'_1, \ldots, x'_{n'})) \vee B'_1(x'_1) \vee \ldots \vee B'_{n'}(x'_{n'})$$

puisque ce sont les seules clauses de type 2 où aucun littéral n'est sélectionné ; et ceci serait contradictoire, puisque $f(x'_1, \ldots, x'_{n'})$ ne s'unifie pas avec $x_1 \oplus \ldots \oplus x_n$. Donc toutes les prémisses sont de type 1 ou 3, et la conclusion est donc de type 3 si elle contient une somme non triviale, et une disjonction de blocs (et d'un $+q$, possiblement) sinon, lequel se décompose en clauses de type 1 par splitting restreint.

□

5.3.2.2 Zones hachurées et oracles de hachures

La preuve du lemme 5.3.1 montre en particulier que, partant d'un ensemble de clauses de type 1, 2, ou 3, on peut décomposer la règle de résolution en deux cas :

- Le cas où l'une des prémisses est de type 3 ("Résolution en zone hachurée") définit deux formes de la règle de résolution :

$$\frac{C \vee -A \qquad + A' \vee C'}{(C \vee C')\sigma} \ (\oplus Res)$$

  où les conditions de la résolution binaire ordonnée avec sélection sont satisfaites, et de plus l'une des prémisses est de type 3, et l'autre de type 1 ou 3 ; et la règle

$$\frac{C \vee -P_1(x) \vee \ldots \vee -P_m(x) \qquad \begin{array}{l} +P_1(x) \\ \ldots \\ +P_k(x) \\ +P_{k+1}(f(x_1, \ldots, x_n)) \vee -B_{(k+1)1}(x_1) \vee \ldots \vee -B_{(k+1)n}(x_n) \\ \ldots \\ +P_m(f(x_1, \ldots, x_n)) \vee -B_{m1}(x_1) \vee \ldots \vee -B_{mn}(x_n) \end{array}}{C \vee \bigvee_{i=k+1}^{m} \bigvee_{j=1}^{n} B_{ij}(x_j)} \ (ExtRes)$$

  où x n'est pas libre dans C, et C est de la forme

$$+P(y_1 \oplus \ldots \oplus y_k) \vee B_1(y_1) \vee \ldots \vee B_k(y_k) \vee B_{k+1}(y_{k+1}) \vee \ldots \vee B_\ell(y_\ell)$$

  avec $\ell \geq k \geq 2$. (La règle ($ExtRes$) décrit ce qui se passe dans la preuve du lemme 5.3.1, dans le cas où la prémisse de gauche $C \vee -A_1 \vee \ldots \vee -A_m$ est de type 3, deuxième cas. (Il s'agit de l'unique règle non décrite dans notre argumentation informelle plus haut.)
- Le cas où aucune des prémisses n'est de type 3 ("Résolution hors zone hachurée"), qui est décrit par la règle de la figure 5.6.

Il est à noter d'autre part que, comme indiqué plus haut, on ne sort de zone hachurée qu'en fabriquant des disjonctions de blocs, modulo le correctif que cette disjonction de blocs est elle-même possiblement en disjonction avec un $+q$. La démonstration du lemme 5.3.1 le montre : la seule façon d'obtenir une clause qui n'est pas de type 3 par résolution ordonnée avec sélection est en résolvant des clauses dont au moins une est de type 3, et le résolvant est toujours soit de type 3 (auquel cas l'on reste en zone hachurée) ou une disjonction de blocs, possiblement en disjonction avec un $+q$.

Abstrayons maintenant tout ce qui peut se passer en zone hachurée par une unique règle : partant d'un ensemble courant S de clauses de type 1, 2, ou 3, nous cherchons à deviner les clauses qui seront les conclusions possibles de zones hachurées partant de S. Par les remarques

$$\frac{C' \vee -P(t') \quad C \vee +P(t)}{(C \vee C')\sigma}$$

où :

(i) t et t' ne sont pas de la forme $x_1 \oplus \ldots \oplus x_n$;

(ii) $\sigma \in \mathsf{csu}(t \doteq t')$;

(iii) $\mathsf{sel}(C \vee +P(t)) = \emptyset$ et $P(t)$ est maximal dans $C \vee +P(t)$;

(iv) $-P(t') \in \mathsf{sel}(C' \vee -P(t'))$, ou bien $\mathsf{sel}(C' \vee -P(t')) = \emptyset$ et $P(t')$ est maximal dans $C' \vee -P(t')$.

Fig. 5.6: Résolution hors zone hachurée

ci-dessus, ces clauses sont d'une part des disjonctions de blocs, possiblement en disjonction avec un littéral $+q$; d'autre part elles sont déductibles par les seules règles $(\oplus Res)$ et $(ExtRes)$.

On en déduit la définition et la proposition suivantes.

Définition – 5.3.1

Un candidat hachuré *est une clause de Horn de la forme* $B_1(x_1) \vee \ldots \vee B_n(x_n)[\vee + q]$*, où les variables* x_i *sont deux à deux distinctes et les* $B_i(x_i)$ *sont des blocs non vides.*

Un oracle de hachures *est une fonction* $\mathcal{O}$ *associant à tout ensemble de clauses de type 1, 2, ou 3, un ensemble de candidats hachurés contenant au moins tous ceux qui sont déductibles par les règles* $(\oplus Res)$ *et* $(ExtRes)$ *de* S*.*

Proposition – 5.3.1

Soit $\mathcal{O}$ *un oracle de hachures quelconque. Alors l'ensemble des règles :*

1. *de résolution hors zone hachurée (figure 5.6) ;*
2. *d'application de l'oracle : à* S *ajouter* $\mathcal{O}(S)$ *;*

est complète pour tout ensemble de clauses de type 1, 2, ou 3 : pour tout ensemble S_0 *de clauses de type 1, 2, ou 3, si* S_0 *est insatisfiable modulo* AC*, alors la clause vide est déductible à partir de* S_0 *par les règles ci-dessus. De plus, on peut effacer à volonté les clauses subsumées linéairement, les tautologies, et appliquer à volonté le splitting restreint.*

On notera bien que nous n'avons pas demandé que $\mathcal{O}$ soit *correct*, c'est-à-dire qu'il ne déduise que des candidats hachurés réellement déductibles de S par $(\oplus Res)$ et $(ExtRes)$. Si $\mathcal{O}$ était correct, il déciderait toute la classe des automates d'arbres bidirectionnels modulo AC qui, comme on l'a vu, est très complexe [GLV02]. En fait, les ensembles de clauses de type 1, 2, ou 3 incluent même la classe des automates d'arbres bidirectionnels *alternants* modulo AC, qui est indécidable [GLV02]. Il n'y a donc tout simplement aucun oracle de hachures à la fois correct et calculable.

Toutefois, tout oracle de hachures donne une procédure d'approximation de la satisfiabilité d'ensemble de clauses de type 1, 2, ou 3 : par complétude (proposition 5.3.1), si la procédure de la proposition 5.3.1 ne déduit pas la clause vide, alors l'ensemble S_0 de clauses de départ est satisfiable. Nous verrons au chapitre 6 que la satisfiabilité de S_0, lorsque S_0 code un protocole cryptographique, est équivalente à l'inexistence d'une attaque. Autrement dit, si la procédure de la proposition 5.3.1 ne déduit pas la clause vide, alors le protocole cryptographique ainsi codé n'a aucune attaque : il est sûr.

5.3.2.3 Un raffinement des zones hachurées

Trouver un oracle de hachures tel que défini en définition 5.3.1 est délicat. Le problème vient essentiellement de la règle $(ExtRes)$, qui est compliquée.

La conclusion

$$C \vee \bigvee_{i=k+1}^{m} \bigvee_{j=1}^{n} B_{ij}(x_j)$$

de cette règle est décomposable par splitting sans splitting, puisqu'aucune des variables $x_1, \ldots, x_n$ n'est libre dans C par construction. (Attention, pas par splitting restreint, puisque C est une $\oplus$-clause non triviale.) Comme on peut appliquer le splitting sans splitting à volonté sans que la complétude en soit affectée, on peut remplacer la règle ($ExtRes$) par la règle :

$$\frac{C \vee -P_1(x) \vee \ldots \vee -P_m(x) \qquad \begin{array}{l} +P_1(x) \\ \ldots \\ +P_k(x) \\ +P_{k+1}(f(x_1,\ldots,x_n)) \vee -B_{(k+1)1}(x_1) \vee \ldots \vee -B_{(k+1)n}(x_n) \\ \ldots \\ +P_m(f(x_1,\ldots,x_n)) \vee -B_{m1}(x_1) \vee \ldots \vee -B_{mn}(x_n) \end{array}}{C \vee -q_{j_1} \vee \ldots \vee -q_{j_p} \qquad +q_{j_1} \vee \bigvee_{i=k+1}^{m} B_{ij_1}(x_{j_1}) \quad \ldots \quad +q_{j_p} \vee \bigvee_{i=k+1}^{m} B_{ij_p}(x_{j_p})} \; (ExtRes')$$

où x n'est pas libre dans C, C est de la forme

$$+P(y_1 \oplus \ldots \oplus y_k) \vee B_1(y_1) \vee \ldots \vee B_k(y_k) \vee B_{k+1}(y_{k+1}) \vee \ldots \vee B_\ell(y_\ell)$$

avec $\ell \geq k \geq 2$, et $\{j_1, \ldots, j_p\}$ est l'ensemble des indices j, $1 \leq j \leq m$, tels que le bloc négatif $\bigvee_{i=k+1}^{m} B_{ij}(x_j)$ est non vide, et $q_j = \ulcorner \bigvee_{i=k+1}^{m} B_{ij}(x_j) \urcorner$ pour tout $j \in \{j_1, \ldots, j_p\}$.

La prémisse de gauche $C \vee -P_1(x) \vee \ldots \vee -P_m(x)$ est elle aussi splittable par définition. (Attention, encore une fois pas par splitting restreint, qui interdit de splitter de telles clauses, puisque C est une $\oplus$-clause non triviale.) On peut alors simuler l'effet de la règle ($ExtRes'$) en combinant les deux dérivations suivantes, où $q = \ulcorner -P_1(x) \vee \ldots \vee -P_m(x) \urcorner$. En premier, la dérivation triviale de $C \vee -q$ par splitting sans splitting ; en second, la dérivation suivante, obtenue à partir de l'autre conclusion $+q \vee -P_1(x) \vee \ldots \vee -P_m(x)$ de la règle de splitting sans splitting, en utilisant répétitivement la règle de résolution binaire ordonnée avec sélection, suivie de splitting sans splitting :

$$\frac{+q \vee -P_1(x) \vee \ldots \vee -P_m(x) \qquad \begin{array}{l} +P_1(x) \\ \ldots \\ +P_k(x) \\ +P_{k+1}(f(x_1,\ldots,x_n)) \vee -B_{(k+1)1}(x_1) \vee \ldots \vee -B_{(k+1)n}(x_n) \\ \ldots \\ +P_m(f(x_1,\ldots,x_n)) \vee -B_{m1}(x_1) \vee \ldots \vee -B_{mn}(x_n) \end{array}}{+q \vee -q_{j_1} \vee \ldots \vee -q_{j_p} \qquad +q_{j_1} \vee \bigvee_{i=k+1}^{m} B_{ij_1}(x_{j_1}) \quad \ldots \quad +q_{j_p} \vee \bigvee_{i=k+1}^{m} B_{ij_p}(x_{j_p})} \tag{5.2}$$

Il ne reste plus qu'à résoudre la conclusion de gauche avec $C \vee -q$ sur q pour réobtenir les conclusions de la règle ($ExtRes'$).

L'intérêt de cette approche est que l'on peut maintenant se passer complètement de la règle prohibitive ($ExtRes$). En revanche, pour que ceci soit légal, nous devons maintenant d'une part autoriser l'application du splitting sans splitting sur les clauses de type 3, et d'autre part autoriser les étapes de résolution de la forme

$$\frac{C \vee -q \qquad +q \vee -q_{j_1} \vee \ldots \vee -qj_p}{C \vee -q_{j_1} \vee \ldots \vee -qj_p}$$

Ces étapes étaient jusque-là prohibées sauf si $p = 0$, au vu des conditions de sélection.

En résumé, on a donc :

Lemme – 5.3.2

Considérons la règle :

$$\frac{C \vee -q \qquad +q \vee -q_1 \vee \ldots \vee -q_m}{C \vee -q_1 \vee \ldots \vee -q_m} \; (qRes)$$

où C est de type 3.

La règle de splitting de blocs négatifs *est la restriction suivante de la règle de splitting sans splitting :*

$$\frac{C \vee B(x)}{C \vee -q(*) \qquad + q(*) \vee B(x)}$$

où $B(x)$ est un bloc négatif non vide, x est non libre dans C, et $q = \ulcorner B(x) \urcorner$.

Alors les règles $(\oplus Res)$, $(qRes)$, de résolution hors zone hachurée (figure 5.6), et de splitting de blocs négatifs, avec effacement à volonté des tautologies et des clauses subsumées linéairement, est complète pour les ensembles de clauses de type 1, 2, ou 3 : pour tout ensemble S_0 de clauses de type 1, 2, ou 3, si S_0 est insatisfiable modulo AC*, alors on peut en déduire la clause vide par application des règles ci-dessus.*

Démonstration. Il suffit d'appliquer l'argumentation ci-dessus à une dérivation π fixée de la clause vide à partir de S_0 par règles $(\oplus Res)$, $(ExtRes')$ ou de résolution hors zone hachurée avec splitting restreint appliqué au plus tôt, et élimination de clauses subsumées linéairement et de tautologies à volonté. Formellement, on montre par récurrence sur la taille de π que si π dérive une clause C, alors on peut dériver C par les règles $(\oplus Res)$, $(qRes)$ et de résolution hors zone hachurée, avec application de la règle de splitting de blocs négatifs au plus tôt.

Dans le cas où π se termine par la règle $(ExtRes')$, c'est l'argumentation ci-dessus, qui utilise les règles $(qRes)$ et de splitting de blocs négatifs. Le cas des autres règles est évident. □

Ce lemme ne dit pas si l'on peut utiliser le splitting de blocs négatifs au plus tôt ou non. Nous corrigeons cette imperfection dans la proposition suivante.

Proposition – 5.3.2

On dit qu'une clause est de type 3' *si et seulement si elle est de Horn et de la forme $C \vee -q_1 \vee \ldots \vee -q_m[\vee + q]$, où C est une $\oplus$-clause non triviale.*

Considérons la règle :

$$\frac{C \vee -q_1 \vee \ldots \vee -q_m \vee -A \qquad + A' \vee C' \vee -q'_1 \vee \ldots \vee -q'_{m'}}{(C \vee C')\sigma \vee -q_1 \vee \ldots \vee -q_m \vee -q'_1 \vee \ldots \vee -q'_{m'}} \, (\oplus Res')$$

où :

(i) $\sigma \in \mathsf{csu}(A \doteq A')$

(ii) $+A'$ *est maximal dans* $+A' \vee C'$ (pas *dans* $+A' \vee C' \vee -q'_1 \vee \ldots \vee -q'_{m'}$*) et aucun littéral n'est sélectionné dans* $+A' \vee C'$ *;*

(iii) $-A$ *est sélectionné dans* $C \vee -A$*,* (pas *dans* $C \vee -q_1 \vee \ldots \vee -q_m \vee -A$*), ou bien aucun littéral n'est sélectionné dans* $C \vee -A$ *et* $-A$ *est maximal dans* $C \vee -A$ *;*

et de plus l'une des prémisses est de type 3', et l'autre de type 1 ou 3'. Les deux règles $(\oplus Res')$ et $(qRes)$ forment la règle de résolution en zone hachurée modifiée.

Alors les règles $(\oplus Res')$, $(qRes)$, de résolution hors zone hachurée (figure 5.6), avec splitting de blocs négatifs au plus tôt, et effacement à volonté des tautologies et des clauses subsumées linéairement, est complète pour les ensembles de clauses de type 1, 2, ou 3 : pour tout ensemble S_0 de clauses de type 1, 2, ou 3, si S_0 est insatisfiable modulo AC*, alors on peut en déduire la clause vide par application des règles ci-dessus.*

Démonstration. Toute clause s'écrit de façon unique $C \vee B_1(x_1) \vee \ldots \vee B_n(x_n)$, où $x_1, \ldots, x_n$ sont deux à deux distinctes et non libres dans C, où $B_1(x_1), \ldots, B_n(x_n)$ sont des blocs négatifs non vides, et où n est maximal avec ces propriétés. Par splitting de blocs négatifs, on peut en dériver $C \vee -q_1 \vee \ldots \vee -q_n$ et $+q_j \vee B_j(x_j)$ pour tout j, $1 \leq j \leq n$, où $q_j = \ulcorner B_j(x_j) \urcorner$. Appelons $C \vee -q_1 \vee \ldots \vee -q_n$ la *splittée* de $C \vee B_1(x_1) \vee \ldots \vee B_n(x_n)$, et les clauses $+q_j \vee B_j(x_j)$ les *définitions* des q_j ; les q_j eux-mêmes sont les *symboles de splitting.*

Nous devons montrer que pour chacune des deux règles $(\oplus Res)$, $(qRes)$, on peut déduire les splittées de la conclusion à partir des splittées des prémisses, ainsi que les définitions des symboles de splitting associés. (Il n'y a rien à prouver pour la règle de résolution hors zone hachurée, puisque l'on peut déjà supposer sans perte de généralité que la règle de splitting de blocs négatifs, qui s'y réduit au splitting restreint, est appliquée au plus tôt.)

- ($\oplus Res$). Nous avons deux cas. Dans le premier cas, les prémisses sont de la forme $C \vee B_1(x_1) \vee \ldots \vee B_n(x_n) \vee -A$ et $+A' \vee C' \vee B'_1(x'_1) \vee \ldots \vee B'_n(x'_{n'})$; dans le deuxième cas, $-A$ fait partie de l'un des blocs B_i.
  - Premier cas : par hypothèse de récurrence on a dérivé $C \vee -q_1 \vee \ldots \vee -q_n \vee -A$ et $+A' \vee C' \vee -q'_1 \vee \ldots \vee -q'_{n'}$, où $q_i = \ulcorner B_i(x_i) \urcorner$ et $q'_i = \ulcorner B'_i(x'_i) \urcorner$ pour tout i ; on en déduit $(C \vee C')\sigma \vee -q_1 \vee \ldots \vee -q_n \vee -q'_1 \vee \ldots \vee -q'_{n'}$ par ($\oplus Res'$), et cette dernière clause est ou se décompose par splitting de blocs négatifs en la splittée de la conclusion $C \vee B_1(x_1) \vee \ldots \vee B_n(x_n) \vee C' \vee B'_1(x'_1) \vee \ldots \vee B'_n(x'_{n'})$.
  - Second cas : notons que comme $-A$ est sélectionné dans la prémisse de gauche, ou bien qu'il est maximal et rien n'est sélectionné, alors la prémisse de gauche est nécessairement de type 1, disons $B(x) \vee -P(x)[\vee + q]$, où $P(x)$ désigne A et où $B(x)$ est un bloc. Noter que sa splittée est elle-même. La prémisse de droite est de la forme $+P(t) \vee C' \vee B'_1(x'_1) \vee \ldots \vee B'_n(x'_{n'})$, de splittée $+P(t) \vee C' \vee -q'_1 \vee \ldots \vee -q'_{n'}$. Par la règle ($\oplus Res'$), on en déduit $B(t)[\vee + q] \vee C' \vee -q'_1 \vee \ldots \vee -q'_{n'}$, dont la splittée est clairement aussi celle du résolvant $B(t)[\vee + q] \vee B'_1(x'_1) \vee \ldots \vee B'_n(x'_{n'})$.
- ($qRes$). Les prémisses sont de la forme $C \vee B_1(x_1) \vee \ldots \vee B_n(x_n) \vee -q$ et $+q \vee -q'_1 \vee \ldots \vee -q'_{n'}$, de résolvant $C \vee B_1(x_1) \vee \ldots \vee B_n(x_n) \vee -q'_1 \vee \ldots \vee -q'_{n'}$. Les splittées des prémisses sont $C \vee -q_1 \vee \ldots \vee -q_n \vee -q$ ($q_i = \ulcorner B_i(x_i) \urcorner$ pour tout i) et $+q \vee -q'_1 \vee \ldots \vee -q'_{n'}$. Par ($qRes$), on en déduit $C \vee -q_1 \vee \ldots \vee -q_n \vee -q'_1 \vee \ldots \vee -q'_{n'}$, qui est la splittée de $C \vee B_1(x_1) \vee \ldots \vee B_n(x_n) \vee -q'_1 \vee \ldots \vee -q'_{n'}$.

□

On en déduit comme plus haut un critère de complétude à base d'oracles de hachures modifiés. À noter que l'on sort maintenant de zone hachurée non plus en fabriquant une clause $B_1(x_1) \vee \ldots \vee B_n(x_n)[\vee + q]$, mais en fabriquant sa splittée $B_1(x_1) \vee -q_2 \vee \ldots \vee -q_n[\vee + q]$ ou $+q \vee -q_1 \ldots \vee -q_n$, plus des définitions de symboles de splitting, incluant les symboles $q_2, \ldots, q_n$ mais pas forcément limitées à ces derniers.

Définition – 5.3.2

Un candidat modifié *est soit une clause $B(x) \vee -q_2 \vee \ldots \vee -q_n[\vee + q]$, où $B(x)$ est un bloc, soit une clause $+q \vee -q_1 \ldots \vee -q_n$, soit une définition $+\ulcorner B'(x) \urcorner \vee B'(x)$.*

Un tel candidat modifié c est accessible *à partir de l'ensemble S de clauses de type 1, 2, ou 3' si et seulement s'il existe une dérivation de c par les règles ($\oplus Res'$) et ($qRes$) avec splitting de blocs négatifs au plus tôt et élimination à volonté des clauses subsumées linéairement et des tautologies, à partir de S.*

Un oracle modifié *est une fonction $\mathcal{O}$ associant à tout ensemble de clauses de type 1, 2, ou 3', un ensemble de candidats modifiés contenant au moins tous ceux qui sont accessibles à partir de S.*

Proposition – 5.3.3

Soit $\mathcal{O}$ un oracle modifié quelconque. Alors l'ensemble des règles :

1. *de résolution hors zone hachurée (figure 5.6) ;*
2. *d'application de l'oracle modifié : à S ajouter $\mathcal{O}(S)$;*

est complète pour tout ensemble de clauses de type 1, 2, ou 3 : pour tout ensemble S_0 de clauses de type 1, 2, ou 3, si S_0 est insatisfiable modulo AC, *alors la clause vide est déductible à partir de S_0 par les règles ci-dessus. De plus, on peut effacer à volonté les clauses subsumées linéairement, les tautologies, et appliquer à volonté le splitting de blocs négatifs.*

Il ne reste plus qu'à trouver un tel oracle modifié.

5.3.2.4 Un oracle modifié simple

L'intérêt majeur des oracles modifiés est que si $\mathcal{O}$ est un oracle modifié, alors la fonction qui à S associe $\mathcal{O}(S_{|1,3'})$, où $S_{|1,3'}$ est le sous-ensemble de S qui ne contient que les clauses de type 1 ou 3', est encore un oracle modifié : c'est parce que les règles $(\oplus Res')$ et $(qRes)$ ne s'appliquent qu'à des clauses de type 1 ou 3', jamais 2. Nous supposerons donc sans perte de généralité que S est un ensemble de clauses de type 1 ou 3'. En particulier, le seul symbole de fonction apparaissant dans S est $\oplus$. (Il n'y a même pas de constante.)

Soit E un ensemble fini quelconque donné, et $+$ une loi associative et commutative sur E : autrement dit, nous supposons que $(E, +)$ est une semi-groupe commutatif fini. En pratique, et dans l'implémentation, nous utiliserons $E = \mathbb{Z}/n\mathbb{Z}$ avec n un entier supérieur ou égal à 2. Nous laisserons souvent l'opération $+$ sous-entendue.

Appelons *E-instance* d'une clause de type 1 ou 3' la clause obtenue en remplaçant chaque variable libre de la clause par un élément de E, en remplaçant $\oplus$ par $+$ et en opérant les simplifications dictées par la définition de $+$.

Formellement, soit ρ un *environnement*, c'est-à-dire une fonction qui à chaque variable associe un élément de E. On pose :

$$[\![x]\!]\,\rho = \rho(x) \qquad [\![u \oplus v]\!]\,\rho = [\![u]\!]\,\rho + [\![v]\!]\,\rho$$

et pour tout atome $P(t)$, $[\![P(t)]\!]\,\rho = P([\![t]\!]\,\rho)$; un tel atome est appelé un *E-atome*. Pour tout littéral $\pm P(t)$, $[\![\pm P(t)]\!]\,\rho$ est défini par $\pm P([\![t]\!]\,\rho)$; un tel littéral est un *E-littéral*. Enfin, pour toute clause $C = \pm_1 A_1 \vee \ldots \vee \pm_n A_n$, $[\![C]\!]\,\rho = \pm_1 [\![A_1]\!]\,\rho \vee \ldots \vee \pm_n [\![A_n]\!]\,\rho$, une *$E$-clause*. Les *$E$-instances* de la clause C sont les E-clauses $[\![C]\!]\,\rho$.

Notre stratégie consiste à simuler les règles $(\oplus Res')$ et $(qRes)$ au niveau des E-instances.

On a donc pour commencer une première règle d'*instantiation* : à partir de toute clause C de type 1 ou 3', ajouter à l'ensemble courant S toutes les E-instances de C :

$$\frac{C}{[\![C]\!]\,\rho}\,(Inst)$$

où C est une clause de type 1 ou 3', et $[\![C]\!]\,\rho$ les E-instances de C.

Ensuite, nous simulons simplement les règles $(\oplus Res')$ et $(qRes)$ au niveau des E-instances par la règle de résolution binaire propositionnelle :

Lemme – 5.3.3

Étant donné un ensemble ES de E-clauses, soit $Res_0(ES)$ l'ensemble de toutes les clauses déductibles de ES par résolution binaire propositionnelle.

Soit S un ensemble de clauses de type 1 ou 3', $E(S)$ l'ensemble de ses E-instances, et $Res_\oplus(S)$ l'ensemble des clauses déductibles à partir de S par $(\oplus Res')$ ou $(qRes)$.

Alors $E(Res_\oplus(S)) \subseteq Res_0(E(S))$.

Démonstration. Par récurrence sur le nombre d'applications des règles $(\oplus Res')$ et $(qRes)$, on montre que si C est déductible à partir de S, alors $[\![C]\!]\,\rho$ est déductible à partir de $E(S)$ par résolution propositionnelle, pour tout environnement ρ. Montrons en général que c'est encore le cas si C est déductible à partir de C par résolution. Le cas où C est dans S est évident. Sinon, on peut supposer que C est de la forme $(C_1 \vee C_2)\sigma$, où par hypothèse de récurrence, $[\![C_1 \vee +A_1 \vee \ldots \vee +A_n]\!]\,\rho'$ et $[\![C_2 \vee -A]\!]\,\rho'$ sont déductible par résolution binaire propositionnelle de $E(S)$, pour tout environnement ρ.

Choisissons un environnement quelconque ρ. Posons ρ' l'environnement qui à tout x associe $[\![\sigma(x)]\!]\,\rho$. On remarque que $[\![(C_1 \vee C_2)\sigma]\!]\,\rho = [\![C_1 \vee C_2]\!]\,\rho'$. D'autre part, $[\![C_1 \vee +A_1 \vee \ldots \vee +A_n]\!]\,\rho' = [\![C_1]\!]\,\rho' \vee +[\![A_1]\!]\,\rho' \vee \ldots \vee +[\![A_n]\!]\,\rho' = [\![C_1]\!]\,\rho' \vee +[\![A]\!]\,\rho'$, puisque pour tout i, $1 \leq i \leq n$, $[\![A_i]\!]\,\rho' = [\![A_i]\!]\,(\lambda x \cdot [\![\sigma(x)]\!]\,\rho) = [\![A_i\sigma]\!]\,\rho = [\![A\sigma]\!]\,\rho = [\![A]\!]\,\rho'$. On en déduit que $[\![(C_1 \vee C_2)\sigma]\!]\,\rho = [\![C_1]\!]\,\rho' \vee [\![C_2]\!]\,\rho'$ est déductible par résolution binaire propositionnelle à partir de $[\![C_1 \vee +A_1 \vee \ldots \vee +A_n]\!]\,\rho' = [\![C_1]\!]\,\rho' \vee + [\![A]\!]\,\rho'$ et de $[\![C_2 \vee -A]\!]\,\rho' = [\![C_2]\!]\,\rho' \vee - [\![A]\!]\,\rho'$. $\square$

Simuler le splitting de blocs négatifs est plus complexe algorithmiquement. Remarquons que l'ensemble des E-instances de $C \vee -P_1(x) \vee \ldots \vee -P_n(x)$, où x n'est pas libre dans C, s'exprime exactement comme l'ensemble des clauses $C_0 \vee -P_1(k) \vee \ldots \vee -P_n(k)$, où C_0 parcourt les E-instances de C et k parcourt E.

Considérons donc la règle suivante, applicable à un ensemble contenant aussi bien des E-clauses que des clauses de type 1, 2, ou 3' :

$$\frac{\overbrace{C_0 \vee -P_1(k) \vee \ldots \vee -P_n(k)}^{k \in E}}{C_0 \vee -q \qquad +q \vee -P_1(x) \vee \ldots \vee -P_n(x)} \,(ASplit)$$

où $q = \ulcorner -P_1(x) \vee \ldots \vee -P_n(x) \urcorner$, $n \geq 1$, et $C_0 \vee -P_1(k) \vee \ldots \vee -P_n(k)$ est une E-clause non triviale. Cette règle se lit comme suit : si dans l'ensemble S de E-clauses et de clauses de type 1, 2, ou 3', on trouve toutes les E-clauses $C_0 \vee -P_1(k) \vee \ldots \vee -P_n(k)$ lorsque k parcourt E, C_0 étant fixé, pour un certain ensemble de prédicats $P_1, \ldots, P_n$, alors on ajoute à S la E-clause $C_0 \vee -q$, ainsi que la définition du symbole de splitting q.

De façon similaire, simulons la sortie de zone hachurée, c'est-à-dire la détection de clauses de la forme $B(x) \vee -q_2 \vee \ldots \vee -q_n[\vee + q]$ ou $+q \vee q_1 \vee \ldots \vee q_n$, par la règle :

$$\frac{\overbrace{B(k) \vee -q_2 \vee \ldots \vee -q_n[\vee + q]}^{k \in E}}{B(x) \vee -q_2 \vee \ldots \vee -q_n[\vee + q]} \,(Exit)$$

Notons que les clauses de la forme $+q \vee q_1 \vee \ldots \vee q_n$ sortent de manière triviale de la zone hachurée puisque ce sont des E-instances triviales.

Lemme – 5.3.4

Soit S un ensemble de E-clauses et de clauses de type 1, 2, ou 3'. Soit $S_{|E}$ le sous-ensemble des E-clauses, et $S_{|1,2,3'}$ le sous-ensemble des clauses de type 1, 2, ou 3'.

Soit $Ded_0(S)$ l'ensemble de toutes les clauses et E-clauses déductibles par résolution binaire propositionnelle entre E-clauses, et par les règles $(ASplit)$ et $(Exit)$.

Soit $Ded_\oplus(S) = Res_\oplus(S_{|1,2,3'})$.

Supposons que $S_{|E}$ contienne $E(S_{|1,2,3'})$. Alors $E(Ded_\oplus(S)) \subseteq Ded_0(S)$. De plus, pour tout candidat modifié C accessible à partir de $S_{|1,2,3'}$, C est dans $Ded_0(S)$.

Démonstration. $E(Ded_\oplus(S)) \subseteq Ded_0(S)$ se montre comme pour le lemme 5.3.3, mais nous devons considérer en plus la règle de splitting de blocs négatifs, pour laquelle nous utilisons la règle $(ASplit)$.

Pour la deuxième partie, soit C un candidat modifié quelconque, accessible à partir de $S_{|1,2,3'}$. Par la première partie, toutes les E-instances de C sont dans $Ded_0(S)$. Si C est une définition $+q \vee B(x)$ de symbole de splitting $q = \ulcorner B(x) \urcorner$, il est facile de voir par récurrence sur le nombre de clauses que $C \in Ded_0(S)$. Sinon, par la première partie du lemme, toutes ses E-instances sont dans $Ded_0(S)$, donc C est déductible grâce à la règle $(Exit)$. □

On en déduit :

Proposition – 5.3.4

La fonction prenant en entrée un ensemble S de clauses de type 1, 2, ou 3', et retournant l'ensemble des candidats modifiés dans $Ded_0(S \cup E(S))$ est un oracle modifié calculable.

Démonstration. C'est un oracle modifié par le lemme 5.3.4. Il est calculable car on peut calculer l'ensemble des conséquences par résolution binaire propositionnelle, par $(ASplit)$ et $(Exit)$ à partir de tout ensemble d'E-clauses et de clauses de type 1, 2, ou 3', et que de plus l'ensemble des clauses ainsi obtenues est fini. □

Par la proposition 5.3.3, on obtient ainsi un algorithme complet, mais pas nécessairement correct, de satisfiabilité d'automates d'arbres bidirectionnels modulo AC (même alternants). Rappelons que c'est la complétude dont nous aurons besoin dans les applications, pas de la correction : l'absence d'une déduction de la clause vide nous permettra notamment d'obtenir une preuve de sécurité d'un protocole cryptographique ; voir le chapitre 6.

Nous verrons dans la suite comment implémenter efficacement les règles $(ASplit)$ et $(Exit)$, et quelles modifications ceci demande à notre prouveur modulaire par résolution.

5.3.3 Modifications du prouveur

Avant de décrire les modifications que nous allons apporter au prouveur, nous allons montrer comment nous pouvons implémenter les règles $(ASplit)$ et $(Exit)$ qui apparaîtront naturellement dans les nouvelles règles du prouveur.

5.3.3.1 Implémentation des règles $(ASplit)$ et $(Exit)$

L'idée de base est de compter les occurrences des $\mathbb{Z}/n\mathbb{Z}$-clauses que l'on examine au fur et à mesure du processus de résolution. Dès que n occurrences d'une même $\mathbb{Z}/n\mathbb{Z}$-clause ont été comptées, alors on peut générer la clause par l'une des règles appropriées $(ASplit)$ et $(Exit)$.

Rappellons que les règles $(ASplit)$ et $(Exit)$ sont :

$$\frac{\overbrace{C_0 \vee -P_1(k) \vee \ldots \vee -P_n(k)}^{k\in E}}{C_0 \vee -q \qquad +q \vee -P_1(x) \vee \ldots \vee -P_m(x)}\,(ASplit) \qquad \frac{\overbrace{B(k) \vee -q_2 \vee \ldots \vee -q_m[\vee +q]}^{k\in E}}{B(x) \vee -q_2 \vee \ldots \vee -q_m[\vee +q]}\,(Exit)$$

En premier, nous allons prendre le cas de la règle $(Exit)$ qui est la plus simple à traiter, et qui nous permettra d'introduire plus facilement le cas de la règle $(ASplit)$.

Soit $C = B(k) \vee -q_2 \vee \ldots \vee -q_{m'}[\vee + q]$, $k \in \mathbb{Z}/n\mathbb{Z}$, une clause de l'ensemble courant S. On crée alors un *EX-squelette* de la clause, c'est-à-dire un ensemble de la forme

$$\{\ \{\pm_1 P_1, \ldots, \pm_m P_m\},\ \{-q_2, \ldots, -q_{m'}[+q]\}\ \}$$

où $\pm_1 P_1(k) \vee \ldots \vee \pm_m P_m(k) = B(k)$. Notons $Squel_{Exit}(C)$ le EX-squelette de C ainsi formé. L'idée va est de compter le nombre de $Squel_{Exit}(C)$. Pour cela, supposons que l'on dispose d'un ensemble, noté Tab_{Exit} dont les éléments sont des couples formés d'un EX-squelette et d'un entier naturel représentant le nombre de fois où ce EX-squelette a été rencontré.

Ensuite, deux cas se présentent :

1. Si aucune autre $\mathbb{Z}/n\mathbb{Z}$-instance de C n'a été rencontrée, c'est-à-dire si $Squel_{Exit}(C)$ n'est la première composante d'aucun élément de Tab_{Exit}, alors on ajoute l'élément $(\ \{\ \{\pm_1 P_1, \ldots, \pm_m P_m\},\ \{-q_2, \ldots, -q_{m'}[, +q]\}\ \},\ 1)$ à l'ensemble Tab_{Exit} ;
2. Si par contre, il existe $(\ \{\ \{\pm_1 P_1, \ldots, \pm_m P_m\},\ \{-q_2, \ldots, -q_{m'}[, +q]\}\ \},\ p)$ appartenant à Tab_{Exit}, on le remplace par $(\ \{\ \{\pm_1 P_1, \ldots, \pm_m P_m\},\ \{-q_2, \ldots, -q_{m'}[, +q]\}\ \},\ p+1)$, et
   - si $p + 1 = n$, on génère la clause $B(x) \vee -q_2 \vee \ldots \vee -q_{m'}[\vee + q]$ dans l'ensemble S ;
   - sinon, on ne fait rien.

Le cas de la règle $(ASplit)$ est un peu plus complexe, dans la mesure où il faut repérer dans la $\mathbb{Z}/n\mathbb{Z}$-clause C en prémisse tous les ensembles de prédicats pour lesquels C est de la forme $C_0 \vee -P_1(k) \vee \ldots \vee -P_m(k)$, et ce pour tout $k \in \mathbb{Z}/n\mathbb{Z}$, mais l'idée sera cependant la même que précédemment. La différence sera que pour une telle $\mathbb{Z}/n\mathbb{Z}$-clause, on devra créer un ensemble de squelettes, et non pas un unique squelette. Décrivons maintenant plus précisément l'idée que nous venons de donner.

Pour toute $\mathbb{Z}/n\mathbb{Z}$-clause C, et pour tout $k \in \mathbb{Z}/n\mathbb{Z}$ tels qu'il existe un $-P(k)$ dans C, écrivons C sous la forme $C_0' \vee -P_1(k) \vee \ldots \vee -P_m(k)$ où m est maximal, c'est-à-dire qu'il n'y a pas de prédicat de la forme $-P'(k)$ dans C_0'. Pour chaque sous-ensemble non vide $\{P_{i_1}, \ldots, P_{i_{m'}}\}$ de $\{P_1, \ldots, P_m\}$, on va créé un *AS-squelette* où apparaîtront les $P_{i_1}, \ldots, P_{i_{m'}}$ et le "reste" de la clause C. Explicitons tout d'abord ce que veut dire ce "reste". Notons $\{P_{j_1}, \ldots, P_{j_l}\}$ l'ensemble des symboles de prédicats égal à $\{P_{i_1}, \ldots, P_{i_m}\}$ moins $\{P_{i_1}, \ldots, P_{i_{m'}}\}$, et notons $C_{m'}$ la clause $P_{j_1}(k) \vee \ldots \vee P_{j_l}(k)$. Ainsi, pour chaque sous-ensemble non vide $\{P_{i_1}, \ldots, P_{i_{m'}}\}$, on crée le AS-squelette

$$\{\ \{P_{i_1}, \ldots, P_{i_{m'}}\},\ \{C'_0 \vee C_{m'}\}\ \}$$

On imagine maintenant facilement la suite du processus, qui va être essentiellement la même chose que pour la règle ($Exit$). Nous avons décrit la façon dont nous créons les AS-squelettes pour toute $\mathbb{Z}/n\mathbb{Z}$-clause C et pour tout $k \in \mathbb{Z}/n\mathbb{Z}$ tels qu'il existe un $-P(k)$ dans C. Notons $Squel_{ASplit}(C)$ l'ensemble de ces AS-squelettes, et notons Tab_{ASplit} l'ensemble dont les éléments sont des couples formés d'un AS-squelette et d'un entier naturel. Nous aurons alors :

Pour toute $\mathbb{Z}/n\mathbb{Z}$-clause C telle qu'il existe au moins un $-P(k)$ dans C pour au moins un $k \in \mathbb{Z}/n\mathbb{Z}$, on construit l'ensemble $Squel_{ASplit}(C)$ de la façon décrite plus haut. Pour tout AS-squelette $Sq = \{\ \{-P_{i_1}, \ldots, -P_{i_{m'}}\},\ \{C'_0 \vee C_{m'}\}\ \}$ de $Squel_{ASplit}(C)$:

1. Si Sq n'est la première composante d'aucun élément de Tab_{ASplit}, alors on ajoute l'élément $(Sq, 1)$ à l'ensemble Tab_{ASplit} ;
2. Si par contre, il existe (Sq, p) appartenant à Tab_{ASplit}, on le remplace par $(Sq, p+1)$, et
   - si $p + 1 = n$, on génère les clauses $C'_0 \vee C_{m'} \vee -q$ et $+q \vee -P_{i_1}(x) \vee \ldots \vee -P_{i_{m'}(x)}$, où $q = \ulcorner -P_{i_1}(x) \vee \ldots \vee -P_{i_{m'}(x)} \urcorner$;
   - sinon, on ne fait rien.

Maintenant que nous avons montrer comment en pratique on pouvait générer les clauses des règles ($ASplit$) et ($Exit$), nous pouvons apporter les modifications nécéssaires à notre prouveur.

5.3.3.2 Nouvelles règles pour le prouveur

Notons $AC\text{-}Res^{\mathsf{sel}}_{\succ}(C, \mathcal{O})$ l'ensemble des résolvants binaires restreints, ordonnés avec $\succ$ et sélectionnés avec sel, de C avec toutes les clauses de $\mathcal{O}$. Les clauses considérées dans cette section étant des clauses de Horn, nous n'avons pas besoin de la règle de factorisation positive. La liste $\mathcal{N}_F$ de clauses n'a alors plus lieu d'être, mais pour ne pas restreindre la généralité, celle-ci apparaîtra toujours dans les règles (elle sera en fait toujours vide), ainsi que la règle de factorisation positive. Notons alors $AC\text{-}Fac^{\mathsf{sel}}_{\succ}(C)$ l'ensemble des facteurs positifs de C, qui en pratique sera toujours l'ensemble vide.

La règle *Splitting négatif* devant être appliquée au plus tôt pour garantir la garantir la complétude de notre approximation, il est implicitement entendu en figure 5.7, que toutes les autres règles sont appliquées pour la clause C uniquement si celle-ci n'est pas splittable par la règle de splitting de blocs négatifs. Notons que la règle *Splitting négatif* remplace la règle du splitting sans splitting de la figure 4.7.

Les règles *E-Instantiation*, *S-Génération* et *Génération* sont rajoutées au prouveur, les deux dernières faisant appel à l'algorithme que nous avons décrit plus haut implémentant les règles ($ASplit$) et ($Exit$).

Notons que la règle *E-Instantiation* fait apparaître deux cas, selon que la clause traitée est une $\oplus$-clause triviale ou non. En effet, l'instantiation d'une $\oplus$-clause triviale C est toujours subsumée par C (même si dans certains cas, la subsomption n'est pas linéaire), ce qui conduirait à éliminer un nombre non négligeable de clauses instanciées par la règle ($Inst$). C'est pourquoi, l'ensemble $Inst_n(C)$ pour une clause triviale C est ajouté à $\mathcal{P}r$ et non à $\mathcal{N}_R$.

La règle (*Déduction*) du prouveur est remplacée par la règle (*AC-Déduction*) décrite à la figure 5.7.

Nous disposons ainsi d'une procédure d'approximation complète pour des automates d'arbres alternants ou non, bidirectionnels modulo associativité et commutativité.

Splitting négatif

$\mathcal{N}_R \cup \{C\} \mid \mathcal{N}_F \mid \mathcal{P}r \mid \mathcal{O} \Longrightarrow \mathcal{N}_R \cup \{C_1, C_2\} \mid \mathcal{N}_F \mid \mathcal{P}r \mid \mathcal{O}$

si C_1 et C_2 sont déductibles de C par la règle de splitting de blocs négatifs

$\mathbb{Z}/n\mathbb{Z}$-Instantiation

$\mathcal{N}_R \cup \{C\} \mid \mathcal{N}_F \mid \mathcal{P}r \mid \mathcal{O} \Longrightarrow \mathcal{N}_R \cup Inst_n(C) \mid \mathcal{N}_F \mid \mathcal{P}r \cup \{C\} \mid \mathcal{O}$

si C n'est ni une tautologie, ni subsumée par une clause de $\mathcal{P}r \cup \mathcal{O}$
si C est une $\oplus$-clause non triviale
si $Inst_n(C)$ est déductible de C par la règle $(Inst)$

$\mathcal{N}_R \cup \{C\} \mid \mathcal{N}_F \mid \mathcal{P}r \mid \mathcal{O} \Longrightarrow \mathcal{N}_R \mid \mathcal{N}_F \mid \mathcal{P}r \cup \{C\} \cup Inst_n(C) \mid \mathcal{O}$

si C n'est ni une tautologie, ni subsumée par une clause de $\mathcal{P}r \cup \mathcal{O}$
si C est une $\oplus$-clause triviale
si $Inst_n(C)$ est déductible de C par la règle $(Inst)$

S-Génération

$\mathcal{N}_R \cup \{C\} \mid \mathcal{N}_F \mid \mathcal{P}r \mid \mathcal{O} \Longrightarrow \mathcal{N}_R \cup \{C_1' \cup C_2'\} \mid \mathcal{N}_F \mid \mathcal{P}r \cup \{C\} \mid \mathcal{O}$

si C n'est ni une tautologie, ni subsumée par une clause de $\mathcal{P}r \cup \mathcal{O}$
et si C_1' et C_2' sont déductibles de C par la règle $(ASplit)$

Génération

$\mathcal{N}_R \cup \{C\} \mid \mathcal{N}_F \mid \mathcal{P}r \mid \mathcal{O} \Longrightarrow \mathcal{N}_R \cup \{C'\} \mid \mathcal{N}_F \mid \mathcal{P}r \cup \{C\} \mid \mathcal{O}$

si C n'est ni une tautologie, ni subsumée par une clause de $\mathcal{P}r \cup \mathcal{O}$
et si C' est déductible de C par la règle $(Exit)$

AC-Déduction

$\emptyset \mid \emptyset \mid \mathcal{P}r \cup \{C\} \mid \mathcal{O} \Longrightarrow \mathcal{N}_R \mid \mathcal{N}_F \mid \mathcal{P}r \mid \mathcal{O} \cup \{C\}$

avec $\mathcal{N}_R = AC\text{-}Res^{\mathsf{sel}}_{\succ}(C, \mathcal{O})$ et $\mathcal{N}_F = AC\text{-}Fac_{\succ}(C)$

Fig. 5.7: Nouvelles règles du démonstrateur

Chapitre 6

Application

La contribution que nous apportons ici utilise le module des termes associatifs et commutatifs que nous avons présenté en section 5.3, et s'applique à la vérification d'un protocole cryptographique d'établissement d'une clef de groupe (section 6.1). Nous décrirons ce protocole, appelé IKA.1 (ou GDH.2), en section 6.1.1, puis nous montrerons comment nous l'avons représenté sous forme de clauses de Horn (section 6.1.2), et les résultats que nous avons obtenu (section 6.1.3). Enfin, nous exposerons, comme un travail futur, les optimisations et les améliorations que nous pouvons effectuer sur notre prouveur (section 6.2).

6.1 Group Diffie-Hellman key distribution : GDH.2

Le protocole IKA.1, formellement connu sous le nom GDH.2, a été introduit dans [STW96] comme une extension du protocole d'échange de clef entre deux participants de W. Diffie and M. Hellman ([DH76]). Il est utilisé pour créer une clef de groupe dans la suite de protocoles CLIQUES.

Le but du protocole pour un groupe de participants $\mathcal{M}_1, \ldots, \mathcal{M}_n$, est d'établir une clef commune qui pourra être utilisée pour des communications ultérieures.

6.1.1 Description

Le protocole IKA.1 a été conçu pour un type particulier d'intrus appelé *pure eavesdropper*, que l'on peut traduire par *voyeur pur*. Ici, on va distinguer trois différents types d'intrus : celui cité ci-dessus qui ne peut que voir ce qui circule sur le réseau mais qui ne peut pas *prendre part* au protocole, c'est-à-dire qu'il ne peut pas construire de messages, effacer des messages, ou rediriger des canaux de communication, mais il peut tout de même déduire des choses de ce qu'il voit passer sur le réseau ; l'intrus *replay* ou *rejoueur* qui a les même capacités que l'intrus précédent, et, contrairement à celui-ci, il peut interférer au cours du déroulement du protocole, mais seulement en rejouant des messages qui ont déjà circulé ; et enfin l'intrus à la *Dolev-Yao* qui est capable de construire et de décomposer les messages qu'il voit passer à tout moment du déroulement du protocole, et qu'il peut envoyer sur le réseau. Nous verrons un peu plus loin comment nous allons distinguer ces trois types d'intrus.

Supposons que nous disposons d'un symbole de fonction libre unaire e, qui va être utilisé comme une façon d'encapsuler des secrets. Supposons aussi que l'on dispose d'un symbole $\oplus$ associatif et commutatif, d'élément neutre $\mathbf{0}$. Chaque participant pourra créer $e(M)$ à partir de M, et $e(M \oplus M')$ à partir de $e(M)$ et de M', mais ne pourra pas faire plus. Afin que $\mathcal{M}_1$ et $\mathcal{M}_2$ puissent obtenir une clef commune, $\mathcal{M}_1$ choisit un secret M_1, envoie à $\mathcal{M}_2$ le message $e(M_1)$; ensuite, $\mathcal{M}_2$ choisit un autre secret M_2 et envoie le message $e(M_2)$ à $\mathcal{M}_1$. Ainsi, tous deux peuvent calculer le secret commun $e(M_1 \oplus M_2)$: $\mathcal{M}_1$ peut le calculer à partir de M_1 et de $e(M_2)$, et $\mathcal{M}_2$ peut le calculer à partir de M_2 et de $e(M_1)$.

Ceci peut être implémenté en utilisant une exponentiation modulaire, en codant $e(M)$ par $\alpha^M \bmod N$ pour des nombres α et N bien choisis, l'opérateur $\oplus$ par la multiplication, et $\mathbf{0}$ par 1. D'autres implémentations utilisent le codage g^M pour $e(M)$, où g est un générateur d'un groupe, pas nécessairement $(\mathbb{Z}/n\mathbb{Z})^*$, typiquement des groupes générés par des courbes elliptiques ou des courbes hyperelliptiques modulo N.

Pour décrire le protocole IKA.1, nous aurons aussi besoin d'un symbole de fonction binaire *cons* et d'une constante *nil* pour représenter les listes. On utilisera souvent la notation $M_1; \ldots; M_n$ pour le terme $cons(M_1, (cons(M_2, \ldots, cons(M_n, nil) \ldots))$.

La figure 6.1 présente, pour un n fixé, le protocole IKA.1 faisant intervenir n participants. La notation $\sum_{1 \leq i \leq n} N_i$ utilisée dans les messages échangés entre les participants, signifie $N_1 \oplus \ldots \oplus N_n$, l'ordre des N_i n'ayant pas d'importance puisque le symbole $\oplus$ est commutatif.

$$
\begin{array}{lclcl}
\mathcal{M}_i & \longrightarrow & \mathcal{M}_{i+1} & : & e(\sum_{\substack{1 \leq j \leq i \\ j \neq 1}} N_j); \ldots; e(\sum_{\substack{1 \leq j \leq i \\ j \neq i}} N_j); \\
 & & & & e(\sum_{1 \leq j \leq i} N_i) \\
\mathcal{M}_n & \longrightarrow & \mathcal{M}_{1\ldots(n-1)} & : & e(\sum_{\substack{1 \leq j \leq n \\ j \neq 1}} N_j); \ldots; e(\sum_{\substack{1 \leq j \leq n \\ j \neq n-1}} N_j)
\end{array}
$$

FIG. 6.1: Protocole GDH-2 pour n participants

pour avoir une meilleure idée du fonctionnement du protocole, la figure 6.2 en donne une illustration dans le cas d'un groupe de quatre participants.

Tout d'abord, le protocole effectue une phase *montante* ("*upflow*") : $\mathcal{M}_1$ envoie à $\mathcal{M}_2$ la paire $e(0); e(N_1)$ où N_1 est une donnée fraîche créée par $\mathcal{M}_1$, un nonce ; ensuite $\mathcal{M}_2$ $e(N_2); e(N_1); e(N_1 \oplus N_2)$ à $\mathcal{M}_3$, où N_2 est un autre nonce frais différent de N_1, puis $\mathcal{M}_3$ envoie $e(N_2 \oplus N_3); e(N_1 \oplus N_3); e(N_1 \oplus N_2); e(N_1 \oplus N_2 \oplus N_3)$ à $\mathcal{M}_4$, où N_3 est un nouveau nonce différent des deux précédents.

Une fois ceci effectué, $\mathcal{M}_4$ passe à la phase *descendante* ("*downflow*"), et diffuse à tous le message $e(N_2 \oplus N_3 \oplus N_4); e(N_1 \oplus N_3 \oplus N_4); e(N_1 \oplus N_2 \oplus N_4)$, où N_4 est un nouveau nonce créé par $\mathcal{M}_4$.

À partir de ce message, chacun peut calculer la clef de groupe $e(N_1 \oplus N_2 \oplus N_3 \oplus N_4)$.

Ainsi, pour une description plus générale, chaque agent $\mathcal{M}_i$ reçoit un tuple de la forme $e(M_1); \ldots; e(M_i)$, effectue l'opération $e(M_j \oplus N_i)$ à partir de $e(M_j)$, $1 \leq j \leq i$, et de son nonce N_i, et copie le dernier élément du tuple dans le message qu'il va renvoyer. La longueur des tuples envoyés entre les membres du groupe est alors toujours égale au nombre qui identifie le receveur (ce nombre étant i pour le participant $\mathcal{M}_i$). Finalement, le dernier membre du groupe, qui lui joue un rôle bien particulier, reçoit un tuple de longueur n dont le dernier élément lui permet de calculer sa clef de groupe avec son nonce N_n. Puis $\mathcal{M}_n$ envoie un même message à tous les autres participants du groupe, qui est un tuple de $n-1$ clefs partielles construites avec son nonce N_n. Ainsi, chaque $\mathcal{M}_i$ peut calculer la clef à partir du $i^{ème}$ élément du tuple et de N_i.

Dans [MD02], Miller et Denker trouvent une attaque à ce protocole, en considérant un intrus capable de rejouer des messages passés, mais pas d'en construire lui-même, que nous avons appelé l'intrus *rejoueur*. Cette attaque est effective grâce à l'ambiguïté du format des messages. Elle est illustrée en figure 6.3. Cette attaque n'est possible que dans la mesure où $\mathcal{M}_1$ ne vérifie pas la taille du message qu'il reçoit, et nous verrons que nous pouvons modéliser cela, mais que nous pouvons tout aussi bien modéliser le cas où $\mathcal{M}_1$ la vérifie, et ainsi cette attaque ne peut plus avoir lieu.

Cette attaque met en scène le cas où $\mathcal{M}_1$ récupère une mauvaise clef de groupe, mais elle peut être généralisée à chaque membre du groupe. Ici, $\mathcal{M}_1$ reçoit un message lors de la phase descendante, et multiplie le premier élément du message par l'exponentielle de N_1, comme le prévoit le protocole. Or, l'intrus a remplacé le message qu'il aurait dû recevoir par $e(0); e(N_1)$, donc $\mathcal{M}_1$ calcule la mauvaise clef $e(0 \oplus N_1)$, soit $e(N_1)$ qui est connue de l'intrus. Mais si $\mathcal{M}_1$ avait vérifié la longueur du message, l'attaque n'aurait pas pu avoir lieu, comme nous l'avons mentionné ci-dessus.

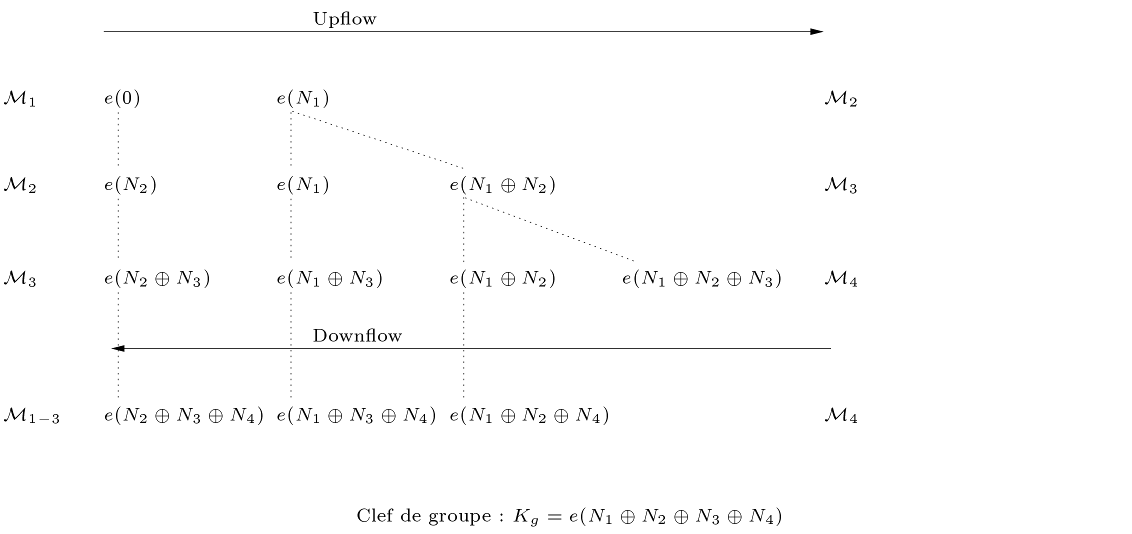


FIG. 6.2: Protocole GDH-2 pour 4 participants

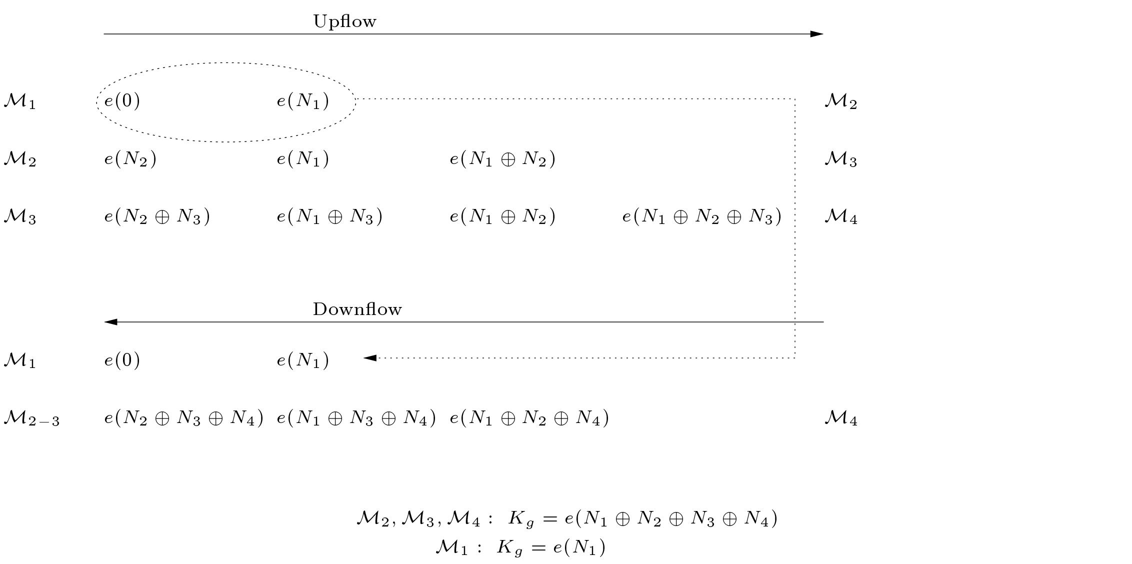


FIG. 6.3: Une attaque pour GDH.2

6.1.2 Mise sous forme clausale

L'arbre d'exécution du protocole est un arbre à n fils, le $i^{ème}$ fils correspondant au cas où c'est $\mathcal{M}_i$ qui exécute une étape du protocole. Supposons qu'à chaque nœud de l'arbre correspond une configuration qui est représentée par un n-uplet d'entiers compris entre 1 et n, la $i^{ème}$ composante désignant l'étape du protocole à laquelle se trouve $\mathcal{M}_i$.

Pour modéliser les communications, on considère que tout message est envoyé sur un canal unique, et que tout message reçu l'est par l'intermédiaire de ce canal unique.

Selon les capacités que l'on donne à l'intrus, les clauses générées ne vont pas être les mêmes. Dans le cas *pure eavesdropper*, l'intrus n'a aucune connaissance initiale, et ne peut que rejouer le dernier message qui a été envoyé. Dans le cas *replay*, l'intrus se souvient de tous les messages qui ont transités, il peut donc, à tout moment, envoyer un message qui a été déjà émis. Dans le cas d'un intrus à la *Dolev-Yao*, il a les capacités des intrus précédents, mais il peut aussi faire des exponentiations, effectuer certaines opérations sur les messages que nous allons expliciter un peu plus loin.

on appelle *configuration finale* une configuration C telle qu'il existe une configuration parente C' de C pour laquelle un $\mathcal{M}_i$ a terminé. A chaque nouvelle configuration C, on crée deux nouveaux symboles de prédicat qui seront notés k_C et E_C, et à chaque configuration finale C, un nouveau symbole de prédicat I_C.

Intuitivement, les différentes significations de ces prédicats sont les suivantes :

- $k_C(M)$ signifie “le message M est dans le canal à la configuration C”,
- $I_C(M)$ signifie “le message M est déductible par l'intrus à la configuration C”,
- $E_C(M)$ signifie “le message M a été lu à la configuration C ou avant”,
- $Obt_{C,i}(M)$ signifie “$\mathcal{M}_i$ a obtenu le message M à la configuration C.

Pour chaque configuration C, notons C_{av} la configuration parente de C. Selon le modèle utilisé pour les capacités de l'intrus, on aura tout ou une partie de l'ensemble de clauses suivant :

$$\begin{aligned}
E_C(x) &\Leftarrow k_C(x) && \text{L'intrus peut lire un message du buffer} && (6.1)\\
E_C(x) &\Leftarrow E_{C_{av}}(x) && \text{Un message lu en } C_{av} \text{ est lu en } C && (6.2)\\
k_C(x) &\Leftarrow k_{C_{av}}(x) && \text{Les messages passés sont dans le canal} && (6.3)\\
k_C(nil) & && \text{La liste vide est dans le canal} && (6.4)\\
k_C(0) & && \text{0 est dans le canal} && (6.5)\\
k_C(e(x)) &\Leftarrow k_C(x) && \text{Si } x \text{ est dans le canal, } e(x) \text{ aussi} && (6.6)\\
k_C(e(x \oplus y)) &\Leftarrow k_C(e(x)), k_C(y) && \text{Le canal peut faire des exponentiations} && (6.7)\\
k_C(cons(x,y)) &\Leftarrow k_C(x), k_C(y) && \text{Le canal peut construire des listes} && (6.8)\\
k_C(x) &\Leftarrow k_C(cons(x,y)) && \text{Le canal peut lire les têtes de liste} && (6.9)\\
k_C(y) &\Leftarrow k_C(cons(x,y)) && \text{Le canal peut lire les queues de liste} && (6.10)\\
k_C(x \oplus y) &\Leftarrow k_C(x), k_C(y) && \text{Le canal peut additionner} && (6.11)\\
k_C(-x) &\Leftarrow k_C(x) && \text{Le canal peut calculer l'opposé} && (6.12)
\end{aligned}$$

Dans le cas d'un intrus *pure eavesdropper*, on ne générera que les clauses (6.1) et (6.2). Dans le cas d'un intrus *replay*, on générera les clauses (6.1) à (6.3). Et enfin dans le cas d'intrus *Dolev-Yao*, on générera toutes les clauses (6.1) à (6.12).

Et ensuite, pour chaque configuration finale C, on générera l'ensemble de clauses suivant :

$$I_C(x) \Leftarrow E_C(x) \quad \text{L'intrus lit un message} \tag{6.13}$$
$$I_C(nil) \quad \text{L'intrus connaît la liste vide} \tag{6.14}$$
$$I_C(0) \quad \text{L'intrus connaît 0} \tag{6.15}$$
$$I_C(e(x)) \Leftarrow I_C(x) \quad \text{L'intrus peut calculer } e(x) \text{ à partir de } x \tag{6.16}$$
$$I_C(e(x \oplus y)) \Leftarrow I_C(e(x)), I_C(y) \quad \text{L'intrus peut faire des exponentiations} \tag{6.17}$$
$$I_C(cons(x,y)) \Leftarrow I_C(x), I_C(y) \quad \text{L'intrus peut construire des listes} \tag{6.18}$$
$$I_C(x) \Leftarrow I_C(cons(x,y)) \quad \text{L'intrus peut lire les têtes de liste} \tag{6.19}$$
$$I_C(y) \Leftarrow I_C(cons(x,y)) \quad \text{L'intrus peut lire les queues de liste} \tag{6.20}$$
$$I_C(x \oplus y) \Leftarrow I_C(x), I_C(y) \quad \text{L'intrus peut additionner} \tag{6.21}$$
$$I_C(-x) \Leftarrow I_C(x) \quad \text{L'intrus peut calculer l'opposé} \tag{6.22}$$

Afin de mieux comprendre cette modélisation, nous allons dérouler un exemple d'un déroulement *normal* du protocole pour quatre participants.

À la configuration $C = 1$, $\mathcal{M}_1$ envoie son message :

$$k_1(e(0); e(N_1))$$

$\mathcal{M}_2$ reçoit le message à la configuration $C = 1$ et en renvoie un autre à la configuration $C = 2$:

$$k_2(e(x \oplus N_2); e(y); e(y \oplus N_2)) \Leftarrow k_1(e(x); e(y))$$

$\mathcal{M}_3$ reçoit le message à la configuration $C = 2$ et en renvoie un autre à la configuration $C = 3$:

$$k_3(e(x \oplus N_3); e(y \oplus N_3); e(z); e(z \oplus N_3)) \Leftarrow k_2(e(x); e(y); e(z))$$

$\mathcal{M}_4$ reçoit le message à la configuration $C = 3$ et en renvoie un autre à la configuration $C = 4$:

$$k_4(e(x_1 \oplus N_4); e(x_2 \oplus N_4); e(x_3 \oplus N_4)) \Leftarrow k_3(e(x_1); e(x_2); e(x_3); y)$$

Ici, $\mathcal{M}_4$ reçoit un message de longueur 3 plus la longueur de y, alors qu'il est censé recevoir un message de longueur 4. On peut spécifier cela en écrivant :

$$k_4(e(x_1 \oplus N_4); e(x_2 \oplus N_4); e(x_3 \oplus N_4)) \Leftarrow k_3(e(x_1); e(x_2); e(x_3); e(y))$$

$\mathcal{M}_1$ reçoit le dernier message à la configuration $C = 4$, et peut ainsi calculer la clef de groupe :

$$Obt_{4,1}(e(x \oplus N_1)) \Leftarrow k_4(e(x); y)$$

Encore une fois, la longueur de la liste reçue n'est pas spécifiée ci-dessus ; écrivons alors :

$$Obt_{4,1}(e(x \oplus N_1)) \Leftarrow k_4(e(x); e(y); e(z))$$

De même, $\mathcal{M}_2$, $\mathcal{M}_3$ et $\mathcal{M}_4$ peuvent calculer la clef, sans ou avec vérification de la longueur de la liste reçue :

$$Obt_{4,2}(e(y \oplus N_2)) \Leftarrow k_4(x; e(y); z)$$
$$Obt_{4,2}(e(y \oplus N_2)) \Leftarrow k_4(e(x); e(y); e(z))$$

$$Obt_{4,3}(e(y \oplus N_3)) \Leftarrow k_4(x; e(y))$$
$$Obt_{4,3}(e(z \oplus N_3)) \Leftarrow k_4(e(x); e(y); e(z))$$

$$Obt_{4,4}(e(y \oplus N_4)) \Leftarrow k_4(x; e(y))$$
$$Obt_{4,4}(e(y \oplus N_4)) \Leftarrow k_4(e(x_1); e(x_2); e(x_3); e(y))$$

Nous allons ensuite spécifier le secret pour chacun des participants, en modélisant l'intrus à une certaine configuration C par le prédicat I_C ; les clauses ci-dessous signifient : "à la configuration C, l'intrus ne connaît pas la donnée que $\mathcal{M}_i$ a obtenu à cette même configuration".

$$\begin{array}{c}\perp \Leftarrow Obt_{4,1}(x), I_4(x)\\ \perp \Leftarrow Obt_{4,2}(x), I_4(x)\\ \perp \Leftarrow Obt_{4,3}(x), I_4(x)\\ \perp \Leftarrow Obt_{4,4}(x), I_4(x)\end{array}$$

Le lecteur aura peut-être remarqué que les prédicats $Obt_{C,i}$ ne sont en fait pas utiles, mais nous les avons introduit pour plus de clarté. Nous pouvons en fait regrouper, pour chaque participant, les deux clauses d'obtention de la clef et de secret en une seule de la manière suivante (encore une fois sans ou avec vérification de la longueur de la liste reçue) :

$$\begin{array}{c}\perp \Leftarrow k_4(e(x); y), I_4(e(x \oplus N_1))\\ \perp \Leftarrow k_4(e(x); e(y); e(z)), I_4(e(x \oplus N_1))\end{array}$$

$$\begin{array}{c}\perp \Leftarrow k_4(e(x); e(y); z), I_4(e(y \oplus N_2))\\ \perp \Leftarrow k_4(e(x); e(y); e(z)), I_4(e(y \oplus N_2))\end{array}$$

$$\begin{array}{c}\perp \Leftarrow k_4(x; e(y)), I_4(e(y \oplus N_3))\\ \perp \Leftarrow k_4(e(x); e(y); e(z)), I_4(e(y \oplus N_3))\end{array}$$

$$\begin{array}{c}\perp \Leftarrow k_4(x; e(y)), I_4(e(y \oplus N_4))\\ \perp \Leftarrow k_4(e(x_1); e(x_2); e(x_3); e(y)), I_4(e(y \oplus N_4))\end{array}$$

Notons que cet exemple d'exécution du protocole que nous avons donné correspond à un bon déroulement. Pour prendre en compte tous les déroulements possibles, nous allons devoir générer les clauses correspondants à tous les entrelacements des différentes actions de chaque participant.

Étant donné un entier n fixé. Nous allons généraliser cela pour un groupe de n participants :

• Envoi de $\mathcal{M}_1$ à la configuration C :

$$k_C(e(0); e(N_1)) \tag{6.23}$$

• $\mathcal{M}_i$, $1 < i < n$, reçoit un message à la configuration C_{av} et en envoie un à la configuration C :

$$k_C(e(x_1 \oplus N_i); \ldots; e(x_{i-1} \oplus N_i); e(x_i); e(x_i \oplus N_i)) \Leftarrow k_{C_{av}}(e(x_1); \ldots; e(x_i)) \tag{6.24}$$

• $\mathcal{M}_n$ reçoit un message à la configuration C_{av} et en envoie un à la configuration C, la première clause ne prenant pas en compte la vérification de la longueur de la liste, contrairement à la deuxième :

$$k_C(e(x_1 \oplus N_n); \ldots; e(x_{n-1} \oplus N_n)) \Leftarrow k_{C_{av}}(e(x_1); \ldots; e(x_{n-1}); x_n) \tag{6.25}$$

$$k_C(e(x_1 \oplus N_n); \ldots; e(x_{n-1} \oplus N_n)) \Leftarrow k_{C_{av}}(e(x_1); \ldots; e(x_n)) \tag{6.26}$$

• Conditions de secret pour $\mathcal{M}_i$, $1 \leq i < n$, et pour toute configuration C finale pour $\mathcal{M}_i$, la première clause ne prenant pas en compte la vérification de la longueur de la liste, la deuxième le faisant :

$$\perp \Leftarrow k_C(e(x_1); \ldots; e(x_i); y), I_C(e(x_i \oplus N_i)) \tag{6.27}$$

$$\perp \Leftarrow k_C(e(x_1); \ldots; e(x_{n-1}), I_C(e(x_i \oplus N_i)) \tag{6.28}$$

• Conditions de secret pour $\mathcal{M}_n$ et pour toute configuration C finale pour $\mathcal{M}_n$:

$$\perp \Leftarrow k_C(x; e(y)), I_C(e(y \oplus N_n)) \tag{6.29}$$

$$\perp \Leftarrow k_C(e(x_1); \ldots; e(x_n)), I_C(e(x_n \oplus N_n)) \tag{6.30}$$

Les clauses (6.11), (6.21) et (6.23) à (6.30) ne sont cependant pas au bon format.

Nous allons devoir transformer cet ensemble de clauses. Avant de décrire cette transformation, introduisons la notion de sur-approximation. Une *sur-approximation* $\mathcal{A}$ d'un ensemble de clauses S est complète mais pas nécessairement correcte. Autrement dit, si S est insatisfiable, alors $\mathcal{A}(S)$="non", c'est-à-dire si $\mathcal{A}(S)$ retourne "oui", alors S est satisfiable, sinon on ne sait rien. Dans le cas qui nous intéresse où S décrit un protocole cryptographique, les hypothèses de sécurité et les propriétés à prouver, si $\mathcal{A}(S)$ retourne "oui", alors il n'y a pas d'attaque sur le protocole, de façon sûre. On obtient ainsi une preuve de sécurité. Par contre, si $\mathcal{A}(S)$ retourne "non", on ne peut pas garantir qu'il y ait une attaque.

Nous allons nous servir d'une procédure de sur-approximation valide pour tout ensemble de clauses de Horn.

Pour cela on utilise un algorithme d'*aplatissement* qui opère sur un ensemble S de clauses, décrit en figure 6.1.2.

Tant qu'il existe un terme t qui n'est ni une variable, ni un terme plat dans une clause $C = C_0 \vee \pm P(t)$ de S, posons $t = f(t_1, \dots, t_n)$, puis :

- Si $\pm$ est le signe $-$, créer n prédicats unaires frais P_j, $1 \leq j \leq n$, et remplacer C dans S par les $n+1$ clauses :

$$C_0 \vee -P_1(t_1) \vee \dots \vee -P_n(t_n)$$

$$-P(f(X_1, \dots, X_n)) \vee +P_j(X_j) \quad (1 \leq j \leq n)$$

où $X_1, \dots, X_n$ sont n variables distinctes deux à deux.

- Si $\pm$ est le signe $-$, créer n prédicats unaires frais P_j, $1 \leq j \leq n$, et remplacer C dans S par les $n+1$ clauses :

$$+P(f(X_1, \dots, X_n)) \vee -P_1(X_1) \vee \dots \vee -P_n(X_n)$$

$$C_0 \vee +P_j(t_j)) \quad (1 \leq j \leq n)$$

où $X_1, \dots, X_n$ sont n variables distinctes deux à deux.

FIG. 6.4: Procédure d'aplatissement d'un ensemble S de clauses de Horn

Le fait que l'approximation soit valide est assez immédiate, puisque la résolution ordonnée avec sélection nous permet de retrouver l'ensemble de clauses S initial à partir de l'ensemble de clauses S' transformé par la procédure. Ainsi, si S' est satisfiable, alors S est satisfiable.

Cependant, même si dans le cas général, cette procédure nous fournit une sur-approximation, dans le cas particulier du format des clauses que nous avons, cette procédure nous fournit un ensemble de clauses équivalent, pour la satisfiabilité et l'insatisfiabilité, à l'ensemble de départ, comme il est mentionné dans [GLV02] : dans le cas d'un intrus Dolev-Yao, envoyer des listes d'éléments ou envoyer chaque élément séparément a le même effet, et dans le cas d'un voyeur pur, chaque k_C reconnaît de toute façon au plus une valeur.

Cet algorithme ne nous fournit pas tout à fait des clauses de la forme voulue. Pour la plupart d'entre elles, il suffit d'appliquer sur les clauses en question le splitting sans splitting. Mais certaines clauses peuvent encore subsister dans le cas où elles ne peuvent pas être splittées. C'est typiquement le cas des clauses de la forme $P(f(x, y)) \Leftarrow Q(g(y, z))$ où les ensembles de variables libres des deux littéraux ne sont pas disjoints. Ce type de clauses sera alors transformé de la façon suivante : on introduit un prédicat auxiliaire Q' et on remplace par les deux clauses $P(f(x, y)) \Leftarrow Q'(y)$ et $Q'(y) \Leftarrow Q(g(y, z))$. L'approximation reste valide pour les mêmes raisons. Cependant, pour le format particulier des clauses qui nous intéressent, nous avons pu vérifier qu'un tel cas ne se produit pas.

Donc, à splitting près, cet algorithme fournit des clauses ayant le bon format. Or, notre algorithme de résolution fait du splitting dès que possible. C'est donc la procédure de splitting du démonstrateur qui finira de mettre les clauses au format voulu.

6.1.3 Résultats de MoP

Nous allons dans cette section décrire les résultats que nous avons obtenus pour la vérification du protocole IKA.1. Rappelons que la procédure d'approximation que nous avons définie est complète (voir section 5.3.2), c'est-à-dire que si l'ensemble de clauses de départ est insatisfiable alors la procédure déduira la clause vide, autrement dit, si la procédure ne déduit pas la clause vide, alors l'ensemble de départ est satisfiable. Ainsi, une non déduction de la clause vide nous donnera une preuve de sécurité du protocole. Par contre, dans le cas d'une déduction de la clause vide, nous ne pouvons pas conclure directement, mais nous pouvons tout de même examiner la trace de preuve pour voir si elle correspond à une attaque ou non.

Les clauses décrivant le protocole sont générées de manière automatique pour toutes les configurations possibles, de même que l'aplatissement des clauses. Ceci ayant déjà été décrit plus haut, nous n'y revenons pas.

Présentons d'ores et déjà les résultats que nous avons obtenu avec cette approximation. Voici les tableaux récapitulant ces résultats sur un Dell PowerEdge 4600, processeur Intel Xeon à 2,4 GHz avec 512 Ko de mémoire cache, disposant d'un gigaoctet de mémoire vive, et tournant sous Linux 2.4.18-3.

Nous avons choisi d'instancier les clauses sur $\mathbb{Z}/3\mathbb{Z}$ pour être relativement fin dans l'approximation (c'est la signification de la mention $n = 3$ dans les tableau ci-dessous), sans pour autant générer un nombre de clauses trop important qui rendrait le programme impratiquable.

Intrus	Voyeur Pur 2 participants $n = 3$	Replay 2 participants $n = 3$	Dolev-Yao 2 participants $n = 3$
Résultat	protocole sûr	$\square$ déduit	$\square$ déduit
Clauses initiales	118	122	162
Clauses créées	1 705	1 755	3 273
par résolution	663	644	1 792
par splitting	27	29	74
par instantiation	1 002	1 068	1 392
par Exit	13	14	15
par AS-Split	0	0	0
Clauses supprimées	319	301	1 299
tautologies	28	28	50
par subsomption	291	273	1 249
Temps (en secondes)	1,10	1,17	2,65
Mémoire (en KB)	3 276	3 024	3 548

Intrus	Voyeur Pur 3 participants $n = 3$	Replay 3 participants $n = 3$	Dolev-Yao 3 participants $n = 3$
Résultat	protocole sûr	$\square$ déduit	$\square$ déduit
Clauses initiales	596	611	761
Clauses créées	11 505	4 232	2 593
par résolution	3 333	778	430
par splitting	183	88	68
par instantiation	7 926	3 315	2 046
par Exit	63	51	49
par AS-Split	0	0	0
Clauses supprimées	1 491	447	431
tautologies	105	83	152
par subsomption	1 383	364	279
Temps (en secondes)	94,80	7,82	3,10
Mémoire (en KB)	7 968	5 124	4 560

Intrus	Voyeur Pur 4 participants $n = 3$	Replay 4 participants $n = 3$	Dolev-Yao 4 participants $n = 3$
Résultat	protocole sûr	$\square$ déduit	$\square$ déduit
Clauses initiales	3 366	3 430	4 070
Clauses créées	100 915	11 645	10 175
par résolution	18 939	1 027	4 532
par splitting	923	345	320
par instantiation	80 688	9 972	9 024
par Exit	365	301	299
par AS-Split	0	0	0
Clauses supprimées	7 858	957	1 236
tautologies	448	328	642
par subsomption	7 410	629	594
Temps (en secondes)	20 566,01	87,19	97,20
Mémoire (en KB)	69 564	12 676	12 676

Dans le cas des modèles d'intrus replay et Dolev-Yao, nous avons pu vérifier en examinant les traces du programme, que les déductions de la clause vide $\square$ obtenues correspondent effectivement à l'attaque sur le protocole IKA.1 que nous avons décrite en section 6.1. Nous disposons de traces de preuve présentées de la même façon que les traces de preuve de SPASS [Wei99a], c'est-à-dire comme une succession de toutes les clauses ayant participé à la déduction de la clause vide, accompagnée pour chacune d'entre elles des clauses et de la règle qui ont servi à la déduction. Mais ces traces ne sont pas encore très facilement lisibles dans la mesure où les clauses de l'ensemble initial sont les clauses aplaties (par l'algorithme présenté en section 6.1.2, figure 6.1.2) à partir de la modélisation du protocole. Il faut donc encore fournir un travail d'analyse pour repérer les clauses non-aplaties mises en jeu dans la déduction, afin de vérifier que la preuve correspond ou non à une attaque, et nous aurions besoin que cette analyse soit automatique (voir section 6.2.2).

Il est à noter que le programme déduit la clause vide $\square$ dans les cas présentés dans les tableaux ci-dessus pour les modèles d'intrus replay et Dolev-Yao, ce qui est normal, puisque ces cas correspondent à des modèles présentant des attaques. Il est donc nécessaire de tester le programme aussi sur des protocoles pour lesquels aucune attaque n'est connue. De ce côté, nos résultats sont aussi satisfaisants puisque dans le cas des intrus voyeurs purs, la clause vide n'est pas déduite, nous donnant ainsi une preuve de sécurité du protocole IKA.1 pour ce modèle d'intrus, pour jusqu'à 4 participants. Ceci confirme les buts des inventeurs de ce protocole [STW96] qui voulaient simplement que le protocole soit sûr pour un intrus voyeur pur.

L'interprétation des traces du prouveur est encore aujourd'hui une tâche malaisée : on dispose juste de la suite de toutes les clauses produites ou effacées, et nous aurions besoin par exemple d'en extraire un arbre de preuve pour faciliter la compréhension des preuves (voir section 6.2.2).

Le prouveur compte aujourd'hui 5 383 lignes de Caml, et a nécessité 9 mois de développement. Comme nous l'avons déjà dit, il n'est pas spécialement optimisé ; nous avons préféré favoriser la modularité plutôt que l'efficacité. Notamment, comme nous l'avons déjà mentionné, la subsomption n'est pas spécialisée à un langage de termes du premier ordre : nous n'avons aucune structure efficace permettant le test rapide de subsomption comme les arbres de discrimination d'Otter [Kal01] ou les arbres de code de Vampire [RV01], et nous énumérons tout simplement toutes les clauses de l'ensemble courant à chaque test de subsomption d'une clause, en particulier à chaque fois qu'une clause est produite par résolution. Il est cependant peu clair comment étendre les techniques d'Otter ou de Vampire à des familles de termes symboliques quelconques.

Malgré ce handicap a priori, on remarque que notre prouveur se débrouille quand même bien, et sait traiter des ensembles de plusieurs milliers de clauses.

Nous pouvons remarquer qu'aucune clause n'est générée par la règle (*AS-Split*). A l'heure actuelle, nous n'avons pas déterminé la cause de ce phénomène, mais nous voyons au moins deux explications possibles :

- soit le cas particulier des clauses modélisant le protocole IKA.1 ne nécessite pas l'intervention de cette règle,
- soit, dans le cas général, nous pourrions en fait tout simplement nous passer de cette règle pour la procédure d'approximation, mais ceci demanderait à être prouvé.

Une autre remarque est que, bien que l'algorithmique de la gestion des $\oplus$-clauses (instantiation dans $\mathbb{Z}/n\mathbb{Z}$, résolution propositionnelle, et génération de candidats modifiés) est a priori coûteuse en temps, et surtout en nombre de clauses à traiter, le nombre total de clauses créées par l'algorithme de résolution est tout à fait raisonnable ; en effet, même si le nombre de $\mathbb{Z}/n\mathbb{Z}$-clauses (pour $n = 3$) créées par le prouveur est important, on remarque que peu de clauses ordinaires du premier ordre sont en fait créées par résolution binaire. Ainsi, la fonction de sélection sur les clauses permet de limiter le nombre de clauses créées par résolution binaire, ce qui limite de fait le nombre de $\mathbb{Z}/n\mathbb{Z}$-clauses créées.

Une analyse formelle de ce protocole, parmi la suite de protocoles Cliques, est faite dans [PQ01], mais c'est la première fois, à notre connaissance qu'une analyse formelle automatique est réalisé sur le protocole IKA.1.

Dans la suite, nous suggérons quelques améliorations et optimisations possibles.

6.2 Optimisations - Améliorations

6.2.1 Optimisations

Indexation Pour améliorer la rapidité du prouveur, nous souhaiterions inclure des algorithmes d'indexation des objets dans les clauses et des clauses elles-mêmes, qui servent à trouver rapidement de quelles clauses on peut calculer les résolvants, quelles clauses sont subsumées par l'ensemble courant S, et quelles clauses de S sont subsumées par une clause donnée.

Cela se présenterait par exemple sous la forme de tables associant un littéral aux clauses dans lesquelles ce littéral est maximal, de telles tables devant être mises à jour à chaque étape d'élimination de clauses, de résolution ou de splitting sans splitting.

Splitting sans splitting Le splitting sans splitting demande de fabriquer des symboles de prédicats $\ulcorner C \urcorner$ tels que pour tout renommage ρ des variables de C, on ait $\ulcorner C \urcorner = \ulcorner C\rho \urcorner$. Une optimisation consisterait à utiliser des tables de hachage liant les clauses et leur prédicat de splitting.

6.2.2 Améliorations

Modularité En l'état actuel, les règles spécifiques de la section 5.3.3 que nous avons rajoutées au prouveur sont intégrées au module général de résolution et ne sont appliquées que lorsque le module de termes utilisé est celui des termes modulo AC de la section 5.3. Afin d'augmenter la modularité du prouveur, nous devrions définir un module indépendant permettant ainsi de ne pas avoir à modifier le module de résolution lorsque l'on souhaite rajouter des règles, mais il est cependant peu clair à l'heure actuelle sur la façon dont il faudrait procéder.

Splitting classique Le splitting sans splitting est plus efficace que le splitting classique dès qu'il s'agit de clauses de Horn, mais certains résultats théoriques laissent à penser qu'il le serait moins pour des clauses qui ne sont pas de Horn. En effet, on peut montrer que la satisfiabilité d'ensembles de clauses de Horn simples ou complexes modulo la théorie équationnelle vide (section 2.4.3) est dans DEXPTIME par résolution ordonnée avec sélection et splitting sans splitting (le problème est DEXPTIME-complet), et d'autre part que la satisfiabilité d'ensembles de clauses simples ou complexes modulo la théorie équationnelle vide, mais pas restreintes à être de Horn est dans NEXPTIME par résolution ordonnée avec sélection et splitting *classique* (le problème est NEXPTIME-complet). Dans le cas de Horn, le splitting classique ne fournit qu'une procédure dans

NEXPTIME, qui n'est donc pas optimale ; dans le cas non Horn, c'est le splitting sans splitting qui ne serait pas optimal, car fournissant une procédure en temps doublement exponentiel.

Cela améliorerait donc le prouveur si on donnait la possibilité de choisir le splitting classique ou le splitting sans splitting. Nous avons en fait préparé le terrain depuis le début : notre prouveur n'utilise aucune structure impérative (sauf dans quelques rares fonctions utilitaires), et a donc un comportement purement applicatif, c'est-à-dire sans aucun effet de bord observable. Il est donc facile de lancer le prouveur sur deux, trois ou davantage d'ensembles de clauses. Un prouveur impératif comme Otter ou Vampire est essentiellement incompatible avec le splitting classique : une étape de résolution sur une branche modifierait les données d'une autre branche.

Trace de preuve Pour le moment, comme nous l'avons vu dans ce chapitre, pour un ensemble S de clauses donné, le prouveur, s'il termine sur cette ensemble de clauses, donne un résultat de satisfiabilité ou un résultat l'insatisfiabilité, accompagné, dans le cas où la clause vide est déduite, d'une trace de preuve, constituées des étapes du processus de résolution qui ont été utiles à trouver la clause vide. Ceci facilite l'interprétation en termes d'attaques possibles sur des protocoles cryptographiques. Cependant, dans le cas du protocole IKA.1, nous aimerions effectuer une analyse automatique de cette trace. En effet, la mise sous forme clausale du protocole nécessite un aplatissement des clauses (voir section 6.1.2) afin d'obtenir le format voulu, faisant apparaître un certain nombre de prédicats auxiliaires. Nous voudrions donc à partir des clauses de la trace retrouver, si cela est possible (c'est-à-dire si les clauses intervenant dans la déduction le permettent), les clauses non aplaties desquelles elles dérivent. Ainsi, nous pourrions clairement voir si une déduction de la clause vide correspond à une attaque.

Conclusion

Nous avons défini un cadre général de résolution permettant de traiter différentes logiques clausales fondées sur une notion de termes abstraits généralisant la notion de termes, et de termes modulo une théorie équationnelle. Avec la notion de familles de termes symboliques, nous avons défini les ensembles de clauses symboliques, fournissant ainsi une généralisation des termes du premier ordre. Nous avons montré en particulier que les termes finis du premier ordre, les termes rationnels, les termes modulo une théorie équationnelle, et en particulier les λ-termes modulo β ou $\beta\eta$ sont des familles normales de termes symboliques, dénotant ainsi la généralité effective de telles familles. Nous avons démontré les résultats fondamentaux de correction et de complétude de la résolution abstraite ordonnée avec sélection, élimination de tautologies et de clauses linéairement subsumées, avec splitting classique ou splitting sans splitting, sur les ensembles de clauses symboliques.

Nous avons présenté l'architecture générale d'un prouveur modulaire que nous avons implémenté, dont le noyau est basé sur la résolution abstraite ordonnée avec sélection, élimination de tautologies et de clauses subsumées linéairement, avec splitting sans splitting, nous servant ainsi des résultats théoriques que nous établis. Nous avons implémenté différents modules de termes concrets : les termes finis du premier ordre, les termes rationnels, et une classe de termes du premier ordre modulo associativité et commutativité. Ces derniers termes nous ont servi à travailler sur des ensembles de clauses de Horn qui sont des automates d'arbres bidirectionnels non alternants modulo associativité et commutativité, le but affiché étant alors de modéliser le protocole cryptographique IKA.1, et par extension tout protocole fondé sur des primitives à la Diffie-Hellman. Nos travaux sont cependant de portée plus générale.

Même si la décidabilité des automates d'arbres bidirectionnels non alternants modulo associativité et commutativité est établie, la procédure de décision est peu utilisable en pratique. Nous avons alors présenté une procédure d'approximation dont nous avons démontré la complétude. Il est à noter que la complétude de cet algorithme est un résultat techniquement difficile. L'implémentation que nous en avons faite repose elle aussi sur des algorithmes non triviaux, notamment de génération de candidats modifiés à partir de leurs $\mathbb{Z}/n\mathbb{Z}$-instances. De plus, cette approximation est aussi complète pour les automates bidirectionnels *alternants* modulo associativité et commutativité, qui est une classe indécidable. Ce travail fournit donc une approximation sûre du problème de la vacuité d'intersection de ces automates, et, par conséquent, de la sécurité de protocoles cryptographiques utilisant des primitives associatives et commutatives, tels que IKA.1.

Nous avons obtenu des résultats satisfaisants avec notre prouveur puisque nous avons pu montrer que le protocole IKA.1 est sûr pour un intrus voyeur pur, jusqu'à quatre participants, confirmant ainsi les choix de conceptions des inventeurs de ce protocole [STW96]. C'est la première fois, à notre connaissance, qu'une analyse formelle automatique de ce protocole est effectuée.

Nous avons, enfin, suggéré des améliorations et des optimisations qui apporteraient certainement un gain significatif à l'ensemble de ces résultats.

Annexe A

Ordres bien fondés

Un *ordre strict* $\succ$ sur l'ensemble L est une relation binaire irréflexive ($x \not\succ x$) et transitive (si $x \succ y$ et $y \succ z$ alors $x \succ z$). Un ordre strict est *bien fondé* si et seulement si il n'existe aucune chaîne infinie décroissante $x_1 \succ x_2 \succ \ldots \succ x_k \succ \ldots$.

Par exemple, l'ordre $>$ sur les entiers naturels est bien fondé.

Une autre façon équivalente de caractériser un ordre strict bien fondé $\succ$ sur un ensemble non vide L est de dire qu'il existe un élément minimal pour $\succ$ dans L. En effet, si $\succ$ est bien fondé et X est non vide, soit $x_1 \in X$ quelconque. Si x_1 est minimal, alors c'est fini. Sinon, soit $x_2 \in X$ tel que $x_2 \succ x_1$. Si x_2 est minimal, c'est encore fini ; sinon soit $x_3 \in X$ tel que $x_3 \succ x_2$. Comme il n'existe aucune chaîne décroissante infini, en itérant le processus, on trouvera forcément un élément x_i, $i \geq 3$, tel que x_i est minimal dans X. Réciproquement, si tout ensemble non vide X contient un élément minimal x, en supposant qu'il existe une chaîne infinie décroissante $x_1 \succ x_2 \succ \ldots \succ x_k \succ \ldots$ d'éléments de X, on aboutit à une contradiction, car alors $x_i \not\succ x$ pour au moins un $i \in \mathbb{N}$. Donc $\succ$ est bien fondé.

Pour tout ordre strict $\succ$, notons $\succeq$ la relation définie par $x \succeq y$ si et seulement si $x \succ y$ ou $x = y$. Il s'agit d'une relation d'*ordre*, c'est-à-dire d'une relation binaire réflexive, transitive et antisymétrique. On dit que $\succ$ est *total* si et seulement si $\succeq$ est total, autrement dit si et seulement si pour tout x, y, $x \succ y$, ou $x = y$, ou $y \succ x$.

Théorème – A.0.1

> **(Zermelo)** *Pour tout ordre strict $\succ$ bien fondé sur L, il existe un ordre strict bien fondé* total $\overline{\succ}$ *sur L étendant $\succ$, c'est-à-dire tel que pour tout x, y, si $x \succ y$, alors $x \overline{\succ} y$.*

Dans le cas particulier où $\succ$ est l'ordre strict vide ($x \not\succ x$ pour tout $x \in L$), le théorème A.0.1 implique tout ensemble peut être muni d'un ordre total bien fondé. Ce dernier résultat est usuellement appelé le théorème de Zermelo.

Dans la suite, nous montrons quelques constructions permettant de combiner des ordres bien fondés à partir d'ordres bien fondés plus simple.

Ordre lexicographique

Définition – A.0.1

> *Si $\succ_1$ et $\succ_2$ sont deux ordres stricts, leur* produit lexicographique *$\succ_1 \times_{lex} \succ_2$ est défini par $(x_1, x_2) \succ_1 \times_{lex} \succ_2 (y_1, y_2)$ si et seulement si*
> - $x_1 \succ_1 y_1$,
> - *ou* $x_1 = y_1$ *et* $x_2 \succ_2 y_2$.

Si $\succ_1$ et $succ_2$ sont bien fondés, alors $\succ_1 \times_{lex} \succ_2$ est bien fondé. Soit en effet un ensemble A non vide de couples (x, y). Soit X l'ensemble $\{x \mid \exists y.(x, y) \in A\}$. Comme X est non vide et $\succ_1$ est bien fondé, il existe un élément minimal x_0 pour $\succ_1$ dans X. Soit Y l'ensemble $\{y \mid (x_0, y) \in A\}$. Par définition de x_0, l'ensemble Y est non vide, et comme $\succ_2$ est bien fondé, il existe un élément y_0 minimal pour $\succ_2$ dans Y. Par définition de x_0 et y_0, le couple (x_0, y_0) appartient à A. De plus, il est minimal car pour tout $(x, y) \in A$ différent de (x_0, y_0), soit $x \succ_1 x_0$ et alors $(x, y) \succ_1 \times_{lex} \succ_2 (x_0, y_0)$, soit $x = x_0$ et $y \succ_2 y_0$, et alors de nouveau $(x, y) \succ_1 \times_{lex} \succ_2 (x_0, y_0)$. Le couple (x_0, y_0) est donc minimal dans A pour l'ordre $\succ_1 \times_{lex} \succ_2$, qui est ainsi un ordre bien fondé.

On peut aussi montrer que si $\succ_1$ et $\succ_2$ sont deux ordres stricts totaux, alors $\succ_1 \times_{lex} \succ_2$ est aussi un ordre strict total. Soient (x_1, x_2) et (y_1, y_2) deux éléments de A. Si $x_1 \succ_1 y_1$, alors $(x_1, x_2) \succ_1 \times_{lex} \succ_2 (y_1, y_2)$; si $y_1 \succ_1 x_1$, alors $(y_1, y_2) \succ_1 \times_{lex} \succ_2 (x_1, x_2)$; si $x_1 = y_1$, soit $x_2 \succ_2 y_2$ et alors $(x_1, x_2) \succ_1 \times_{lex} \succ_2 (y_1, y_2)$, soit $x_2 = y_2$ et alors $(x_1, x_2) = (y_1, y_2)$, soit $y_2 \succ_2 x_2$ et alors $(y_1, y_2) \succ_1 \times_{lex} \succ_2 (x_1, x_2)$. Ainsi, $\succ_1 \times_{lex} \succ_2$ est un ordre strict total.

On peut généraliser et montrer qu'un produit lexicographique *fini* d'ordres stricts bien fondés est un ordre strict bien, et qu'un produit lexicographique fini d'ordres stricts totaux est un ordre strict total.

Ordre lexicographique sur suites finies triées

Soit L un ensemble muni d'un ordre strict bien fondé $\succ$. Soit $L^\bullet$ l'ensemble de toutes les suites finies *triées* $[x_1, \ldots, x_n]$, c'est-à-dire telles que $x_1 \preceq x_2 \preceq \ldots \preceq x_n$.

Définition – A.0.2

> *On définit l'ordre $\succ^\bullet$ sur $L^\bullet$ par :*
> - ⋆ $[x_1, \ldots, x_n] \succ^\bullet []$ *si* $n \geq 1$ *;*
> - ⋆ $[x_1, \ldots, x_n] \succ^\bullet [y_1, \ldots, y_m]$ *si* $n, m \geq 1$ *et :*
>   - *soit* $x_n \succ y_m$ *;*
>   - *soit* $x_n = y_m$ *et* $[x_1, \ldots, x_{n-1}] \succ^\bullet [y_1, \ldots, y_{m-1}]$.

La définition de $[x_1, \ldots, x_n] \succ^\bullet [y_1, \ldots, y_m]$ est bien formée, par récurrence sur $n + m$. On notera parfois $\overrightarrow{x}$ la suite $[x_1, \ldots, x_n]$; sa *longueur* est n, son *max* est x_n.

Si $\succ$ est un ordre strict bien fondé sur L, alors $\succ^\bullet$ est un ordre bien fondé sur $L^\bullet$. En effet, supposons que $\succ^\bullet$ n'est pas bien fondé. Il existe alors une suite infinie décroissante $\overrightarrow{x}^0 \succ^\bullet \ldots \succ^\bullet \overrightarrow{x}^k \succ^\bullet \ldots$ dans $L^\bullet$ telle que $\overrightarrow{x}^0$ n'est pas la suite vide $[]$ et telle que le max x de $\overrightarrow{x}^0$ soit minimal, et ce max étant fixé, telle que la longueur de $\overrightarrow{x}^0$ soit minimale. Par minimalité de x, tous les max des $\overrightarrow{x}^i$ ont le même max x. On en déduit que $\overrightarrow{y}^0 \succ^\bullet \ldots \succ^\bullet \overrightarrow{y}^k \succ^\bullet \ldots$ est une suite infinie décroissante, où $\overrightarrow{y}^i$ est la suite $\overrightarrow{x}^i$ de laquelle on a retiré le dernier élément (égal à x). Mais comme $\overrightarrow{x}^0$ est une suite triée, le max de $\overrightarrow{y}^0$ est inférieur ou égal à x, et sa longueur est strictement inférieure à celle de $\overrightarrow{x}^0$, contredisant la minimalité de x, et, pour x fixé, la minimalité de la longueur de $\overrightarrow{x}^0$.

On peut aussi montrer que si $\succ$ est total sur L, alors $\succ^\bullet$ est total sur $L^\bullet$. Soient $\overrightarrow{x}^n = [x_1, \ldots, x_n]$ et $\overrightarrow{y}^m = [y_1, \ldots, y_m]$ deux éléments de $L^\bullet$. Si $\overrightarrow{y}^m$ est la liste vide, alors $\overrightarrow{x}^n \succ^\bullet \overrightarrow{y}^m$, et sinon si $\overrightarrow{x}^n$ est la liste vide, alors $\overrightarrow{y}^m \succ^\bullet \overrightarrow{x}^n$. Dans le cas où aucune des deux suites n'est la suite vide, comme $\succ$ est un ordre total, soit $x_n \succ y_m$ et alors $\overrightarrow{x}^n \succ^\bullet \overrightarrow{y}^m$, soit $y_m \succ x_n$ et alors $y_m \succ^\bullet x_n$, soit $x_n = y_m$ et on itère ce raisonnement avec $\overrightarrow{x}^i = [x_1, \ldots, x_i]$, $i \leq n-1$, et $\overrightarrow{y}^j = [y_1, \ldots, y_{j-1}]$, $j \leq m-1$, qui se termine soit lorsqu'on peut conclure que $x_i \succ y_j$ ou $y_j \succ x_i$ pour $i, j \geq 1$, et donc que $\overrightarrow{x}^n \succ^\bullet \overrightarrow{y}^m$ ou $\overrightarrow{y}^m \succ^\bullet \overrightarrow{x}^n$, soit lorsque i ou j devient nul, c'est-à-dire lorsque l'on arrive à l'étape d'itération où au moins une des deux listes est vide ; si $i = 0$ et $j = 0$, alors $\overrightarrow{x}^n = \overrightarrow{y}$, sinon, si seul i est nul, alors $\overrightarrow{y}^j \succ^\bullet []$ et donc $\overrightarrow{y}^m \succ^\bullet \overrightarrow{x}^n$, si seul j est nul alors $\overrightarrow{x}^i \succ^\bullet []$ et donc $\overrightarrow{x}^n \succ^\bullet \overrightarrow{y}^m$.

Ainsi, $\succ^\bullet$ est total sur $L^\bullet$.

Ordre multi-ensemble

Un *multi-ensemble* fini m sur un ensemble A est une application de A vers $\mathbb{N}$ telle que $m(x) = 0$ sauf pour un nombre fini d'éléments x de A. L'ensemble $\{x \in A \mid m(x) \neq 0\}$ est le *support* de m. Intuitivement, un multi-ensemble est un ensemble fini où chaque élément peut apparaître une ou plusieurs fois. Par exemple, le multi-ensemble qui à x associe 1 et à y associe 3 est celui où x apparaît une fois et y trois fois, et on le notera usuellement $\{x, y, y, y\}$.

Si m et m' sont deux multi-ensembles, leur *union*, notée $m \uplus m'$, est la fonction qui à x associe $m(x) + m'(x)$.

Définition – A.0.3

> *Si $\succ$ est un ordre strict sur A, son* extension multi-ensemble $\succ_{mul}$ *est le plus petit ordre strict tel que $m \uplus \{x\} \succ_{mul} m \uplus \{x_1, \ldots, x_n\}$ pour tout x et tout multi-ensemble d'éléments $x_1, \ldots, x_n$ tels que $x \succ x_1, \ldots, x_n$.*

Si $\succ$ est un ordre bien fondé, alors son extension multi-ensemble $\succ_{mul}$ est bien fondé. Par le théorème A.0.1, il existe une extension totale bien fondée $\overline{\succ}$ de $\succ$. Alors, $\overline{\succ}_{mul}$ est une extension de $\succ_{mul}$, et il suffit de montrer que $\overline{\succ}_{mul}$ est bien fondé pour en déduire que $\succ_{mul}$ est bien fondé. Donc, sans perte de généralité, nous pouvons supposer que $\succ$ est un ordre strict bien fondé total.

Comme $\succ$ est total, on peut trier les éléments d'un multi-ensemble par ordre croissant : pour tout multi-ensemble m, de support l'ensemble fini X, soit $x_1 \succ \ldots \succ x_n$ les éléments de X, posons $\widehat{m}$ la suite :

$$\underbrace{x_1 \ldots, x_1}_{m(x_1)}, \underbrace{x_2 \ldots, x_2}_{m(x_1)}, \ldots, \underbrace{x_n \ldots, x_n}_{m(x_n)}$$

L'application qui à tout multi-ensemble m associe la suite finie triée $\widehat{m}$ est clairement une bijection.

De plus, on vérifie aisément que $m \succ_{mul} m'$ si et seulement si $\widehat{m} \succ^{\bullet} \widehat{m'}$. Il s'en suit que si $\succ$ est bien fondé, alors $\succ_{mul}$ est bien fondé.

Finalement, si $\succ$ est total, comme $\succ^{\bullet}$ est total sur $L^{\bullet}$, $\succ_{mul}$ est lui aussi total sur l'ensemble des multi-ensembles d'éléments de L.

Annexe B

Signature sur les termes abstraits

Nous avons présenté en section 4.1.1 une version simplifiée de la signature sur les termes abstraits. Nous allons présenté plus en détails cette signature, donnée en figure B.1. Rappelons

```
module type TERM =
sig
     type term
     type subst
     type ordering = Greater | Equal | NotGE
     val app-sub : subst -> term -> term
     val comp-sub : subst -> subst -> subst
     val unify : (term * term) list -> subst list
     val match-term : (term * term) list -> subst list
     val build-new-sub : term list -> subst
     val new-var-list : int -> term list
     val app-ZnZ-sub : subst -> int -> term -> term
     val is-ZnZ-epsilon-bloc : int -> term list -> bool
     val there-is-no-f-libre : term list -> bool
     val cond-acm : bool
     val cond-res : term list -> bool
     val order : term * term -> ordering
     val order-litt : (string * term list) -> (string * term list)
            -> ordering
     val special-pred : (string * term list) list
            -> (string * term list) list -> string
     val string-of-terms : term list -> string
end;;
```

Fig. B.1: Signature définissant les termes abstraits (version complète)

que les types `term` et `subst` sont des types abstraits de Caml, que le type `ordering` est un type somme de Caml défini avec les constructeurs constants `Greater` (signifiant qu'un terme est plus grand qu'un autre), `Equal` (signifiant que deux termes sont égaux) et `NotGE` (signifiant qu'un terme n'est ni plus grand ni égal à un autre).

Le type des fonctions `app-sub` et `comp-sub` sont définis de manière cohérente avec l'application *Subst* définissant les termes symboliques (section 3.1.1).

Rappelons aussi que `unify` et `match-term` prennent en entrée une liste de couples de termes à unifier simultanément et renvoient une liste de substitutions représentant le csu.

L'application `build-new-sub` sert au renommage des clauses : à partir d'une liste de termes présents dans une clause, elle construit une substitution qui à chaque variable associe une variable fraîche. L'application `new-var-list` renvoie une liste de variables fraîches de longueur spécifiée par la donnée de type `int`. Cette application est utile à la construction des clauses modélisant le protocole IKA.1 (section 6.1.2).

Les trois fonctions `app-ZnZ-sub`, `is-ZnZ-epsilon-bloc` et `there-is-no-f-libre` sont spécifiques au module des termes associatifs et commutatifs. Étant donnée une substitution (qui associe à chaque variable un élément de $\mathbb{Z}/n\mathbb{Z}$ sous la forme d'une constante), un entier n et un terme, la fonction `app-ZnZ-sub` sert à effectuer l'instantiation des clauses sur $\mathbb{Z}/n\mathbb{Z}$. La fonction `is-ZnZ-epsilon-bloc` sert à tester qu'une clause est une instantiation sur $\mathbb{Z}/n\mathbb{Z}$ d'une disjonction d'ϵ-blocs. Ce test permettra, le cas échéant, de générer de nouvelles clauses. La fonction `there-is-no-f-libre` sert à tester qu'une clause est instantiable. Pour les modules différents de celui des termes associatifs et commutatifs, ces fonctions pourront être définies de manière quelconque (on peut par exemple mettre la valeur `false` pour celles qui renvoient un booléen), l'emploi de ces fonctions étant limité dans le module de résolution par la valeur `cond-acm` (elle est à `true` dans le module des termes associatifs et commutatifs, et à `false` dans les autres modules). La fonction `cond-res` représente la condition (i) de la résolution binaire ordonnée avec sélection (figure 5.6) pour les clauses sur des termes associatifs et commutatifs. Dans le cas où il n'y a pas de condition supplémentaire à la règle de résolution ordonnée avec sélection, il suffit de poser `let cond-res = true`.

Les fonction `order` et `order-litt` servent à définir un ordre sur les termes et un sur les littéraux respectivement (le deuxième étant en général une extension du premier). L'ordre sur les littéraux est défini dans les modules de termes, et non dans le module de résolution où il serait défini comme une extension générique d'un ordre sur les termes. Ainsi, comme nous l'avons laissé entendre en section 4.1.1, une plus grande liberté est laissée pour le choix de l'ordre.

Enfin, la fonction `special-pred` sert à fabriquer des symboles de prédicats spéciaux pour le splitting sans splitting (section 3.5.2) ; rappelons qu'étant données deux clauses C et $C\varrho$, où ϱ est un renommage, on doit avoir $\ulcorner C \urcorner = \ulcorner C\varrho \urcorner$, ce qui explique qu'on ait besoin de définir cette fonction dans les modules de termes. Pour finir, la fonction `string-of-terms` est simplement une fonction d'affichage (qui en pratique fait appel à un parseur).

Table des figures

Bibliographie

[AC02] R. Amadio and W. Charatonik. On name generation and set-based analysis in the dolev-yao model. In *Proceedings of CONCUR 02*. Springer-Verlag, 2002.

[AG97] Martin Abadi and Andrew Gordon. A calculus for cryptographic protocols – the spi-calculus. In *Proceedings of the 4th conference on computer and communication security*, Zurich, Switzerland, april 1-4 1997.

[AL00] R. Amadio and D. Lugiez. On the reachability problem in cryptographic protocols. volume 1877 of *Lecture Notes in Computer Science*, pages 380–394, 2000.

[ARS78] L. Adleman, R. Rivest, and A. Shamir. A method of obtaining digital signatures and public key cryptosystem. In *Communication of the ACM*, volume 21, pages 120–126, 1978.

[BAN89] M. Burrows, M. Abadi, and R. Nedham. A logic of authentication. In *Proceedings of the Royal Society of London*, volume 426, pages 233–271, 1989.

[Bar84] H. Barendregt. *The Lambda-Calculus*, volume 103. Elseiver Science Publishing Compagny, Amsterdam, 1984.

[BG01a] Leo Bachmair and Harald Ganzinger. *Resolution theorem proving*, volume 1 of *Handbook of automated reasonning*, chapter 2, pages 19–99. North-Holland, 2001.

[BG01b] Leo Bachmair and Harald Ganzinger. *Resolution theorem proving*, chapter 2, pages 41–46. Handbook of automated reasonning. J. Alan Robinson and Andrei Voronkov, 2001.

[Bla01] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings of the 14th IEEE Computer Security Fundation Workshop*. IEEE Computer Society, 2001.

[BN02] Johannes Borgström and Uwe Nestmann. On bisimulations for the spi calculus. In *AMAST'2002*, volume 2422 of *LNCS*, pages 287–303, Saint Gilles les Bains, Reunion Island, France, september 9-13 2002.

[Bol96] Dominique Bolignano. An approach to the formal verification of cryptographic protocols. In *Proceedings of the 3rd conference on computer and communication security*, march 1996.

[Bol97a] Dominique Bolignano. Towards a mechanization of cryptogaphic protocol verification. In *Proceedings of the 9th international conference on computer aided verification (CAV'97),Haifa, Israel, June 1997*, volume 1254 of *Lecture notes in computer science*, pages 131–142. Springer, 1997.

[Bol97b] Dominique Bolignano. Towards the formal verification of electronic commerce protocols. In *10th IEEE Computer Security Fondations Workshop*, 1997.

[BP03] B. Blanchet and A. Podelski. Verification of cryptographic protocols : Tagging enforces termination. In Springer Verlag, editor, *Andrew Gordon, editor, Foundations of Software Science and Computation Structures (FoSSaCS'03)*, volume 2620 of *Lecture Notes on Computer Science*, pages 136–152, Warsaw, Poland, April 2003.

[CDJK99] H. Comon, M. Dincbas, J.P. Jouannaud, and C. Kirchner. A methodological view of constraint solving. *Constraints*, 4(4) :337–361, 1999.

[CJ97] H. Comon and J.-P. Jouannaud. Les termes en logique et en programmation (version préliminaire). Notes de Cours, 1997. `http ://www.lsv.ens-cachan.fr/~comon/Sommaire.html`.

[CKR+03] Y. Chevalier, R. Küsters, M. Rusinowitch, M. Turuani, and L. Vigneron. Extending the dolev-yao intruder for analyzing an unbounded number of sessions. In *Computer Science Logic*, Vienna, Austria, August 2003.

[CKRT03] Y. Chevalier, R. Küsters, M. Rusinowitch, and M. Turuani. An np decision procedure for protocol insecurity with xor. In *Proceedings of LICS'2003*, 2003.

[CLC03a] H. Comon-Lundh and V. Cortier. New decidability results for fragments of first-order logic and application to cryptographic protocols. In *Proceedings of RTA'2003*, 2003.

[CLC03b] H. Comon-Lundh and V. Cortier. Security properties : two agents are sufficient. In *Proceedings of European Symposium on Programming*, 2003.

[CLS03] H. Comon-Lundh and V. Shmatikov. Intruder deductions, constraint solving and insecurity decision in presence of exclusive or. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science (LICS '03)*, pages 271–280, 2003.

[CMP00] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, 2000.

[com79] *Computers and intractability : a guide to the theory of NP-completeness*. W. H. Freeman and Company, 1979.

[DH76] Whit Diffie and Martin Hellman. New directions in cryptography. In *IEEE Transactions on Information Theory*, volume 22, chapter 6, pages 644–654. November 1976.

[dN94] H. de Nivelle. Resolution games and non-liftable resolution orderings. In *CSL'94*, 1994.

[DS81] D.E. Denning and G.M. Sacco. Timestamps in key distribution protocoles. In *Communication ACM*, volume 24, pages 533–536, August 1981.

[DY83] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE transactions on information theory*, IT-29(2) :198–208, march 1983.

[enc00] *Cryptologie*. Encyclopædia Universalis S.A., 2000.

[FLHT01] C. Fermüller, A. Leitsch, U. Hustadt, and T. Tammet. *Resolution Decision Procedures*, volume 2 of *Handbook of automated reasonning*, chapter 25, pages 1791–1849. North-Holland, 2001.

[FLTZ93] C. Fermüller, A. Leitsch, T. Tammet, and N. Zamov. *Resolution methods for the decision procedure*, volume 679 of *LNCS*. Springer Verlag, 1993.

[FSVY91] Thom Frühwirth, Ehud Shapiro, Moshe Y. Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In *6th IEEE Symposium on Logic in Computer Sciences*, pages 300–309, Amsterdam, July 1991.

[GK99] T. Genêt and F. Klay. Rewriting for cryptographic protocol verification (extended version). Technical report, CNET-France Telecom, `http ://www.loria.fr/ genet/Publications/GenetKlay-RR99.ps`, 1999.

[GL00] Jean Goubault-Larrecq. A method for automatic cryptographic protocol verification (extended abstract). In *Proc. 15 IPDPS 2000 Workshops*, volume 1800 of *Lecture Notes in Computer Science*, pages 977–984, Cancun, Mexico, May 2000. Springer.

[GLM97] Jean Goubault-Larrecq and Ian Mackie. *Proof Theory and Automated Deduction*, volume 6 of *Applied Logic Series*, chapter 7, pages 235–240. Kluwer Academic Publishers, 1997.

[GLV02] J. Goubault-Larrecq and K. N. Verma. Alternating two-way ac-tree automata. Technical Report LSV-02-11, Lab. Specification and Verification, ENS de Cachan, Cachan, France, September 2002. 21 pages.

[Hil03] T. Hillenbrand. CITIUS ALTIUS FORTIUS : lessons learned from the theorem prover WALDMEISTER (invited paper). In I. Dahn and L. Vigneron, editors, *Proceedings of the 4th International Workshop on First-Order Theorem Proving*, volume 86.1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2003.

[Hue73] G. Huet. A complete mechanization of type theory. In *Proceedings of the Third International Joint Conference on Artificial Intelligence*, 1973.

[Ibe97] O. Ibens. The setheo system, 1997.

[JK90] J.P. Jouannaud and C. Kirchner. Solving equations in abstract algebras : A rule-based survey of unification. Technical Report 561, LRI, CNRS UA 410, march 1990.

[Kal01] J. A. Kalman. *Automated Reasoning with Otter*. Rinton Press, 2001.

[Ker83] August Kerckhoffs. Cryptologie militaire. *Journal des sciences militaires*, 1883.

[Kha01] N. El Khabi. Automatic verification of confidentiality properties of cryptographic programs. In *Networking and Information Systems*, 2001.

[Lal90] R. Lalement. *Logique réduction résolution*. études et recherches en informatique. Masson, 1990.

[LBP03] Y. Lakhnech L. Bozga and M. Périn. Abstract interpretation for secrecy using patterns. In *Proceedings of TACAS*, LNCS, 2003.

[Low95] G. Lowe. An attack on the needham-shroeder public-key protocol. *Information Processing Letters*, 56(3) :131–133, 1995.

[Low96] G. Lowe. Breaking and fixing the needham-shroeder public-key protocol using fdr. volume 1055 of *Lecture Notes in Computer Science*, pages 147–166, 1996.

[Low97] G. Lowe. A hierarchie of authentication specifications. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, 1997.

[MD02] Miller and Denker. Capsl and mucapsl. *Journal of Telecommunications and Information Technology*, april 2002.

[Mon99] D. Monniaux. Abstracting cryptographic protocols with tree automata. In *6th International Static Analysis Symposium (SAS'99)*, LNCS 1694, pages pages 149–163. Springer-Verlag, 1999.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus for mobile processes, part 1 and 2. In *Information and Computation*, pages 1–40 & 41–77, 1992.

[NDS99] J. Mitchell N. Durgin, P. Lincoln and A. Scedrov. Undecidability of bounded security protocols. In *Proceedings of the Workshop of formal methods in security protocols*, 1999.

[NS78] R.M. Needham and M.D. Schroeder. Using encryption for authentication in a large networks of computers. In *Communications of the ACM*, volume 21, pages 993–999, 1978.

[Pau98] Larry Paulson. The inductive approach to verify cryptographic protocols. *Journal of Computer Security*, 6, 1998.

[Pel86] F.J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Automated Reasonning*, 2 :191–216, 1986.

[PQ01] Olivier Pereira and Jean-Jacques Quisquater. A security analysis of the cliques protocols suites. In *Proceedings of the 14th IEEE Computer Security Fundation Workshop*. IEEE Computer Society, 2001.

[Rob65] J. A. Robinson. A machine oriented logic based on the resolution principle. *ACM*, 12 :23–41, 1965.

[RSV01] I. V. Ramakrishnan, R. Sekar, and A. Vonronkov. *Term indexing*, volume 2 of *Handbook of automated reasonning*, chapter 26, pages 1853–1954. Robinson and Voronkov, 2001.

[RT01] M. Rusinowitch and M. Turiani. Protocol insecurity with infinite number of sessions is np-complete. In *14th IEEE Computer Security Fundation Workshop*, pages 174–190. IEEE Computer Society, 2001.

[RV01] A. Riazanov and A. Voronkov. Vampire 1.1 (system description). In *IJCAR*, volume 2083 of *LNAI*, pages 376–380, 2001.

[Sch94] B. Schneier. *Applied Cryptography : Protocols, Algorithm and Source code in C.* John Wiley and Sons, 1994.

[Ste98] J. Stern. *La science du secret.* Odile Jacob, 1998.

[Sti87] M. E. Stickel. A prolog technology theorem prover : Implementation by an extended prolog compiler. In *Proceedings of the 8th Conference on Automated Deduction*, 1987.

[STW96] Michael Steiner, Gene Tsudik, and Michael Waidner. Diffie-hellman key distribution extended to group communication. In *Proceedings of the 3rd ACM Conference on Computer and Communications Security*, New Delhi, India, March 14-16 1996.

[Ver03] K. N. Verma. Two-way equational tree automata for ac-like theories : Decidability and closure properties. In Springer, editor, *Proceedings of the 14th Int. Conf. Rewriting Techniques and Applications (RTA'2003)*, volume 2706 of *Lecture Notes in Computer Science*, pages 180–196, 2003.

[Wei99a] C. Weidenbach. Spass : Combining superposition, sorts and splitting. Handbook of Automated Reasonning, Elsevier, To appear, 1999.

[Wei99b] C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In H. Gantzinger, editor, *Proceedings of the 16th Conference on Automated Deduction*, volume 1632 of *Lecture Notes on Computer Science*, pages 314–328, 1999.