Certification des raisonnements formels portant sur des systèmes d'information critiques

Amira Henaien

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Certification des raisonnements formels portant sur des systèmes d'information critiques

THÈSE

présentée et soutenue publiquement le 11 mars 2015

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Amira HENAIEN

Composition du jury

Rapporteurs :	Évelyne CONTEJEAN Olga KOUCHNARENKO
Examinateurs :	Stephan MERZ Xavier URBAIN
Encadrants :	Maurice MARGENSTERN Sorin STRATULAT

Laboratoire d'Informatique Théorique et Appliquée — EA 3097

Mis en page avec la classe thesul.

Remerciements

C'est avec un grand plaisir que j'adresse mes sincères remerciements à toutes les personnes qui ont contribué de près ou de loin à la réalisation de cette thèse et qui m'ont apporté leur aide tout au long de son élaboration.

J'adresse tout d'abord mes remerciements à mes directeurs de thèse Maurice MARGENSTERN, Professeur de l'Université de Lorraine, et Sorin STRATULAT, Maître de conférences de l'Université de Lorraine, pour leur encadrement, leurs conseils et leurs remarques.

Je tiens à témoigner toute ma reconnaissance à Olivier PERRIN, Professeur de l'Université de Lorraine et Responsable du DFD informatique à l'école doctorale IAEM, et Stephan MERZ, Chercheur à l'INRIA Nancy, pour leur suivi de ces travaux et leurs conseils précieux durant ma quatrième année. Ainsi, je tiens à remercier Stephan MERZ pour ses lectures de mon mémoire et ses remarques sur son contenu, ainsi que pour son acceptation de faire partie du jury de ma soutenance de thèse.

Je voudrais exprimer ma profonde gratitude à Évelyne CONTEJEAN, Chercheur CNRS, et à Olga KOUCHNARENKO, Professeur de L'université de Franche-Comté, pour avoir accepté d'être rapporteurs de ma thèse et membres du jury et pour le temps consacré à la lecture et la compréhension de ce document. Leurs commentaires et leurs questions ont contribué à améliorer de manière significative le document.

Je remercie vivement Xavier URBAIN, Maître de conférences d'ENSIIE, qui m'a fait l'honneur d'accepter de faire partie du jury de ma soutenance de thèse.

Je tiens à remercier également tous les membres du laboratoire de recherche LITA et la direction de l'école doctorale IAEM.

Enfin, merci de tout coeur à mes parents, à mes frères et à mes amis pour leurs encouragements et leur soutien amical.

Je dédie ce modeste travail à mes parents qui ont tout déployé et n'ont lésiné sur rien pour le succès dans mes études, ainsi qu'à mes frères Amir, Khaled et Abbes, mes proches et mes amis qui m'ont toujours encouragé.

Table des matières

Table des figures

Résumé

Les preuves par récurrence sont parfaitement adaptées au raisonnement sur des structures de données non-bornées, comme par exemple les entiers et les listes, ou, de manière plus générale, sur des ensembles d'éléments non-vides munis d'ordres noethériens. Leur domaine d'application est très vaste, une utilité particulière portant sur la validation des propriétés d'applications industrielles dans des domaines critiques tels que les télécommunications et les cartes à puces. Le principe de récurrence noethérienne est à la base d'un ensemble de techniques de preuve par récurrence modernes, dont celles basées sur la récurrence implicite.

Dans cette thèse, nous nous intéresserons à l'intégration du raisonnement par récurrence implicite tel qu'il est implémenté dans le démonstrateur Spike en utilisant l'environnement de preuve certifié Coq. Basé sur la récurrence implicite, Spike est capable de raisonner automatiquement sur des théories conditionnelles de premier ordre. L'implémentation de Spike n'est pas encore certifiée, même si les fondements théoriques sous-jacents ont été approuvés à plusieurs reprises par la communauté scientifique. Une alternative convenable serait de certifier seulement les preuves générées par Spike. Dans ce cas, le processus de certification doit être automatique car les scripts de preuves de Spike sont souvent longs. Des travaux précédents ont montré la possibilité de certifier automatiquement des preuves par récurrence implicite générées par Spike à l'aide de l'environnement certifié de l'assistant de preuve Coq. Nous proposerons des nouvelles tactiques Coq qui seront capables de prouver automatiquement des théorèmes par récurrence implicite. Deux approches seront étudiées.

La première approche consiste à utiliser Spike comme un outil externe. Elle est limitée au traitement des spécifications Coq qui peuvent être traduites dans des spécifications conditionnelles, ainsi qu'à des théorèmes convertibles dans des équations conditionnelles. Les traces de preuves générées par Spike sont ensuite traduites dans des scripts Coq qui sont finalement validés par son noyau. Une autre limitation est due à la traduction des applications d'un sous-ensemble de règles d'inférence de Spike.

La deuxième approche est l'utilisation des strategies à la Spike pour construire automatiquement des preuves par récurrence implicite dans Coq. Cette approche se base sur des tactiques Coq qui simulent des règles d'inférence de Spike pour générer de nouveaux sous-buts. Par rapport à la première approche, ces tactiques peuvent utiliser des techniques de raisonnement de Coq qui ne sont pas présentes dans Spike et ouvre la possibilité de mélanger des étapes de preuves automatiques et manuelles.

Ces deux approches ont été mises en oeuvre et testées sur différents exemples dont des lemmes utilisés dans la preuve de validité de l'algorithme de conformité du protocole de télécommunication ABR.

Mots-clés: raisonnement formel, démonstration automatique, preuves par récurrence, certification des preuves formelles, applications critiques.

Abstract

Proofs by induction are perfectly adequate to reasoning on unbounded data structures, for example naturals, lists and more generally on non-empty sets of elements provided with noetherian orders. They are largely used on different fields, particularly for the validation of properties of industrial applications in critical areas such as telecommunications and smart cards. The principle of noetherian induction is the basis of a set of modern techniques of proof by induction, including those based on implicit induction.

In this thesis, we will focus on the integration of implicit induction reasoning like it is implemented by spike using the certified proof environnement Coq. Spike is an automatic theorem prover based on implicit induction that is capable of reasoning on conditional first-order theories. The implementation of Spike is not yet certified, even if the underlying theoretical foundations have been approved repeatedly by the scientific community. A suitable alternative is to certify only the proofs produced by Spike. In this case, the certification process must be automatic because scripts of Spike's proofs are often long. Previous work has shown the possibility of certifying automatically some proofs by implicit induction generated by Spike using the certified environment provided by the Coq proof-assistant. We will propose new Coq tactics that are able to prove automatically theorems by implicit induction. Two approaches will be studied.

The first approach consists on using Spike as an external tool. It is limited to process Coq specifications which can be translated in conditional specifications, as well as theorems convertible in conditional equations. Proofs generated by Spike are then translated into Coq scripts finally validated by its kernel. Another limitation is due to the translation of the application of a subset of the Spike inference rules.

The second approche is to use strategies à la Spike to automatically build implicit induction proofs in Coq. This approach consists on creating tactics that perform like Spike inference rules to generate new subgoals in Coq. Comparing to the first approach, these tactics permit the use of Coq reasoning techniques which are not present in Spike and opens the possibility of mixing automatic and manual proof steps.

Both approaches have been implemented and tested on several examples including lemmas used in the proof of validity of the conformity algorithm for the ABR telecommunications protocol.

Keywords: formal reasoning, automatic proving, proof by induction, certification of formal proofs, critical applications.

Introduction

Depuis son ouverture sur les différents secteurs de la vie courante, l'informatique nous a permis de rouler des véhicules sans conducteur, de déplacer des robots sur Mars, d'assister des séances de chirurgie par des robots, de gérer des bases des données de banques, etc. En effet, la liste des applications assistées par ordinateur est devenue innombrable. Des applications critiques, telles que nous avons cité, mettent en jeu des ressources humaines, matérielles et financières importantes. La vérification que ces systèmes informatiques accomplissent leurs rôles souhaités en utilisant des méthodes basées sur des notions mathématiques et logiques, appelées méthodes formelles [1], est fortement recommandée. Ces méthodes permettent de définir un système, de décrire ses propriétés et d'établir leurs preuves de validation dans le même environnement formel. Ces preuves se construisent formellement et conformément à la description du système.

Ancien sujet mathématique et philosophique, la théorie de la preuve joue de nos jours un rôle important en informatique, notamment pour le développement des logiciels critiques. Elle permet d'étudier les preuves en tant qu'objets formels. Jadis, on se contentait des preuves développées par l'homme. De nos jours, ces preuves sont devenues grandes et portent sur des systèmes de taille industrielle. L'homme est devenu incapable de les certifier et est remplacé par l'ordinateur.

Nous trouvons aujourd'hui une large variété d'assistants de preuve et de démonstrateurs des théorèmes. Souvent, ces outils doivent prendre en considération les différentes structures de données et les fonctions qui sont largement utilisées en programmation, dont celles qui sont non-bornées et récursivement définies.

1 Raisonnement par récurrence

Le raisonnement par récurrence est parmi les raisonnements mathématiques les plus adéquats pour le traitement des structures des données ou des fonctions récursivement définies. Les premières traces de ce raisonnement remontent à la philosophie grecque :

> « De l'induction, son importance égale à celle du syllogisme [2][...] ; l'induction est plus évidente que le syllogisme » Le premier maître [3] [Saint-Hilaire1843].

Depuis, il est utilisé par plusieurs mathématiciens et philosophes dans le monde oriental et occidental. Des mathématiciens comme Al-Karkhi [4], Pascal [5] et Fermat [6] ont utilisé ce raisonnement pour résoudre des problèmes mathématiques [Al-Karkhi et Woepcke1853, Pascal et Lafuma1963, Brassinne1853]. D'autres savants l'ont considéré parmi les ressources de développement des mathématiques et de toutes les sciences.

1. Le terme « formel » désigne toute utilisation des notions mathématiques et logiques.
2. Déduction
3. Aristote [384 av.J.C - 322 av.J.C]
4. Al-Karkhi [mort vers 1020]
5. Pascal [19 juin 1623 - 19 août 1662]
6. Fermat [1601 - 12 janvier 1665]

> « Si les mathématiques n'en avaient pas d'autre elles seraient donc tout de suite arrêtées dans leur développement ; mais elles ont de nouveau recours au même procédé, c'est-à-dire au raisonnement par récurrence et elles peuvent continuer leur marche en avant.
> ...
> C'est donc bien là le raisonnement mathématique par excellence et il nous faut l'examiner de plus près. » Poincaré [7] [Poincaré1898]

Pour une longue période, ce raisonnement était plus reconnu en arithmétique sous la forme d'un axiome de l'ensemble des axiomes de Peano [8] qui définissent les entiers naturels [Peano1908]. Nous pouvons le présenter sous la forme suivante :

> « Soit $\mathcal{P}$ une propriété. Alors $\mathcal{P}$ est valide pour tous les entiers naturels, si $\mathcal{P}$ est valide pour le premier entier naturel 0 et pour tout entier naturel $n+1$ en supposant qu'elle est valide pour n. Le principe peut être représenté par la règle suivante :
>
> $$(\texttt{Principe de Peano})\ \frac{\mathcal{P}(0),(\forall n{\geq}0,\mathcal{P}(n){\Rightarrow}\mathcal{P}(n{+}1))}{\forall n,\mathcal{P}(n)}\ \text{»}$$

Ce raisonnement qui à la base relève du domaine des mathématiques, est aujourd'hui utilisé dans de différents domaines, dont l'informatique. Dés qu'on a commencé à apprendre à la machine d'assister des preuves des théorèmes, on a réalisé que la récurrence est parmi les outils mathématiques indispensables. Nous citons parmi les premiers essais l'intégration de la méthode de la récurrence récursive [9] utilisée pour montrer l'équivalence entre des fonctions récursivement définies dans Lisp [McCarthy1959]. Plusieurs informaticiens se sont mis d'accord sur la nécessité de développer des raisonnements mathématiques théoriques pour l'informatique dont la récurrence a joué un rôle principal. Ce raisonnement est devenu largement utilisé et essentiel en théorie de la preuve. Au début, il était souvent intégré dans les assistants de preuve sous sa forme structurelle [McCarthy et Painter1967, Burstall1969], jusqu'à ce qu'on a commencé à le renforcer par d'autres techniques de preuve comme la preuve par cohérence, la réécriture et des procédures de déduction [Musser1980, Knuth et Bendix1983]. À partir de ce moment, la récurrence implicite commence à apparaître dans le monde de la théorie de la preuve et à avoir un succès sur des applications concrètes [Rusinowitch *et al.*2000, Barthe et Stratulat2003].

1.1 Raisonnement par récurrence noethérienne

Fréquemment utilisé dans la théorie de la preuve, le principe général de raisonnement par récurrence est le suivant : une proposition $\mathcal{P}$ définie sur un ensemble d'éléments $\mathcal{E}$ est vraie s'il existe une démonstration que cette proposition est vraie pour tout élément x de $\mathcal{E}$ en supposant qu'elle l'est pour tous les éléments plus petits que x. On dit que la proposition est montrée vraie par récurrence. Ce principe ne peut être appliqué que sur des ensembles d'éléments noethériens munis d'`ordres de récurrence` qu'on note par $<$. D'une façon formelle, on définit le principe général de récurrence noethérienne comme suit :

$$(RN) : (\forall y \in \mathcal{E}, (\forall\ z \in \mathcal{E},\ z < y \Rightarrow \mathcal{P}(z)) \Rightarrow \mathcal{P}(y)) \Rightarrow \forall x \in \mathcal{E}, \mathcal{P}(x).$$

7. Henri Poincaré [29 avril 1854 - 17 juillet 1912]
8. Giuseppe Peano [27 août 1858 - 20 avril 1932].
9. *Recursion Induction*

L'application du principe RN sur l'ensemble $\mathcal{E}$ distingue les instances de $\mathcal{P}$ qui jouent le rôle des `conclusions de récurrence`, telles que $(P(y), \forall y \in \mathcal{E})$, et celles qui sont des `hypothèses de récurrence`, telles que $P(z), \forall z \in \mathcal{E}, z < y$. La relation établie entre toutes les conclusions et leurs hypothèses de récurrence est nommée un `schéma de récurrence`. Ce schéma contient des `cas de base` qui n'ont pas d'hypothèses de récurrence, et des `cas de récurrence` qui en ont. Le schéma de récurrence se définit à partir du contexte général de la preuve.

La preuve par raisonnement par récurrence est considérée comme `fainéante`, ou `non-gourmande`, si les hypothèses de récurrence sont définies au moment de leurs applications. Soient les deux propositions P_1 et P_2. La preuve par récurrence de P_1 est `mutuelle` avec celle de la preuve de P_2, si la preuve de P_1 nécessite P_2 ou éventuellement une de ses instances, et réciproquement, la preuve de P_2 nécessite P_1 ou une de ses instances.

Le raisonnement par récurrence souvent implémente la récurrence noethérienne et possède une variété d'instances qui dépendent de la nature des éléments de $\mathcal{E}$. Des démonstrateurs de théorèmes qui sont basés sur ces instances sont : Coq [Coq development team], ISABELLE [The ISABELLE development team], ACL2 [Boyer et Moore1988a], PVS [Owre *et al.*1993], Spike [Bouhoula *et al.*1995, Stratulat2001] etc. Nous nous intéressons aux instances de raisonnement par récurrence noethérienne utilisées dans la logique de premier ordre.

1.2 Correction des preuves par récurrence noethérienne

La correction de la preuve par récurrence noethérienne dépend fortement de l'ordre de récurrence noethérienne. En effet, la preuve par récurrence d'une proposition P ne peut être considérée comme correcte que si l'ordre $<$ est noethérien, i.e. garantit qu'il n'existe aucune séquence d'éléments de $\mathcal{E}$ infiniment décroissante. La vérification de la correction de ces preuves consiste à la vérification que toute hypothèse de récurrence introduite est plus petite que sa conclusion.

1.3 Raisonnement par récurrence dans la logique du premier ordre

En logique de premier ordre, le raisonnement par récurrence noethérienne peut avoir l'une des deux formes suivantes : raisonnement par récurrence basée soit sur les termes, soit sur les formules [Stratulat2012].

A. Raisonnement par récurrence basée sur les termes

Ce raisonnement se base sur des schémas de récurrence dont les hypothèses et les conclusions de récurrence sont des instances d'une même formule. Dans ce cas, l'ordre de récurrence se définit sur les termes au niveau du schéma de récurrence. Une preuve par récurrence basée sur les termes peut avoir plusieurs schémas de récurrence dont chacun possède son propre ordre. Cette instance du raisonnement par récurrence est relativement simple à intégrer dans les démonstrateurs de théorèmes parce qu'elle définit l'ordre de récurrence localement au niveau des règles d'inférence. Par contre, le raisonnement est par récurrence non fainéante et non mutuelle.

Exemple : preuves par récurrence structurelle Elles se basent sur le calcul des schémas de récurrence à partir des définitions des éléments de $\mathcal{E}$. Cette instance de raisonnement par récurrence possède les mêmes avantages et désavantages que le raisonnement par récurrence basée sur les termes : elle est implémenté en une seule règle, se base sur un ordre de récurrence local, et est gourmande et non-mutuelle. Elle est devenue un outil indispensable dans les assistants à la preuve. L'une de ces formes la plus reconnue est celle définie par le `principe de Peano`.

B. Raisonnement par récurrence basée sur les formules

Ce raisonnement s'applique sur un ensemble de formules dont n'importe laquelle «peut» jouer le rôle d'une hypothèse de récurrence. La condition primordiale pour qu'une formule soit une hypothèse de récurrence est qu'elle soit plus petite que la formule sur laquelle elle s'applique. Cette condition garantit la correction du raisonnement et exige la définition de l'ordre d'une façon globale sur l'ensemble des formules. Contrairement au raisonnement par récurrence basée sur les termes, celui basé sur les formules effectue naturellement le raisonnement par récurrence mutuelle et fainéante. Par contre, la définition de l'ordre de récurrence emporte des contraintes supplémentaires sur son implémentation.

Exemple : preuves par récurrence implicite Dans cette thèse, nous nous intéressons aux preuves par récurrence implicite qui sont des preuves par récurrence basée sur l'ensemble des formules d'une preuve. Cet ensemble contient des formules initiales, dont nous souhaitons prouver leur validité, et d'autres générées au cours de la preuve. Cette instance de raisonnement par récurrence possède les mêmes avantages et désavantages que le raisonnement par récurrence basée sur les formules : elle est fainéante, mutuelle et se base sur un ordre de récurrence global. Il serait pertinent de prendre en considération l'ordre pendant la construction de la preuve ; en d'autres termes, de construire la preuve d'une façon réductible. Ainsi, en une étape quelconque de la preuve, les nouvelles formules générées doivent être plus petites que la formule courante. Sur le plan pratique, cette proposition imposera des contraintes d'ordre supplémentaires à respecter par la procédure générale d'un système d'inférence basé sur ce raisonnement. Les travaux [Stratulat2000, Rusinowitch *et al.*2000, Stratulat2008] proposent un système abstrait qui implémente la récurrence implicite et qui respecte ces contraintes d'ordre. Ce système a été démontré comme étant correct et a été instancié par le système d'inférence du démonstrateur Spike [Bouhoula *et al.*1995, Rusinowitch *et al.*2000, Barthe et Stratulat2003, Stratulat2008].

2 Exemple illustratif

Nous mettons en évidence notre intérêt à la récurrence implicite à l'aide d'un exemple. Nous considérons l'ensemble des entiers naturels, dénoté par nat. C'est un ensemble non-borné et défini récursivement. En effet, son plus petit élément est le zéro, noté par 0. Tout autre naturel peut être construit à partir de son prédécesseur en utilisant la fonction S, appelé successeur, de la façon suivante : si n est naturel alors son successeur est égal à $S(n)$. 0 et S sont les constructeurs de nat. On dénote aussi par $bool$ l'ensemble des booléens qui contient les deux éléments $true$ et $false$. Nous considérons les fonctions mathématiques usuelles : add, $even$, odd qui correspondent respectivement à l'addition sur les naturels, la parité et l'imparité d'un naturel. Elles peuvent être définies par l'ensemble d'axiomes suivants :

L'addition :

(add_1) $\forall y, add(0, y) = y$ et

(add_2) $\forall x\ y, add(S(x), y) = S(add(x, y))$.

La parité :

$(even_1)$ $even(0) = true$,

$(even_2)$ $even(S(0)) = false$,

$(even_3)$ $\forall x, odd(x) = false \Rightarrow even(S(S(x))) = true$ et
$(even_4)$ $\forall x, odd(x) = true \Rightarrow even(S(S(x))) = false$.

L'imparité :
(odd_1) $odd(0) = false$,
(odd_2) $\forall x, even(x) = false \Rightarrow odd(S(x)) = false$ et
(odd_3) $\forall x, even(x) = true \Rightarrow odd(S(x)) = true$.

Nous supposons que le lemme suivant est déjà prouvé :

$$(l) : \forall x\ y, add(x, S(y)) = S(add(x, y)).$$

Nous allons essayer de prouver la proposition suivante par récurrence structurelle ensuite par récurrence implicite :

Proposition 1 : L'addition d'un naturel à lui-même est un naturel pair :

$$(e) : \forall x, even(add(x, x)) = true.$$

Avant de commencer les preuves, nous considérons que :
- Une conjecture ayant l'une de deux formes suivantes : $t = t$ ou $t = s \Rightarrow t = s$ est prouvé et éliminée de la preuve. Toute opération d'élimination d'une conjecture sera notée par **T**.
- Une conjecture de la forme $t = t \Rightarrow e$ peut être simplifiée à la conjecture e. Cette opération sera notée par **S**.
- Une simplification de conjecture permet de remplacer des termes égaux par égaux, i.e. un terme t' dans une conjecture e est remplacé par s' si $t = s$ et t' (respectivement s') est une instance de t (respectivement s). La conjecture $t = s$ peut être un axiome, un lemme ou une hypothèse de récurrence. Cette opération est notée par $\mathbf{R}(nom)$, où nom est le nom de $t = s$.
- Deux axiomes ayant la forme suivante : $(ax_1) : t_1 = true \Rightarrow t = s$ et $(ax_2) : t_1 = false \Rightarrow t = s$ peuvent être utilisés pour simplifier une conjecture e. Cette simplification engendre les deux conjectures suivantes : $t'_1 = true \Rightarrow e_1$ et $t'_1 = false \Rightarrow e_2$. Les conjectures e_1 et e_2 sont la conjecture e en remplaçant le terme t' par s', où les termes t', s' et t'_1 sont respectivement des instances des termes t, s et t_1. Ces opérations seront notées par $\mathbf{R}(ax_1)$ et $\mathbf{R}(ax_2)$.

Des définitions plus précises et formelles de ces faits sont détaillées et justifiées dans la suite de ce document.

Preuve par récurrence structurelle de la Proposition 1 (Figure 1) : Nous commençons la preuve par récurrence structurelle en appliquant le `principe de Peano` sur e, où P est la conjecture e, et nous obtenons :

$$(e_1) : even(add(0, 0)) = true \text{ et}$$
$$\forall x, (HI_1) : even(add(x, x)) = true \Rightarrow (e_2) : even(add(S(x), S(x))) = true.$$

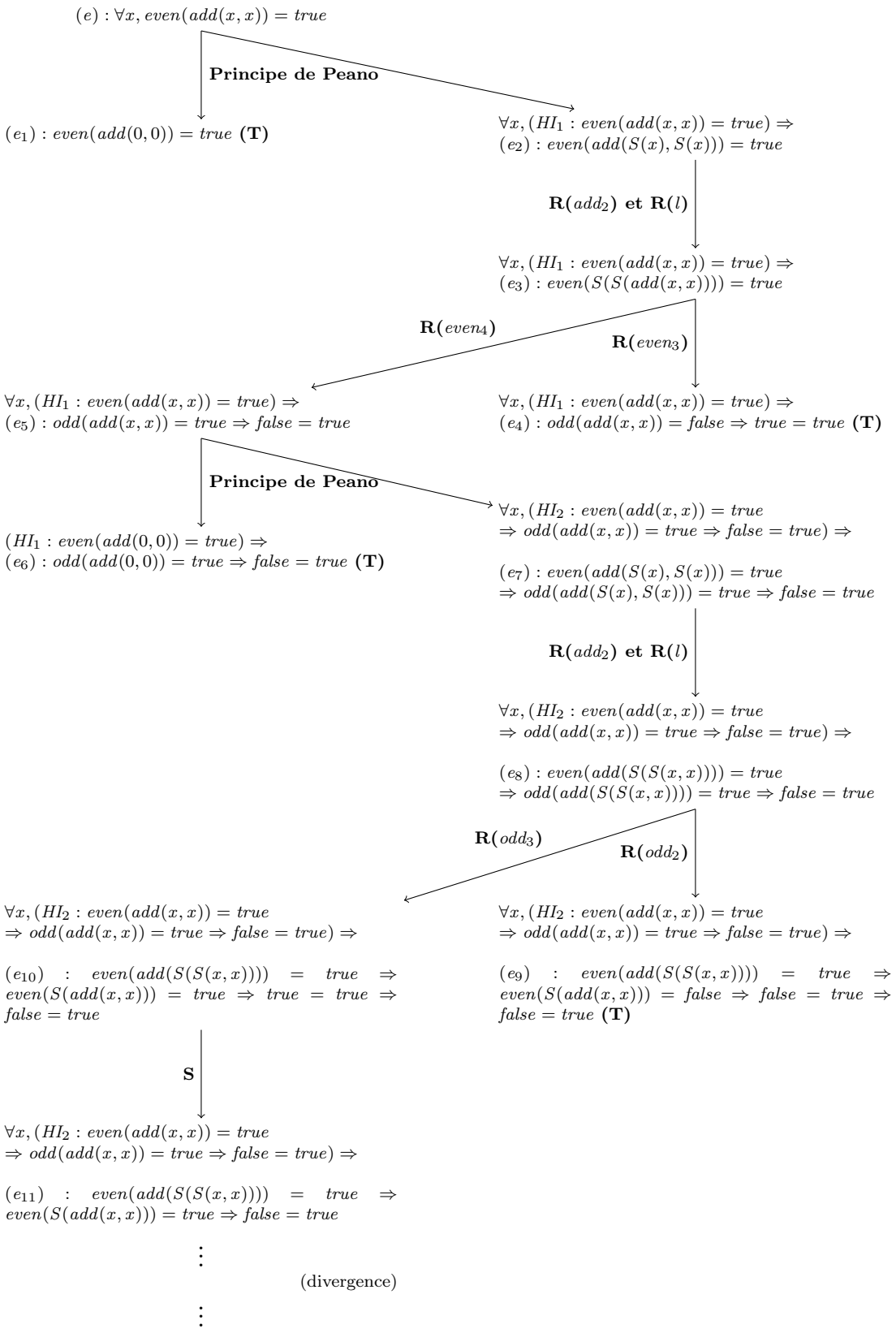

FIGURE 1 – Divergence de la construction de la preuve par récurrence structurelle de la **Proposition 1**

La conjecture e_1 est prouvée après deux simplifications successives en utilisant add_1 et $even_1$. La conjecture e_2 est simplifiée avec add_2 et le lemme l et nous obtenons :

$$\forall x, (HI_1) : even(add(x, x)) = true \Rightarrow (e_3) : even(S(S(add(x, x)))) = true.$$

La simplification de e_3 par $even_3$ et $even_4$ donne les deux conjectures suivantes :

$$\forall x, (HI_1) : even(add(x, x)) = true \Rightarrow (e_4) : odd(add(x, x)) = false \Rightarrow true = true \text{ et}$$
$$\forall x, (HI_1) : even(add(x, x)) = true \Rightarrow (e_5) : odd(add(x, x)) = true \Rightarrow false = true.$$

La conjecture $\forall x, HI_1 \Rightarrow e_4$ est prouvée et éliminée de la preuve. Nous ré-appliquons de nouveau le `principe de Peano` sur $\forall x, HI_1 \Rightarrow e_5$ et nous obtenons :

$$(e_6) : even(add(0, 0)) = true \Rightarrow odd(add(0, 0)) = true \Rightarrow false = true \text{ et}$$

$$\forall x, (HI_2) : even(add(x, x)) = true \Rightarrow odd(add(x, x)) = true \Rightarrow false = true \Rightarrow$$
$$(e_7) : even(add(S(x), S(x))) = true \Rightarrow odd(add(S(x), S(x))) = true \Rightarrow false = true.$$

La conjecture e_6 est prouvée et éliminée de la preuve après sa simplification avec add_1 et odd_1. La conjecture e_7 est simplifiée en :

$$(e_8) : even(S(S(add(x, x)))) = true \Rightarrow odd(S(S(add(x, x)))) = true \Rightarrow false = true.$$

La simplification de e_8 avec les deux axiomes odd_2 et odd_3 donne les deux conjectures suivantes :

$$(e_9) : even(S(S(add(x, x)))) = true \Rightarrow even(S(add(x, x))) = false \Rightarrow false = true \Rightarrow false = true \text{ et}$$
$$(e_{10}) : even(S(S(add(x, x)))) = true \Rightarrow even(S(add(x, x))) = true \Rightarrow true = true \Rightarrow false = true.$$

La conjecture e_9 est prouvée et éliminée de la preuve. En simplifiant e_{10}, nous obtenons

$$(e_{11}) : even(S(S(add(x, x)))) = true \Rightarrow even(S(add(x, x))) = true \Rightarrow false = true.$$

Il est clair que e_{11} ne peut pas être réduite trivialement, ce qui nous fait penser à effectuer d'autres étapes de récurrence, à simplifier des hypothèses de récurrence ou à introduire des nouvelles lemmes. Néanmoins, cette conjecture peut être résolue, d'une façon plus simple, par la récurrence implicite comme c'est illustré par la preuve suivante.

Preuve par récurrence implicite de la Proposition 1 (Figure 2) : La preuve par récurrence, telle qu'elle est effectuée par Spike, commence par une analyse par cas, notée dans la suite par **A**. Cette analyse engendre un ensemble des instances de e. Le calcul de cet ensemble s'effectue en se basant sur les axiomes qui définissent add. Pour prouver e, il suffit de prouver ses instances suivantes :

$$(e'_1) : even(0) = true \text{ (calculée à partir de } add_1) \text{ et}$$
$$(e'_2) : \forall x, even(S(add(x, S(x)))) = true \text{ (calculée à partir de } add_2).$$

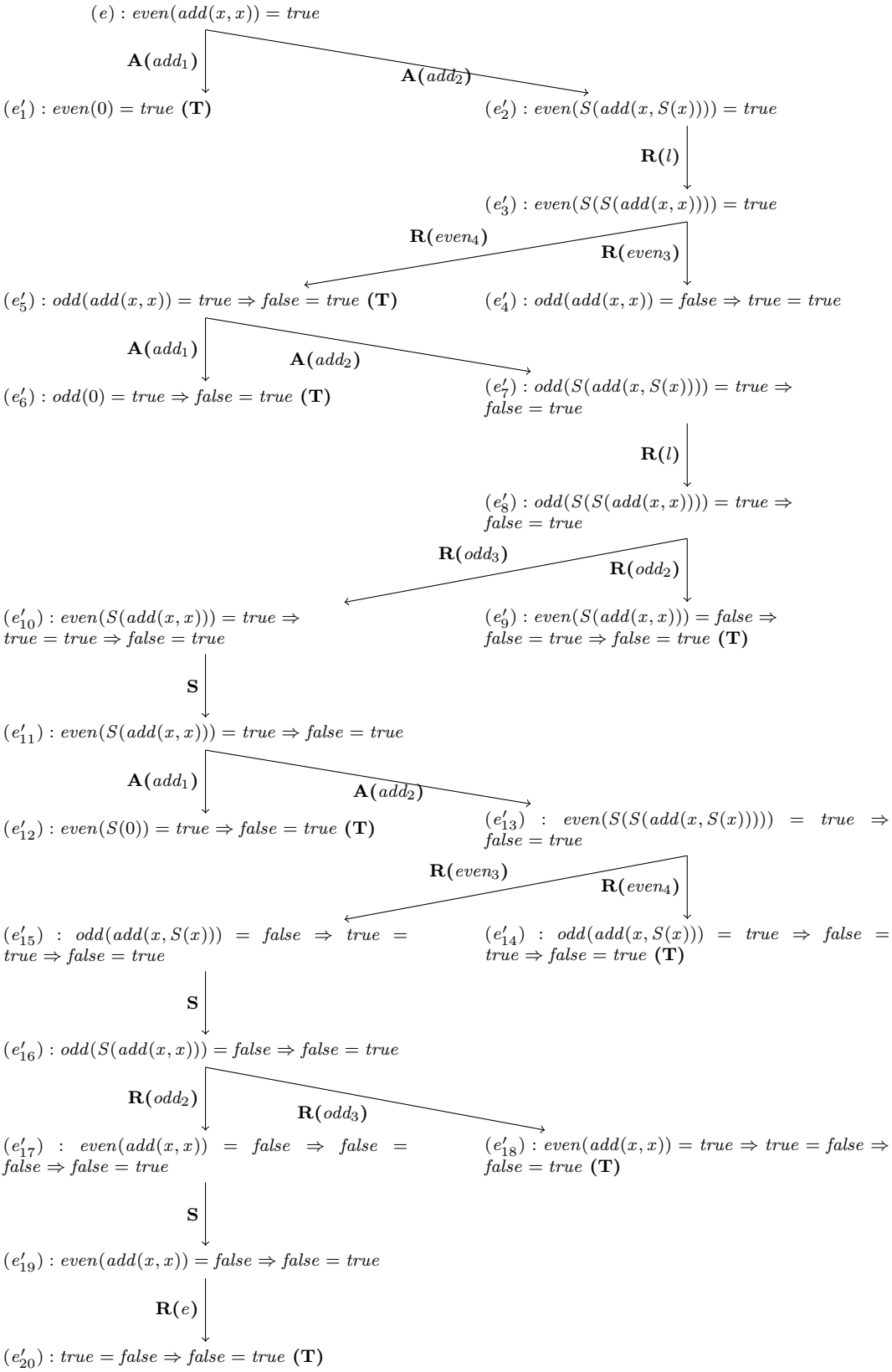

FIGURE 2 – Preuve par récurrence implicite de la **Proposition 1**

La conjecture e'_1 est prouvée et éliminée de la preuve. La conjecture e'_2 est simplifiée par le lemme l pour donner :

$$(e'_3) : \forall x, even(S(S(add(x, x)))) = true.$$

La simplification de e'_3 avec $even_3$ et $even_4$ donne :

$$(e'_4) : \forall x, odd(add(x, x)) = false \Rightarrow true = true \text{ et}$$
$$(e'_5) : \forall x, odd(add(x, x)) = true \Rightarrow false = true.$$

La conjecture e'_4 est prouvée et éliminée de la preuve. L'analyse par cas de e'_5 donne :

$$(e'_6) : odd(0) = true \Rightarrow false = true \text{ et}$$
$$(e'_7) : \forall x, odd(S(add(x, S(x)))) = true \Rightarrow false = true.$$

La conjecture e'_6 est prouvée et éliminée de la preuve. Par contre, la conjecture e'_7 est simplifiée tout d'abord avec le lemme l et qui donne :

$$(e'_8) : \forall x, odd(S(S(add(x, x)))) = true \Rightarrow false = true,$$

ensuite par les deux axiomes odd_2 et odd_3 pour donner les deux conjectures suivantes :

$$(e'_9) : \forall x, even(S(add(x, x))) = false \Rightarrow false = true \Rightarrow false = true \text{ et}$$
$$(e'_{10}) : \forall x, even(S(add(x, x))) = true \Rightarrow true = true \Rightarrow false = true.$$

La conjecture e'_9 est prouvée et éliminée de la preuve. La simplification de e'_{10} donne :

$$(e'_{11}) : \forall x, even(S(add(x, x))) = true \Rightarrow false = true.$$

Nous effectuons une analyse par cas sur e'_{11} pour obtenir :

$$(e'_{12}) : even(S(0)) = true \Rightarrow false = true \text{ et}$$
$$(e'_{13}) : \forall x, even(S(S(add(x, S(x))))) = true \Rightarrow false = true.$$

La conjecture e'_{12} est prouvée et éliminée de la preuve. La conjecture e'_{13} se simplifie en utilisant l, puis en utilisant $even_3$ et $even_4$, et nous obtenons ainsi :

$$(e'_{14}) : \forall x, odd(add(x, S(x))) = false \Rightarrow true = true \Rightarrow false = true \text{ et}$$
$$(e'_{15}) : \forall x, odd(add(x, S(x))) = true \Rightarrow false = true \Rightarrow false = true.$$

La conjecture e'_{15} est prouvée et éliminée de la preuve. La simplification de e'_{14} donne :

$$(e'_{16}) : \forall x, odd(S(add(x, x))) = false \Rightarrow false = true.$$

La simplification de e'_{16} en utilisant odd_2 et odd_3 donne les deux conjectures suivantes :

$$(e'_{17}) : \forall x, even(add(x, x)) = false \Rightarrow false = false \Rightarrow false = true \text{ et}$$
$$(e'_{18}) : \forall x, even(add(x, x)) = true \Rightarrow true = false \Rightarrow false = true.$$

La conjecture e'_{18} est prouvée et éliminée de la preuve. La simplification de e'_{17} donne :

$$(e'_{19}) : \forall x, even(add(x, x)) = false \Rightarrow false = true.$$

La conjecture e'_{19} est simplifiée par la conjecture initiale e et nous obtenons la conjecture :

$(e'_{20}) : true = false \Rightarrow false = true$ qui est prouvée et éliminée de la preuve.

3 Intégration des preuves par récurrence implicite dans Coq

Le démonstrateur de théorèmes Spike peut raisonner automatiquement sur des preuves de grande taille et contenant des structures de données et des fonctions mutuellement définies. Ses résultats sont prometteurs, et nous en citerons par exemple la validation de la plateforme Javacard [Barthe et Stratulat2003] et la preuve de conformité du protocole ABR [Rusinowitch *et al.*2003]. Néanmoins, on n'a jamais prouvé que son implémentation est correcte. De ce fait, une question importante s'impose : *Les preuves par récurrence implicite générées par Spike sont-elles correctes ?* Une telle question nous amène à la certification de démonstrateur des théorèmes Spike. Deux approches seront envisagées : i) la vérification de l'implémentation du système d'inférence de Spike, et ii) la vérification de ses preuves.

La vérification de la conformité de l'implémentation de Spike par rapport au système d'inférence proposé dans [Stratulat2000, Stratulat2001] est une tâche longue est fastidieuse. Elle nous amènera à la vérification des milliers de lignes de code en OCAML [Spike Development Team, Leroy *et al.*2014]. Un exemple comme la validation du noyau *seL4* du système d'exploitation *L4* [Klein *et al.*2010] montre clairement la complexité et le coût très élevé de cette approche. Pour certifier la correction de 8700 lignes de code C et 600 lignes d'assembleur, il a nécessité de faire une preuve de 200,000 lignes de script ISABELLE [The ISABELLE development team] et un effort total de 20 personnes-année.

Pour cette raison, dans cette thèse nous nous contentons de valider uniquement les preuves de Spike en les certifiant à l'aide de l'environnement de preuve certifié de l'assistant à la preuve Coq [Coq development team]. Un tel environnement possède une propriété très importante qui consiste à réaliser une séparation entre le moteur de preuve qui contient les outils de la construction des preuves, et le noyau qui effectue la vérification de ces preuves. Le noyau étant déjà vérifié, il permet de valider toutes les preuves construites dans Coq. Néanmoins, cet assistant ne possède pas les outils qui permettent de valider trivialement le raisonnement par récurrence implicite. De ce fait, nous proposerons ici différentes manières d'intégration de ce raisonnement dans Coq.

3.1 Appeler Spike à partir de Coq

L'une des méthodes d'intégration du raisonnement par récurrence implicite dans Coq est l'appel de Spike pour construire la preuve et la vérifier ensuite par Coq. Cet appel doit être précédé d'une préparation de la spécification Spike qui contient une description du contexte et la conjecture à prouver qui correspond au théorème courant. Spike essaie de trouver une preuve par récurrence implicite de cette conjecture. En cas de réussite, il traduit la preuve en un script que Coq le valide, ce qui a pour conséquence la validation de la preuve par récurrence implicite fournie par Spike. Cette solution nécessite la création de deux composantes automatiques dans les deux assistants afin de permettre la traduction dans les deux sens.

3.2 Intégrer la récurrence implicite à la Spike dans Coq

Une preuve par récurrence implicite peut être construite directement dans Coq sans passer par Spike dans le cas où il existe un moyen qui simule sa procédure générale directement dans Coq. Pour ce faire, tout d'abord il faut créer des outils permettant d'effectuer des raisonnements de Spike (tels que les analyses par cas, la réécriture et la subsumption). Ensuite, il est nécessaire de trouver un moyen qui permet de définir un ordre de récurrence sur les formules. Puis, une étape de vérification supplémentaire est essentielle pour valider les hypothèses de récurrence, et ce en appliquant le principe de récurrence sur l'ensemble des formules de la preuve.

3.3 Effectuer le raisonnement par récurrence paresseuse et mutuelle interactivement dans Coq

Pour effectuer interactivement le raisonnement par récurrence mutuelle et paresseuse dans Coq, il faut permettre l'utilisation des informations qui ne sont pas encore prouvées. Une telle tâche ne peut être triviale car elle peut être poursuivie par une étape de validation qui se base sur la définition d'un schéma de récurrence général justifiant l'application de toutes les hypothèses de récurrence de la preuve.

4 Plan de la thèse

À travers ce rapport, nous présentons nos travaux concernant les différentes méthodes d'intégration des preuves par récurrence implicite dans Coq. Outre l'introduction générale, il contient les chapitres suivants :

Chapitre 1 : Il introduit les algèbres de termes utilisées pour définir la conséquence inductive entre un ensemble de clause égalitaire et une égalité. Ces notions de base sont utilisées pour définir d'une façon générale le raisonnement par récurrence dans la logique de premier ordre et ses instances telles que le raisonnement par récurrence basée sur les termes et le raisonnement par récurrence basée sur les clauses égalitaires. Nous définirons ainsi l'argument de la correction de la récurrence noethérienne qui consiste à l'ordre de récurrence.

Chapitre 2 : Nous présenterons dans ce chapitre le démonstrateur des théorèmes Spike : sa spécification, sa méthode de construction de preuve et son système d'inférence. Afin de montrer que ce dernier est correct, nous définirons ce système en tant qu'une instance d'un système abstrait qui implémente la récurrence implicite et qui est correct. Nous discuterons aussi la correction des preuves par récurrence implicite générées par Spike.

Chapitre 3 : Il présente notre problématique générale qui consiste à la certification des preuves par récurrence basée sur des clauses égalitaires. En s'inspirant des exemples concrets ayant le même objectif que le notre -la certification des preuves construites automatiquement-, nous proposerons trois solutions possibles. Nous expliquerons ces solutions par des algorithmes généraux et nous discuterons les avantages et les limites de chacun.

Chapitre 4 : Nous introduirons l'environnement de preuve certifiée de Coq que nous utiliserons pour réaliser les approches générales proposées. Nous mettrons l'accent surtout sur les outils et les notions qui nous permettent de réaliser notre objectifs. Nous présenterons aussi la représentation de principe de récurrence noethérienne dans Coq.

Chapitre 5 : Via un exemple qui consiste à la certification de la preuve par récurrence implicite de notre exemple illustratif, nous exposerons la méthode basée sur l'appel de Spike

à partir de Coq. Nous présenterons les différentes étapes de cette approche en mettant l'accent sur la traduction des preuves par récurrence implicite à des script Coq.

Chapitre 6 : Nous présenterons une tactique de preuve automatique dans Coq qui permet de construire des preuves par récurrence implicite à la Spike. Cette approche évite l'utilisation de Spike pour la génération des preuves. Néanmoins, elle utilise la même technique de validation de la preuve.

Chapitre 7 : Nous décrirons un prototype de solution interactive pour construire et valider des preuves par récurrence mutuelle et paresseuse dans Coq, indépendamment de Spike. Cette approche permet enfin de construire et de valider des preuves basées sur les clauses mais qui ne sont pas réductibles.

Enfin, nous conclurons et présenterons d'éventuels travaux qui proposeraient des améliorations et des extensions.

Chapitre 1

Preuves par récurrence noethérienne dans la logique de premier ordre

Sommaire

Dans ce chapitre nous présenterons le raisonnement par récurrence et ses instances dans la logique égalitaire. Ainsi, nous étudierons quelques propriétés de certaines de ces instances et des relations entre elles. Mais avant tout, nous introduirons des notions de base liées à la logique égalitaire de premier ordre.

Plusieurs notions et définitions sont inspirées de [Baader et Nipkow1998, Barwise *et al.*2000, Buss1998a, Stratulat2012] et les rapports de thèse [Bouhoula1994, Stratulat2000] et le mémoire de l'habilitation à diriger les recherches [Contejean2014].

1.1 Algèbres de termes

Notation 1.1 *Nous considérons l'alphabet construit par l'union des ensembles disjoints suivants :*

- *un ensemble dénombrable de symboles de variables noté par* $\mathscr{X}$*,*
- *un ensemble dénombrable de symboles de fonctions noté par* $\mathscr{F}$*, et*
- *un ensemble des symboles de ponctuation contenant les parenthèses -(-, -)- et la virgule -,-.*

Nous notons par $\mathscr{L}$ *le langage défini sur cet alphabet.*

Afin de préciser les compositions des symboles autorisées sur $\mathscr{L}$, nous utilisons des étiquettes appelées des symboles de **sortes**. Ces étiquettes sont associées à chaque terme du vocabulaire pour exprimer sa nature. Nous notons l'ensemble de symboles de sortes par $\mathscr{S}$ et nous supposons qu'il est fini.

Définition 1.2 (Signature) *Une* ***signature*** Σ *est définie par la paire* $(\mathscr{S}, \mathscr{F})$*. Tout symbole de* ***fonction*** f *est muni d'une* ***arité*** n *et d'un* ***profil*** $f : s_1 \times \ldots \times s_n \to s_{n+1}$*, où* $n \in \mathbb{N}$ *et* $\{s_1, \ldots, s_n, s_{n+1}\} \subseteq \mathscr{S}$*. Tout symbole d'arité zéro est appelé une* ***constante****.*

Exemple 1.3 *Soit* $\Sigma_e = (\mathscr{S}_e, \mathscr{F}_e)$ *la signature de l'exemple présenté dans l'****Introduction****, où :*

- $\mathscr{S}_e := \{nat, bool\}$ *et*
- $\mathscr{F}_e := \{0, S, true, false, add, even, odd\}$*, où :*
  - $0, true$ *et* $false$ *sont d'arité* 0 *et des profils :* $0 : \to nat, true : \to bool$ *et* $false : \to bool$*,*
  - S *est d'arité* 1 *et de profil* $S : nat \to nat$*,*
  - add *est d'arité* 2 *et de profil* $add : nat \times nat \to nat$*,*
  - $even$ *est d'arité* 1 *et de profil* $even : nat \to bool$*, et*
  - odd *est d'arité* 1 *et de profil* $odd : nat \to bool$*.*

Les symboles de fonction $0, true$ *et* $false$ *sont des constantes.*

Définition 1.4 (Terme) *Soit* $\mathscr{T}(\Sigma, \mathscr{X})$ *l'ensemble des* ***termes*** *construits sur la signature* Σ *et les symboles de variables* $\mathscr{X} = \uplus_{s \in \mathscr{S}} \mathscr{X}_s$*, où* $\mathscr{X}_s$ *est l'ensemble des variables de sorte* s*. L'ensemble* $\mathscr{T}(\Sigma, \mathscr{X})$ *est le plus petit ensemble défini récursivement de la façon suivante :*

1. *si* $x \in \mathscr{X}_s$*,* $s \in \mathscr{S}$ *alors* x *est un terme variable, ou tout simplement une variable, de sorte* s *et on obtient* $x \in \mathscr{T}(\Sigma, \mathscr{X})$ *; ou*
2. *si* $f \in \mathscr{F}$*,* $f : s_1, \ldots, s_n \to s_{n+1}$ *et* $t_1, \ldots, t_n$ *sont des termes respectivement de sortes* $s_1, \ldots, s_n$ *alors le terme* $f(t_1, \ldots, t_n)$ *est un terme non-variable de sorte* s_{n+1} *et on obtient* $f(t_1, \ldots, t_n) \in \mathscr{T}(\Sigma, \mathscr{X})$*.*

L'ensemble $\mathscr{T}(\Sigma, \mathscr{X})_s$ *dénote l'ensemble des termes de sorte* $s \in \mathscr{S}$*.*

Exemple 1.5 *Dans l'****Exemple 1.3****, soient* x *et* y *deux variables de sorte* nat*. Alors :*

– $add(x, x)$ *et* $add(0, y)$ *sont des termes de sorte nat, et*
– $even(add(x, x))$ *et* $odd(S(S(add(x, x))))$ *sont des termes de sorte bool.*

Les prédicats dans la logique de premier ordre sont des symboles qui permettent d'exprimer des propriétés ou des relations sur les termes. Dans nos travaux, nous nous intéressons uniquement au **prédicat d'égalité** de profil $=: \mathscr{T}(\Sigma, \mathscr{X})_s \times \mathscr{T}(\Sigma, \mathscr{X})_s \to bool, s \in \mathscr{S}$. Nous considérons que les variables sont quantifiées universellement. Une **égalité** est une paire de termes $(t, t') \in \mathscr{T}(\Sigma, \mathscr{X})_s \times \mathscr{T}(\Sigma, \mathscr{X})_s, s \in \mathscr{S}$, notée par $t = t'$.

Les égalités sont généralisées par des clauses égalitaires, obtenues par la composition des différentes égalités en utilisant des connecteurs logiques usuels tels que : la conjonction -$\wedge$-, la disjonction -$\vee$- et l'implication -$\Rightarrow$-. La définition des clauses égalitaires se base sur la notion de multi-ensemble. Pour cela, nous définissons d'abord les multi-ensembles ensuite les clauses égalitaires.

Définition 1.6 (Multi-ensemble) *Un* ***multi-ensemble*** *sur des éléments d'un ensemble A se définit en tant qu'une fonction M de profil $M : A \to nat$. On dit que $M(x)$ tel que $x \in A$ est le nombre des copies de x dans M.*

Notation 1.7 *Soient M et N deux multi-ensembles. Les notations suivantes $\setminus\!\setminus$ et $\Cup$ représentent :*
– *la différence entre M et N : si $M(x) > N(x)$ alors $(M \setminus\!\setminus N)(x) := M(x) - N(x)$ sinon $(M \setminus\!\setminus N)(x) := 0$;*
– *l'union de M et N : $(M \Cup N)(x) := M(x) + N(x)$.*

Définition 1.8 (Clause égalitaire) *Une* ***clause égalitaire*** *est de la forme :*

$$e_1 \wedge \ldots \wedge e_n \Rightarrow e'_1 \vee \ldots \vee e'_m,$$

où $\{e_1, \ldots, e_n, e'_1, \ldots, e'_m\}$ est un multi-ensemble d'égalités. Les égalités $\{e_1, \ldots, e_n\}$ (respectivement $\{e'_1, \ldots, e'_m\}$) sont les conditions (respectivement les conclusions) de la clause.
L'ensemble des clauses égalitaires construites avec des $\mathscr{T}(\Sigma, \mathscr{X})$-termes est noté par $CE(\Sigma, \mathscr{X})$.

Nous nous intéressons particulièrement aux **clauses de Horn** qui sont des clauses égalitaires ayant l'une de deux formes suivantes :
– $e_1 \wedge \ldots \wedge e_n \Rightarrow$, ou
– $e_1 \wedge \ldots \wedge e_n \Rightarrow e'_1$.
La dernière forme est appelée **égalité conditionnelle** si l'ensemble de ses conditions est non vide. Dans le cas contraire, elle est une **égalité non-conditionnelle**.

Exemple 1.9 *Dans l'**Introduction**, nous avons rencontré les égalités non-conditionnelles : $add(x, 0) = x$ et $even(x) = true$, et les égalités conditionnelles : $odd(add(x, x)) = false \Rightarrow true = true$ et $odd(add(x, S(x))) = true \wedge false = true \Rightarrow false = true$. Néanmoins, nous ne pouvons pas construire les égalités : $even(x) = add(x, x)$, $even(x) = x$ et $add(x, x) = true$.*

Des clauses égalitaires reconnues par leur forme syntaxique particulière sont les tautologies.

Définition 1.10 (Tautologie) *Une tautologie est une clause égalitaire qui a l'une des formes suivantes :*
– $P \Rightarrow P'_1 \vee t = t \vee P'_2$,

- $P_1 \wedge s = t \wedge P_2 \Rightarrow P'_1 \vee s = t \vee P'_2$, *ou*
- $P_1 \wedge s = t \wedge P_2 \Rightarrow P'_1 \vee t = s \vee P'_2$

avec P, P_1 *et* P_2 *sont des conjonctions d'égalités éventuellement vides et* P'_1 *et* P'_2 *sont des disjonctions d'égalités éventuellement vides.*

Dans la suite, nous introduisons différentes notions et notations définies sur des termes ou des clauses.

Définition 1.11 (Sous-termes d'un terme et termes d'une clause)

- *Soit* $t \in \mathscr{T}(\Sigma, \mathscr{X})$. *L'ensemble des* ***sous-termes*** *de* t, *noté par* $ST(t)$, *se définit de la façon suivante :*
  - *si* t *est une variable,* $ST(t) := \{t\}$, *et*
  - *si* t *est le terme* $f(t_1, \ldots, t_n)$, $ST(t) := \{f(t_1, \ldots, t_n)\} \cup \bigcup_{i=1}^{n} ST(t_i)$.
- *Soit* $c \in CE(\Sigma, \mathscr{X})$ *ayant la forme suivante :* $s_1 = t_1 \wedge \ldots \wedge s_n = t_n \Rightarrow s'_1 = t'_1 \vee \ldots \vee s'_m = t'_m$. *Le multi-ensemble des termes de* c, *noté par* $MT(c)$, *est* $\{s_1, t_1, \ldots, s_n, t_n, s'_1, t'_1, \ldots, s'_m, t'_m\}$.

Exemple 1.12

- $ST(even(add(x, x))) := \{even(add(x, x)), add(x, x), x\}$.
- $MT(odd(add(x, S(x))) = true \wedge false = true \Rightarrow false = true) := \{odd(add(x, S(x))), true, false, true, false, true\}$.

Notation 1.13 (Variables d'un terme (d'une clause)) *Soient* $t \in \mathscr{T}(\Sigma, \mathscr{X})$ *et* $c \in CE(\Sigma, \mathscr{X})$.

- *L'ensemble des variables de* t, *noté par* $Var(t)$, *est l'ensemble de ses sous-termes variables, i.e.* $Var(t) := ST(t) \cap \mathscr{X}$.
- *L'ensemble des variables de* c, *noté par* $Var(c)$, *est l'ensemble des variables de tous ses termes, i.e.* $Var(c) := \cup_{t \in MT(c)} Var(t)$.

Exemple 1.14

- $Var(even(add(x, x))) := \{x\}$
- $Var(odd(0)) := \emptyset$
- $Var(odd(add(x, S(x))) = true \wedge false = true \Rightarrow false = true) := \{x\}$

Les **termes clos** et les **clauses closes** sont des termes et des clauses sans variables. Nous supposons qu'il existe au moins un terme clos dans chaque sorte.

Notation 1.15

- *L'ensemble des termes clos définis sur* Σ *est noté par* $\mathscr{T}(\Sigma)$.
- *L'ensemble des clauses égalitaires closes définies sur* $\mathscr{T}(\Sigma)$ *est noté par* $CE(\Sigma)$.

Exemple 1.16

- $\{even(0), odd(0), even(S(0))\} \subset \mathscr{T}(\Sigma_e)$.
- $\{even(0) = true, odd(0) = true \Rightarrow false = true, even(S(0)) = true \Rightarrow false = true, true = false \Rightarrow false = true\} \subset CE(\Sigma_e)$.

Afin de se déplacer dans un terme et d'accéder à ses sous-termes, nous utiliserons la notion de position.

Définition 1.17 (Position) *Soit t un terme. L'ensemble des* ***positions*** *d'un terme t est un ensemble des chaînes de naturels, noté $Pos(t)$. Notons que -.- est l'opération de concaténation de deux chaînes de naturels. La position se définit récursivement sur le terme t comme suit :*

- *si t est une variable alors $Pos(t) := \{\epsilon\}$, où ϵ est la chaîne d'entiers vide, et*
- *si t est le terme $f(t_1, \ldots, t_n)$ alors $Pos(t) := \{\epsilon\} \cup \bigcup_{i=1}^{n} \{i.p | p \in Pos(t_i)\}$.*

Soient t un terme et p une position de t. Le sous-terme unique de t en p, noté par $t|_p$, se définit comme suit :

- si p est ϵ alors $t|_\epsilon$ est égal à t,
- si p est $i.q$ et t est égal à $f(t_1, \ldots, t_i, \ldots, t_n)$ alors $t|_{i.q}$ est défini par $t_i|_q$. Nous notons aussi le sous-terme t_i par $t[t_i]_p$.

Un terme t d'une clause c est noté par $c[t]$. Un sous-terme u de t en une position p est noté par $c[t[u]_p]$.

Exemple 1.18 *$Pos(even(S(S(add(x, x)))))) := \{\epsilon, 1, 1.1, 1.1.1, 1.1.1.1, 1.1.1.2\}$.*
Le sous-terme $add(x, x)$ est à la position 1.1.1 *dans le terme $even(S(S(add(x, x))))$.*

Définition 1.19 (Substitution) *Une substitution σ est la famille d'applications :*

$$\sigma := \{x_1 \to t_1, \ldots, x_n \to t_n\}, \forall i \in [1..n], x_i \in \mathscr{X}_{s_i}, t_i \in \mathscr{T}(\Sigma, \mathscr{X})_{s_i}, s_i \in \mathscr{S},$$

où $x_i \neq x_j$ si $i \neq j$ pour tout $i, j \in [1..n]$.

L'ensemble des variables $\{x_1, \ldots, x_n\}$ est appelé le **domaine** de σ et est noté par $Dom(\sigma)$. L'ensemble des termes $\{t_1, \ldots, t_n\}$ est appelé l'**image** de σ et noté par $Image(\sigma)$. On dit que σ **instancie** x_i par t_i pour toute variable $x_i \in Dom(\sigma)$ et substitue un terme t_i à x_i pour tout terme $t_i \in Image(\sigma)$, et on appelle t_i un **terme image** de x_i par σ.

Soient σ une substitution et t un terme. L'application de σ sur un terme t est notée par $t\sigma$ et est définie récursivement comme suit :

- si t est une variable x_i telle que $x_i \in Dom(\sigma)$ alors $t\sigma$ est égal à t_i ,
- si t est une variable x telle que $x \notin Dom(\sigma)$ alors $t\sigma$ est égal à x, et
- si t est un terme qui a la forme suivante $f(t_1, \ldots, t_n)$ tel que $f : s_1 \times \ldots \times s_n \to s_{n+1}, \{s_1, \ldots, s_n, s_{n+1}\} \subseteq \mathscr{S}$ alors $t\sigma$ est égal à $f(t_1\sigma, \ldots, t_n\sigma)$.

Soient σ une substitution et c la clause égalitaire :

$$s_1 = t_1 \wedge \ldots \wedge s_n = t_n \Rightarrow s'_1 = t'_1 \vee \ldots \vee s'_m = s'_m.$$

L'application de σ sur c, notée par $c\sigma$, est :

$$s_1\sigma = t_1\sigma \wedge \ldots \wedge s_n\sigma = t_n\sigma \Rightarrow s'_1\sigma = t'_1\sigma \vee \ldots \vee s'_m\sigma = t'_m\sigma.$$

Une substitution est dite **close** si tous ses termes images sont des termes clos ($Image(\sigma) \subset \mathscr{T}(\Sigma)$), et **renommage** si tous ses termes images sont des variables ($Image(\sigma) \subset \mathscr{X}$).

Notation 1.20 *L'ensemble de toutes les $\mathscr{T}(\Sigma, \mathscr{X})$-substitutions (respectivement $\mathscr{T}(\Sigma, \mathscr{X})$-substitutions closes) est noté par $Sub(\mathscr{T}(\Sigma, \mathscr{X}))$ (respectivement $Sub(\mathscr{T}(\Sigma))$).*

Définition 1.21 (Instance, filtre et unificateur) *Soient s et t deux $\mathscr{T}(\Sigma, \mathscr{X})$-termes. On dit que :*

- *t est une **instance** de s s'il existe une substitution σ telle que t est égal à $s\sigma$. On dit aussi que σ est un **filtre** de s vers t (s filtre t), et*
- *s s'unifie avec t s'il existe une substitution σ telle que $s\sigma$ est égal à $t\sigma$. On dit aussi que σ est un **unificateur** de s et t.*

Soient c_1 et c_2 deux clauses égalitaires. On dit que c_1 est une instance de c_2 s'il existe une substitution σ telle que c_1 est égale à $c_2\sigma$.

Exemple 1.22 *Soient les substitutions $\sigma_1 := \{x \to 0\}$ et $\sigma_2 := \{x \to S(x_1)\}$. Notons que la substitution σ_1 est une substitution close.*

- *En appliquant σ_1 (respectivement σ_2) au terme $even(add(x, x))$, nous obtenons le terme $even(add(0, 0))$ (respectivement $even(add(S(x_1), S(x_1)))$).*
- *En appliquant σ_1 (respectivement σ_2) à l'égalité $even(add(x, x)) = true$, nous obtenons l'égalité $even(add(0, 0)) = true$ (respectivement $even(add(S(x_1), S(x_1))) = true$).*

Par analogie avec les mathématiques, nous définirons la composition de deux substitutions.

Définition 1.23 (Composition de substitutions) *La composition de deux substitutions σ et δ, écrite $\delta\sigma$, se définit par $x(\delta\sigma) := (x\delta)\sigma$, où x est un terme ou une clause égalitaire.*

Exemple 1.24 *Soient l'égalité (e) : $even(add(x, x)) = true \in CE(\Sigma_e, \{x, x_1\})$ et les substitutions $\sigma_1 := \{x \to S(x_1)\}$ et $\sigma_2 := \{x_1 \to 0\}$. $e\sigma_1\sigma_2 := even(add(S(0), S(0))) = true$.*

Si s et t deux termes unifiables alors il existe un unificateur σ tel que pour tout autre unificateur θ on peut trouver une substitution δ vérifiant $\theta := \delta\sigma$. Cet unificateur σ, noté par $mgu(s, t)$, s'appelle **unificateur principal** de s et t. Il est unique à renommage près de variables.

Dans la suite, nous séparons l'ensemble des symboles de fonction $\mathscr{F}$ en des symboles de fonction définie $\mathscr{D}$ et des symboles de fonction constructeur $\mathscr{C}$ tel que $\mathscr{D} \cap \mathscr{C} = \emptyset$. À chaque sorte, nous associons un ensemble de symboles de fonction constructeur (appelés simplement des **constructeurs**). Nous supposons que cet ensemble contient au moins un constructeur. Un constructeur est **libre** s'il n'a pas de relations d'égalité avec d'autres constructeurs. À chaque symbole de $\mathscr{D}$ on associe un ensemble d'axiomes le définissant, contenu par des spécifications conditionnelles.

Définition 1.25 (Spécifications conditionnelle) *Une **spécification conditionnelle** est une paire $Spec := (\Sigma, Ax)$ dont Σ est sa signature et Ax est son ensemble d'égalités (conditionnelles ou non-conditionnelles).*

Exemple 1.26
Dans la signature Σ_e, $C_e := \{0, S, true, false\}$ où :

- *les fonctions 0 et S sont les constructeurs de sorte nat, et*
- *les fonctions true et false sont les constructeurs de sorte bool.*

Nous notons que ces constructeurs sont libres.

*Soit Ax_e l'ensemble d'axiomes de l'**Introduction** :*

- *$add(0, y) = y$*
- *$add(S(x), y) = S(add(x, y))$*

- $even(0) = true$
- $even(S(0)) = false$
- $odd(x) = false \Rightarrow even(S(S(x))) = true$
- $odd(x) = true \Rightarrow even(S(S(x))) = false$

- $odd(0) = false$
- $even(x) = false \Rightarrow odd(S(x)) = false$
- $even(x) = true \Rightarrow odd(S(x)) = true$

Les axiomes Ax_e définissent respectivement add, even et odd. La spécification $Spec := (\Sigma_e, Ax_e)$ est une spécification conditionnelle.

Définition 1.27 (Algèbre) *La Σ-algèbre $\mathscr{A}$ est une structure constituée :*
- *d'un support non vide $\mathscr{A}_s$ associé à chaque sorte s de $\mathscr{S}$, et*
- *d'une application qui associe à chaque symbole de fonction f de $\mathscr{F}$ une fonction $f^{\mathscr{A}}$ telle que si $f : s_1 \times \ldots \times s_n \to s_{n+1}$ alors $f^A : \mathscr{A}_{s_1} \times \ldots \times \mathscr{A}_{s_n} \to \mathscr{A}_{s_{n+1}}$.*

Exemple 1.28 *$\mathscr{T}(\Sigma, \mathscr{X})$ et $\mathscr{T}(\Sigma)$ sont des Σ-algèbres. À chaque sorte s_i de $\mathscr{S}$, nous associons le support $\mathscr{T}(\Sigma, \mathscr{X})_{s_i}$ (respectivement $\mathscr{T}(\Sigma)_{s_i}$) qui est supposé non vide -ayant au moins une constante-. Et à chaque symbole de fonction, nous associons le symbole même. Ces deux algèbres sont l'**algèbre de termes** et l'**algèbre de termes clos**.*

Définition 1.29 (Précongruence et congruence) *Soit $\mathscr{A}$ une Σ-algèbre. Une précongruence est une relation définie sur $\mathscr{A}$ compatible avec la structure de Σ-algèbre :*
- *deux éléments en relation appartiennent au support de la même sorte, et*
- *pour tout symbole de fonction $f : s_1 \times \ldots \times s_n \to s_{n+1}$, on a :*
$$si\ \forall i \in [1..n], t_i \approx u_i\ alors\ f^{\mathscr{A}}(t_1, \ldots, t_n) \approx f^{\mathscr{A}}(u_1, \ldots, u_n).$$

Une précongruence est une congruence si elle est également une relation d'équivalence (réflexive, symétrique et transitive).

1.2 Conséquence inductive

Dans cette partie, nous souhaitons évaluer la valeur de vérité des clauses, i.e. -*Vrai* ou *Faux*-. D'une façon générale, l'évaluation des formules de premier ordre s'effectue à l'aide d'une structure construite par :
- un domaine d'interprétation qui donne du sens aux objets manipulés, et
- une fonction d'interprétation qui associe un sens à chaque symbole de fonction et de prédicat, et
- une fonction d'assignation des variables qui permet de définir leurs valeurs dans le domaine d'interprétation.

Dans notre cas, nous avons uniquement des prédicats d'égalité associés à chaque sorte. Leur interprétation peut être engendrée par l'interprétation des sortes. Et la fonction d'interprétation est utilisée uniquement pour interpréter les symboles de fonction à l'aide des algèbres. Soit $\mathscr{A}$ une Σ-algèbre interprétation de $\mathscr{L}$:
- le domaine d'interprétation est le support de $\mathscr{A}$, et
- le prédicat d'égalité $=$ est interprété par différentes relations d'égalité définies entre des objets de même sorte, noté par $=_{\mathscr{A}}$.

Exemple 1.30 *Nous pouvons définir la Σ_e-algèbre suivante :*
- *Le support de la sorte nat est l'ensemble des naturels $\mathbb{N}$ et le support de la sorte bool est $\{true, false\}$.*
- *L'interprétation de 0 est égale au naturel 0, celle de S est la fonction $n \rightarrow n+1$, celle de add est l'addition naturelle, celle de even est la parité naturelle et celle de odd est l'imparité naturelle.*

Les algèbre des termes $\mathscr{T}(\Sigma, \mathscr{X})$ et $\mathscr{T}(\Sigma)$ peuvent être utilisées pour l'interprétation des clauses de $CE(\Sigma, \mathscr{X})$, où les symboles de fonction de Σ et le symbole de prédicat d'égalité sont interprétés par des égalités sur des termes de même sorte.

Définition 1.31 (Assignation) *Soit $\mathscr{A}$ une Σ-algèbre. Une assignation θ de $\mathscr{X}$ à valeur dans $\mathscr{A}$ est la famille d'applications :*

$$\theta_1 \biguplus \ldots \biguplus \theta_n,$$

où θ_i $(\forall i \in [1 \ldots n])$ est une application de l'ensemble de variables $\mathscr{X}_{s_i}$ de sorte s_i dans le support $\mathscr{A}_{s_i}$ de même sorte s_i $(\theta_i : \mathscr{X}_{s_i} \rightarrow \mathscr{A}_{s_i})$.

L'application de l'assignation θ à un terme t, notée par $t\theta$, donne un objet de l'algèbre $\mathscr{A}$. Cet objet est défini de la façon suivante :
- si t est une variable x alors $t\theta$ est l'image de t par θ et
- si t est un terme ayant la forme suivante $f(t_1, \ldots, t_n)$ alors $t\theta$ est égale à $f^{\mathscr{A}}(t_1\theta, \ldots, t_n\theta)$.

Définition 1.32 (Satisfaisabilité et validité d'une clause égalitaire) *Soient $\mathscr{A}$ une Σ-algèbre et la clause égalitaire c :*

$$s_1 = t_1 \wedge \ldots \wedge s_n = t_n \Rightarrow s'_1 = t'_1 \vee \ldots \vee s'_m = t'_m.$$

*On dit que $\mathscr{A}$ **satisfait** c si et seulement si pour toute assignation θ de $\mathscr{X}$ dans $\mathscr{A}$ telle que pour tout $i \in [1..n]$ on a $s_i\theta =_{\mathscr{A}} t_i\theta$, alors il existe $j \in [1..m]$ tel que $s'_j\theta =_{\mathscr{A}} t'_j\theta$. Dans ce cas, on dit que c est **valide** dans $\mathscr{A}$.*

Définition 1.33 (Modèle) *Soient $\mathscr{M}$ une Σ-algèbre et Ax un ensemble d'égalités. L'algèbre $\mathscr{M}$ est un **modèle** de Ax si et seulement si elle satisfait toutes les égalités de Ax.*

Dans nos travaux, nous nous intéresserons à des modèles de Herbrand particuliers.

Définition 1.34 (Modèle de Herbrand) *Un modèle $\mathscr{M}$ est appelé **modèle de Herbrand** si toutes ses assignations transforment les variables en des termes clos.*

Définition 1.35 (Conséquence inductive) *Soient c une clause égalitaire et Ax un ensemble d'égalités (conditionnelles ou non-conditionnelles). La clause c est une **conséquence inductive** de Ax si elle est valide dans tous les modèles de Herbrand de Ax et on note $Ax \models_{ind} c$.*

Une Σ-algèbre particulière et intéressante est l'**algèbre initiale** ou **algèbre quotient des termes**. C'est une Σ-algèbre dont le support est $\mathscr{T}(\Sigma)/_{=_{Ax}}$. Dans notre cas, la relation de congruence $=_{Ax}$ entre les éléments de $\mathscr{T}(\Sigma)$ est établie par la relation d'égalité induite par l'ensemble d'axiomes Ax de la spécification conditionnelle qui définissent les symboles de fonction définie de Σ.

Définition 1.36 (Algèbre quotient des termes) *Soit $=_{Ax}$ une congruence sur $\mathscr{T}(\Sigma)$. L'ensemble $\mathscr{T}(\Sigma)/_{=_{Ax}}$ des classes de congruence $[t]_{=_{Ax}}$ de $\mathscr{T}(\Sigma)$ constitue le domaine d'interprétation. L'algèbre quotient des termes, notée par $\mathscr{A}(\Sigma, Ax)$, est l'algèbre telle que*

$$f_{\mathscr{T}(\Sigma)/_{=_{Ax}}}([t_1]_{=_{Ax}}, \ldots, [t_n]_{=_{Ax}}) = [f(t_1, \ldots, t_n)]_{=_{Ax}},$$

pour tous $f \in \mathscr{F}, f : s_1 \times \ldots \times s_n \to s_{n+1}, t_1 \in \mathscr{T}(\Sigma)_{s_1}, \ldots, t_n \in \mathscr{T}(\Sigma)_{s_n}$.
La relation $=_{Ax}$ est la plus petite congruence engendrée par Ax sur $\mathscr{T}(\Sigma)$ qui se construit par récurrence de la façon suivante :

- *soit $=_0$ la plus petite relation de congruence qui contient la relation R_0 définie par uR_0v si et seulement si il existe une égalité $s = t$ dans Ax et une substitution σ de $\mathscr{X}$ dans $\mathscr{T}(\Sigma)$ telles que $s\sigma$ est égal à u et $t\sigma$ est égal à v.*
- *soit $=_{i+1}$ la plus petite congruence qui contient la relation R_{i+1} sur $\mathscr{T}(\Sigma)$ définie par $uR_{i+1}v$ si et seulement si $u =_i v$ ou s'il existe une égalité conditionnelle $\wedge_{j=1}^{n} u_j = v_j \Rightarrow s = t$ dans Ax, une substitution σ de $\mathscr{X}$ dans $\mathscr{T}(\Sigma)$ et une position $p \in Pos(u)$ telles que $s\sigma$ est égal à $u|_p$, $u[t\sigma]p$ est égal à v et pour tout j dans $[1..n]$, $u_j\sigma =_i v_j\sigma$.*

Alors $=_{Ax}$ est définie comme $\cup_{i\geq 0} =_i$.

Définition 1.37 (Morphisme) *Soient $\mathscr{A}$ et $\mathscr{B}$ deux Σ-algèbres et μ un Σ-morphisme de $\mathscr{A}$ vers $\mathscr{B}$ tel que pour chaque nom d'opération de la signature Σ on a $f : s_1 \times \ldots \times s_n \to s_{n+1}$ et pour tout n-uplet $(t_1, \ldots, t_n) \in (\mathscr{A}_{s_1}, \ldots, \mathscr{A}_{s_n})$ on ait :*

$$\mu(f^{\mathscr{A}}(t_1, \ldots, t_n)) := f^{\mathscr{B}}(\mu(t_1), \ldots, \mu(t_n))$$

Définition 1.38 (Modèle initial) *Un modèle est dit **initial** et noté par $\mathscr{M}^{ini}$ si pour tout modèle $\mathscr{M}$, il existe un morphisme μ tel que $\mu : \mathscr{M}^{ini} \to \mathscr{M}$.*

On peut montrer que le modèle basé sur l'algèbre quotient de termes dont les classes de congruence sont définies à partir d'un ensemble d'égalités Ax est initial et unique par rapport à tous les modèles de Herbrand de Ax [Bouhoula1994].

Définition 1.39 (Conséquence initiale) *Toute clause égalitaire valide dans le modèle initial d'un ensemble d'égalités Ax est une conséquence initiale de Ax, notée par $\models_{ini}$.*

Pour les différentes raisons abordées dans [Gramlich2005], et comme c'est mentionné dans [Stratulat2012], dans le cas où Ax est un ensemble d'égalités quantifiées universellement, nous pouvons raisonner uniquement sur son modèle initial. Nous obtenons la proposition suivante :

Proposition 1.40 *[Bouhoula1994] Soient c une clause de Horn égalitaire et Ax un ensemble d'égalités. La clause c est une conséquence inductive de Ax si et seulement elle est valide dans le modèle initial de Ax.*

Définition 1.41 (Contre exemple) *Une clause égalitaire est fausse si au moins une de ses instances closes n'est pas initialement valide. Cette instance est appelée un **contre exemple**.*

1.3 Ordres noethériens sur des clauses égalitaires

Dans cette partie, nous présenterons une définition de l'ordre noethérien.

Définition 1.42 (Ordre partiel et strict) *Un* ***ordre partiel*** $\leq$ *sur un ensemble* $\mathcal{E}$ *est une relation binaire, transitive, antisymétrique et réflexive sur* $\mathcal{E}$*. La paire* $(\mathcal{E}, \leq)$ *représente l'ensemble* $\mathcal{E}$ *partiellement ordonné. L'ordre* $<$ *est la partie stricte de* $\leq$ *qui est une relation binaire, transitive et irréflexive*[10] *et notée par* $<$*.*

Définition 1.43 (Ordre bien fondé ou noethérien) *Soit* $\leq$ *un ordre partiel sur un ensemble* $\mathcal{E}$*. L'ordre* $\leq$ *est* ***bien fondé*** *s'il n'existe pas de suite infinie d'éléments de* $\mathcal{E}$ *telle que :* $t_1 > t_2 \ldots > t_n \ldots$*.*

Définition 1.44 (Stabilité par substitution et par contexte) *Soit* $\leq$ *un ordre sur un ensemble de termes ou clauses* $\mathcal{E}$*. On dit que* $\leq$ *est* ***stable par substitution*** *si pour tous* $x, y \in \mathcal{E}$ *et toute substitution* σ*, si* $x \leq y$ *alors* $x\sigma \leq y\sigma$*. Lorsque* $\mathcal{E}$ *est une ensemble de termes, on dit que* $\leq$ *est* ***stable par contexte****, ou monotone, si* $x \leq y$*,* $x, y \in T(\Sigma, \mathscr{F})_{s_i}, s_i \in \mathscr{S}$ *et pour tout terme* $t \in \mathcal{E}$ *tel que* x *est le sous-terme de* t *à une position* p *alors* $t \leq t[y]_p$*.*

Dans nos travaux, nous considérons également des ordres de réduction sur les termes.

Définition 1.45 (Ordre de réduction) *Un ordre bien-fondé, stable par substitution et stable par contexte, est un* ***ordre de réduction****.*

Un ordre strict (respectivement partiel) défini sur un ensemble $\mathcal{E}$, peut être étendu à un ordre strict (respectivement partiel) sur des multi-ensembles d'éléments de $\mathcal{E}$.

Définition 1.46 (Ordre multi-ensemble) *Soient* $\leq$ *un ordre sur un ensemble* $\mathcal{E}$ *et* $\mathcal{E}_1, \mathcal{E}_2$ *deux multi-ensembles sur* $M(\mathcal{E})$*. On définit les deux multi-ensembles* $\mathcal{E}_1', \mathcal{E}_2'$ *comme suit :* $\mathcal{E}_1' := \mathcal{E}_1 \setminus\setminus \mathcal{E}_2$ *et* $\mathcal{E}_2' := \mathcal{E}_2 \setminus\setminus \mathcal{E}_1$*. Alors on écrit :*

1. $\mathcal{E}_1 << \mathcal{E}_2$ *si pour tout* $a \in \mathcal{E}_1'$ *il existe un élément* $b \in \mathcal{E}_2'$ *tel que* $a < b$*, et*
2. $\mathcal{E}_1 = \mathcal{E}_2$ *si* $\mathcal{E}_1' := \mathcal{E}_2' := \emptyset$*, et*
3. $\mathcal{E}_1 \underline{<<} \mathcal{E}_2$ *si* $\mathcal{E}_1 << \mathcal{E}_2$ *ou* $\mathcal{E}_1 = \mathcal{E}_2$*.*

Dans nos travaux, nous considérons l'ordre syntaxique *rpo*[11] [Dershowitz1982] avec statut multi-ensemble. C'est un ordre syntaxique qui compare deux termes récursivement en suivant les chemins qui partent de leurs têtes et arrivent à leurs feuilles. Il commence par comparer les deux symboles de la racine de deux termes, ensuite par comparer leurs sous-termes. La comparaison les sous-termes est une comparaison entre deux multi-ensembles dont chacun contient les sous-termes de terme correspondant.

Dans nos travaux, nous utilisons des ordres *rpo* avec statut multi-ensemble se base sur un ordre sur les symboles de fonctions appelé **précédence** (un ordre réflexive et transitive) sur $\mathscr{F}$.

Définition 1.47 (Ordre récursif sur les chemins avec statut multi-ensemble) *[Baader et Nipkow1998] Soit* $\leq_{\mathscr{F}}$ *une précédence entre les symboles de fonctions de* Σ*. Un* ***ordre récursif sur les chemins avec statut multi-ensemble****, noté par* $\leq_{rpo}$*, est défini sur les termes* $\mathscr{T}(\Sigma, \mathscr{X})$ *de la façon suivante :*

$$\forall s, t \in \mathscr{T}(\Sigma, \mathscr{X}), t \leq_{rpo} s \text{ si :}$$

1. $t \in Var(s)$ *et* $s \neq t$*, ou*

10. Une relation est irréflexive si elle est binaire et ne relie jamais un élément à lui même.
11. *Recursive Path Order*

2. $t := g(t_1, \ldots, t_n)$, $s := f(s_1, \ldots, s_m)$ *et*
   2.1. $\exists i \in [1..m]$ *avec* $t \leq_{rpo} s_i$, *ou*
   2.2. $g <_{\mathscr{F}} f$ *et* $\forall j \in [1..n]$ $t_j <_{rpo} s$, *ou*
   2.3. $f =_{\mathscr{F}} g$, $\forall j \in [1..n]$ $t_j \leq_{rpo} s$ *et* $\{t_1, \ldots, t_n\} \underline{\leq}_{rpo} \{s_1, \ldots, s_m\}$.

Exemple 1.48 *Soit* $<_{\mathscr{F}_e}$ *la précédence sur les symboles de fonction de la signature* Σ_e *définie de la façon suivante :*

$$0 <_{\mathscr{F}_e} S <_{\mathscr{F}_e} true <_{\mathscr{F}_e} false <_{\mathscr{F}_e} add <_{\mathscr{F}_e} odd \text{ et } odd =_{\mathscr{F}_e} even.$$

La comparaison entre les deux multi-ensembles de termes $\{even(add(x,x)), true\}$ *et* $\{even(0), true\}$ *s'effectue comme suite :*

1. *on élimine les termes communs entre ces deux multi-ensembles et on obtient :* $\{even(add(x,x))\}$ *et* $\{even(0)\}$ *;*
2. *on compare ces deux termes* $even(add(x,x))$ *et* $even(0)$ *;*
3. *ayant le même symbole d'en-tête, la comparaison de ces deux termes correspond au point 2.3 de la* ***Définition 1.47*** *où les deux multi-ensembles à comparer sont* $\{0\}$ *et* $\{add(x,x)\}$ *;*
4. *étant chacun un singleton, la comparaison de ces deux multi-ensemble consiste à comparer les deux termes* 0 *et* $add(x,x)$ *;*
5. *ayant des symboles d'en-tête différents, la comparaison de ces deux termes correspond au point 2.2 de la* ***Définition 1.47*** *avec* $0 <_{\mathscr{F}_e} add$ *et* $\emptyset << _{rpo} \{x, x\}$.

Donc nous obtenons : $0 <_{rpo} add(x,x)$, $\{0\} << _{rpo} \{add(x,x)\}$, $even(0) <_{rpo} even(add(x,x))$ *et* $\{even(0), true\} << _{rpo} \{even(add(x,x)), true\}$.

Les ordres sur les clauses se définissent comme des extensions multi-ensemble d'ordres sur les termes.

Définition 1.49 (Ordre sur les clauses égalitaires) *Soient* $\leq_{rpo}$ *un ordre sur les termes* $\mathscr{T}(\Sigma, \mathscr{X})$, $\underline{\leq}_{rpo}$ *son extension multi-ensemble et* c_1, c_2 *deux clauses définies sur* $\mathscr{T}(\Sigma, \mathscr{X})$. *Alors, l'****ordre sur les clauses égalitaires****, noté par* $\leq_c$, *est défini de la façon suivante :*

$$c_1 \leq_c c_2 \text{ si } MT(c_1) \underline{\leq}_{rpo} MT(c_2).$$

Exemple 1.50 *Soient les deux clauses* $even(add(x,x)) = true$ *et* $even(add(x,x)) = false \rightarrow false = true$. *On a* $\{even(add(x,x)), true\} << _{rpo} \{even(add(x,x)), false, false, true\}$ *parce que après l'élimination des redondances dans les deux multi-ensembles on obtient* $\emptyset << _{rpo} \{false, false\}$. *Et donc, on obtient* $even(add(x,x)) = true <_c even(add(x,x)) = false \rightarrow false = true$.

1.4 Raisonnement par récurrence noethérienne dans la logique égalitaire

Le raisonnement par récurrence permet en analysant une vérité sur des éléments plus petits de l'étendre sur la totalité des éléments d'un ensemble non vide. Il permet de prouver la validité d'une propriété pour tout élément d'un ensemble partiellement ordonné suivant un ordre

noethérien. Les mathématiciens effectuent la récurrence de la façon suivante : en commençant par une conjecture à prouver, on effectue une analyse par cas sur cette conjecture pour obtenir des nouveaux sous-buts (équivalents aux instances de la conjecture initiale). Dans le traitement de chaque nouveau sous-but, la conjecture initiale peut être instanciée comme un lemme pour résoudre d'autres conjectures, appelées des conclusions, en ne gardant dans la tête que ces instances qui correspondent à des hypothèses de récurrence. De même, la récurrence peut être re-appliquée sur l'un de ces sous-buts dont les instances joueront le rôle d'hypothèses de récurrence par rapport à d'autres conjectures. À la fin de la preuve, nous cherchons un ordre de récurrence noethérien respecté par toutes les instances des conjectures utilisées comme des hypothèses de récurrence -un ordre qui permet de prouver que toute hypothèse de récurrence soit plus petite que sa conclusion- [Wirth2004]. Ce raisonnement peut traiter un ensemble non-borné, ce qui conduit à sa large utilisation, aussi bien en mathématique qu'en informatique.

1.4.1 Principe général de la récurrence noethérienne

Le raisonnement par récurrence a pour forme générale la récurrence noethérienne[12] dont dérive une grande variété d'approches qui peuvent raisonner par récurrence sur des ensembles non-bornés bien fondés. C'est un raisonnement qui permet de prouver la validité d'une propriété pour un ensemble muni d'un ordre noethérien.

Définition 1.51 (Récurrence noethérienne) *Si $\mathcal{E}$ est un ensemble bien-fondé muni d'un ordre $<$ alors :*

$$(\forall y \in \mathcal{E}, (\forall z \in \mathcal{E}, z < y \Rightarrow \phi(z)) \Rightarrow \phi(y)) \Rightarrow \forall x \in \mathcal{E}, \phi(x).$$

La version contre-positive de la récurrence noethérienne est la Descente Infinie [Wirth2004]. Son principe consiste à supposer que si un ensemble de conjectures muni d'un ordre noethérien contient un contre exemple alors ce dernier possède un contre exemple plus petit. L'ordre noethérien garantit l'existence d'un plus petit contre exemple.

Définition 1.52 (Descente Infinie) *Si $\mathcal{E}$ un ensemble bien-fondé muni d'un ordre $<$ alors :*

$$(\forall m \in \mathcal{E}, \neg\phi(m) \Rightarrow (\exists k \in \mathcal{E}, k < m \wedge \neg\phi(k))) \Rightarrow \forall p \in \mathcal{E}, \phi(p).$$

Correction de l'application de la récurrence : Elle est garantie par l'existence d'un ordre noethérien sur les éléments de $\mathcal{E}$. Cet ordre garantit l'exclusion de toute séquence infinie et strictement décroissante d'éléments.

Classification des principes de récurrence : Ces deux principes cités ci-dessus sont l'essence de différentes méthodes et techniques formelles dont la classification suit de différents critères [Gramlich2005]. Nous différons les principes de récurrence suivant le type des éléments de $\mathcal{E}$ auxquels on applique la récurrence. Dans notre cas, nous appliquons la récurrence sur des termes ou sur des clauses égalitaires. Chaque classe possède ses avantages et ses inconvénients, et parmi les points qui font la différence entre les classes de récurrence, nous avons le calcul du schéma de récurrence et la réalisation de la récurrence paresseuse et mutuelle.

12. La récurrence noethérienne se base sur les conditions de chaînes descendantes et ascendantes qui sont définies par Emmy Noether (1882–1935).

Récurrence paresseuse : Une preuve par récurrence est dite paresseuse si elle effectue le calcul de ses hypothèses de récurrence au moment de leur emploi.

Récurrence mutuelle : Une preuve par récurrence de ϕ_1 et ϕ_2, est dite mutuelle, si la preuve de ϕ_1 utilise comme hypothèses de récurrence des instances de ϕ_2 et, réciproquement, celle de ϕ_2 utilise comme hypothèses de récurrence des instances de ϕ_1.

Schéma de récurrence : Un schéma de récurrence doit permettre de couvrir toutes les instances closes possibles de la conjecture sur laquelle il s'applique et d'exprimer pour chaque cas les hypothèses de récurrence si elles existent. On peut souvent distinguer à partir d'un schéma des cas de base et d'autres cas de récurrence, les cas de base ne possédant pas d'hypothèses de récurrence. L'application de la récurrence sur une conjecture permet de simplifier sa preuve en des preuves de sous-buts parmi lesquelles on trouve ceux qui correspondent aux cas de base et d'autres qui correspondent aux cas de récurrence.

Définition 1.53 (Schéma de récurrence et hypothèse de récurrence) *Un* ***schéma de récurrence*** *à appliquer sur une conjecture est un ensemble de règles :*

$$(\phi\sigma_1 \wedge \ldots \wedge \phi\sigma_n) \Rightarrow \phi\theta,$$

L'ensemble $\{\phi\sigma_1, \ldots, \phi\sigma_n\}$ *est l'ensemble d'****hypothèses de récurrence*** *associé à la conclusion* $\phi\theta$*. Si cet ensemble est vide, alors* $\phi\theta$ *est un* ***cas de base****, sinon c'est un* ***cas de récurrence****. À noter que les hypothèses de récurrence* $\{\phi\sigma_1, \ldots, \phi\sigma_n\}$ *peuvent participer dans la preuve de* $\phi\theta$*.*

1.4.2 Récurrence basée sur les termes vs récurrence basée sur les clauses

Dans cette thèse, nous nous intéressons aux instances de la récurrence noethérienne dans la logique de premier ordre : la récurrence basée sur les termes et la récurrence basée sur les formules (les clauses égalitaires). Ce paragraphe est une présentation de ces deux sous-classes de récurrence ainsi leurs avantages et leurs inconvénients. Ce classement et ces définitions sont inspirés de [Stratulat2012].

A. Récurrence basée sur les termes

Le raisonnement par récurrence basé sur les termes consiste à appliquer la récurrence noethérienne sur un ensemble de termes.

Définition 1.54 (Récurrence basée sur les termes) *Si* $\mathcal{E}_t$ *un ensemble de vecteurs des termes muni d'un ordre* $<_t$ *alors :*

$$(\forall \overrightarrow{m} \in \mathcal{E}_t, (\forall \overrightarrow{k} \in \mathcal{E}_t, \overrightarrow{k} <_t \overrightarrow{m} \Rightarrow \phi(\overrightarrow{k})) \Rightarrow \phi(\overrightarrow{m})) \Rightarrow \forall \overrightarrow{p} \in \mathcal{E}_t, \phi(\overrightarrow{p}).$$

Cette récurrence se base sur la définition explicite d'un schéma de récurrence qui servira à effectuer une analyse par cas sur une conjecture, et à valider les hypothèses de récurrence. Il existe plusieurs techniques pour le calcul du schéma de récurrence qui sont souvent basées sur une analyse de la conjecture courante et sur la spécification des constructeurs et des fonctions. Ainsi, cette analyse permet de déterminer les variables de récurrence, i.e. les variables de la conjecture courante qui seront instanciées.

Avantages La récurrence basée sur les termes est la forme la plus connue de la récurrence. Elle peut être mise en oeuvre par des règles d'inférence et elle ne demande ni étapes ni contraintes supplémentaires pour la valider.

Inconvénients La récurrence basée sur les termes n'effectue ni récurrence paresseuse ni récurrence mutuelle. D'une part, les hypothèses de récurrence peuvent être définies longtemps avant leur utilisation ou définies sans jamais être utilisées. D'autre part, dans plusieurs cas, on ne peut deviner laquelle des conjectures sera utilisée comme hypothèse de récurrence. Pour cela, la construction d'une preuve de récurrence basée sur les termes peut demander plusieurs essais.

Parmi les instances de récurrence basée sur les termes, nous présenterons la récurrence structurelle.

A.a. Récurrence structurelle Cette récurrence calcule les schémas de récurrence par rapport à la définition des sortes, plus précisément par rapport à l'ensemble des constructeurs.

Définition 1.55 (Récurrence structurelle) *Soit s_i une sorte munie d'un ensemble des constructeurs C_i telle que $\forall c \in C_i$ de profil $c : s_1, \ldots, s_n \to s_i$. Si ϕ est une clause égalitaire alors :*

$$(\phi(\ldots, x'_1, \ldots) \wedge \ldots \wedge \phi(\ldots, x'_k, \ldots) \Rightarrow \phi(\ldots, c(x_1, \ldots, x_n), \ldots)) \Rightarrow \forall p \in s_i, \phi(\ldots, p, \ldots),$$

où $(x_1, \ldots, x_n) \in s_1 \times \ldots \times s_n$, et $\{x'_1, \ldots, x'_k\} := \{x_1, \ldots, x_n\} \cap \mathscr{X}_{s_i}$.

Exemple 1.56 *Le **Principe de Peano** est une instance de la récurrence structurelle où la sorte est nat et les constructeurs sont 0 et S. Soit la clause égalitaire $even(add(x, x)) = true$. Le schéma de récurrence utilisé dans la preuve par récurrence structurelle de l'**Introduction** est le suivant :*

$(even(add(0, 0)) = true) \wedge\ (\forall x \in nat, even(add(x, x)) = true \Rightarrow even(add(S(x), S(x))) = true)$
$\Rightarrow \forall x \in nat, even(add(x, x)) = true.$

B. Récurrence basée sur les clauses égalitaires

Le raisonnement par récurrence basée sur les clauses égalitaires consiste à appliquer la récurrence noethérienne sur un ensemble de clauses égalitaires.

Définition 1.57 (Récurrence basée sur les clauses égalitaires) *Si $\mathcal{E}$ un ensemble clauses égalitaires muni d'un ordre $<_c$ alors :*

$$(\forall \psi \in \mathcal{E}, (\forall \delta \in \mathcal{E}, \delta <_c \psi \Rightarrow \delta) \Rightarrow \psi) \Rightarrow \forall \phi \in \mathcal{E}, \phi.$$

Cette instance est dérivée du principe général de la récurrence nothérienne avec $\phi(x) = x, \forall x \in \mathcal{E}$.

Contrairement à la récurrence basée sur les termes, la récurrence basée sur les clauses ne définit pas un schéma de récurrence. Son application sur un ensemble de clauses muni d'un ordre bien fondé permet l'utilisation de n'importe quelle clause en tant qu'hypothèse par récurrence. Cet ensemble de clauses peut contenir aussi bien des clauses initiales à prouver que d'autres clauses générées au cours de la preuve.

Avantages Ce type de récurrence effectue naturellement la récurrence paresseuse et mutuelle. En effet, elle ne définit les hypothèses de récurrence qu'au moment de leur emploi, et elle autorise l'utilisation de toutes les clauses de la preuve comme hypothèses de récurrence.

Inconvénients L'inconvénient majeur de ce type de récurrence est la définition d'un ordre global sur l'ensemble des clauses de la preuve.

Parmi les instances de ce principe, nous nous intéresserons à la récurrence implicite.

B.a. Récurrence implicite La récurrence implicite s'applique à l'ensemble $\mathcal{E}$ contenant toutes les instances de toutes les clauses contenues par une preuve. Le principe de la récurrence réside dans la possibilité d'utiliser n'importe quelle instance de clause $\phi \in \mathcal{E}$ comme hypothèse de récurrence à une autre de $\psi \in \mathcal{E}$ tant que $\phi <_c \psi$ suivant l'ordre global $<_c$ défini sur les clauses égalitaires de $\mathcal{E}$ [Kounalis et Rusinowitch1990].

Définition 1.58 (Règle d'inférence, système d'inférence et preuve réductibles) *Une règle d'inférence* ***réductible*** *est une règle d'inférence dont les nouvelles conjectures générées dans une étape de preuve sont inférieures ou égales à des instances de conjectures de l'étape courante suivant un ordre global défini sur toutes ses clauses. Un* ***système d'inférence réductible*** *est un système d'inférence dont toutes les règles sont réductibles. Une* ***preuve réductible*** *est une preuve construite par un* ***système réductible****.*

Définition 1.59 (Récurrence implicite) *Si $\mathcal{E}$ un ensemble d'instance des clauses égalitaires d'une preuve réductible alors :*

$$(\forall \psi \in \mathcal{E}, (\forall \delta \in \mathcal{E}, \delta <_c \psi \Rightarrow \delta) \Rightarrow \psi) \Rightarrow \forall \phi \in \mathcal{E}, \phi.$$

Exemple 1.60 *Soit $\mathcal{E}_e$ l'ensemble qui contient toutes les instances closes de toutes les égalités de la preuve par récurrence implicite de la* ***Figure 2****. Nous considérons que l'ordre de récurrence est l'ordre $<_c$ défini dans l'****Exemple 1.50****. Cette preuve est réductible parce qu'elle est construite par le système d'inférence réductible de Spike. Ce système et les arguments de sa réductibilité seront détaillés dans le chapitre suivant. Toutefois, nous remarquons la réductibilité dans les différentes étapes de la preuve. Par exemple, dans la première étape les nouvelles conjectures générées :*

$$(e'_1) : even(0) = true \text{ et } (e'_2) : even(S(add(x, S(x)))) = true$$

*sont plus petites que $(e) : even(add(x, x)) = true$. En effet, comme nous avons montré dans l'****Exemple 1.48****, nous avons $\{even(0), true\} <<_{rpo} \{even(add(x, x)), true\}$. En suivant les mêmes étapes que l'****Exemple 1.48****, nous pouvons montré aussi que $\{even(S(add(x, S(x)))), true\} <<_{rpo} \{even(add(x, x)), true\}$.*
Ainsi, nous remarquons que le raisonnement par récurrence implicite figure à l'avant dernière étape de la preuve. En effet, la simplification de la conjecture

$$(e'_{19}) : \forall x, even(add(x, x)) = false \Rightarrow false = true$$

*s'effectue à l'aide de l'hypothèse de récurrence qui consiste à (e) à renommage prés. Cette simplification est correcte parce que on a $e <_c e'_{19}$ montré dans l'****Exemple 1.50****.*

1.4.3 Exécution de la récurrence paresseuse et mutuelle par la récurrence explicite

L'étude du lien entre la récurrence implicite et la récurrence explicite, ou éventuellement entre leurs instances, paraît très intéressante. Généralement, les instances de ces deux raisonnements ne coexistent pas dans le même environnement. La motivation principale de cette étude est l'imagination d'un environnement formel qui permet de raisonner par récurrence sur des structures mutuellement définies de façon paresseuse et mutuelle en relaxant les contraintes supplémentaires dues à la définition d'ordre de récurrence. Des efforts considérables sont fournis pour rassembler ces deux classes de raisonnement dont nous évoquerons quelques-uns d'entre eux.

A. Calcul des hypothèses de récurrence au cours de la preuve

Protzen [Protzen1994] a proposé une stratégie qui permet de raisonner par récurrence paresseuse dans une classe de preuves par récurrence explicite. Contrairement à la récurrence structurelle, les hypothèses de récurrence seront calculées et générées `à la volée` au fur et à mesure pendant la construction automatique de la preuve. Néanmoins, cette méthode garde la même analyse de récurrence [13] pour déterminer les hypothèses comme dans la récurrence structurelle. La correction de la récurrence dans cette méthode est déduite à partir des descriptions des relations bien fondées calculées à partir des définitions des fonctions. Cette approche a été intégrée dans le système INKA [Hutter et Sengler1996] pour permettre la construction automatique des preuves par récurrence paresseuse, mais, elle reste une récurrence explicite tant que ces hypothèses de récurrence sont des instances de la même conjecture.

B. Transformation d'une sous-classe de preuves par récurrence implicite en des preuves par récurrence explicite

Courant [Courant1996] a identifié une classe de preuves de récurrence implicite qui peuvent être reconstruites dans des preuves par récurrence explicite. La méthode a été utilisée la première fois par Courant pour créer une sorte de coopération entre Spike et Coq dans le but d'effectuer des preuves de spécifications booléennes définies sur des entiers naturels. Ces travaux sont suivis par Kaliszyk [Kaliszyk2005] qui a développé un mécanisme permettant de traduire une classe de preuves basées sur la récurrence implicite de Spike en scripts Coq contenant des preuves basées sur la récurrence explicite. Cette technique est restreinte aux spécifications dont les hypothèses sont utilisées uniquement dans les preuves de conjectures pour lesquelles elles sont introduites. Cette méthode ne s'applique donc pas aux preuves traitant des fonctions mutuellement définies. En plus, elle applique la récurrence basée sur l'ensemble couvrant à des conjectures qui instancient deux variables aux maximum.

C. Intégration de la récurrence basée sur les termes dans Coq

Récemment, Voicu et Li [Voicu et Li2009] ont proposé une tactique en Coq qui identifie les sous-buts répétés dans une preuve par l'analyse du terme de preuve pour les utiliser comme hypothèses de récurrence. Pendant l'analyse, la tactique construit une fonction récursive et un schéma de récurrence. En rencontrant un sous-but qui correspond à une instance de but initial, la tactique recommence la preuve en introduisant cette instance comme hypothèse de récurrence. La tactique reste limitée aux sous-buts déjà visités par la preuve et qui dérivent de la même proposition.

13. *Recursion Analysis*

Conclusion

Dans ce chapitre, nous avons introduit les notions de base nécessaires pour nos travaux telles que l'algèbre de termes, la conséquence inductive et l'ordre noethérien. Aussi, nous avons défini le raisonnement par récurrence ainsi que ses instances dans la logique de premier ordre et particulièrement notre concept clé la récurrence implicite. Dans la suite, nous présenterons le système d'inférence de Spike basé sur ce raisonnement.

Chapitre 2

Spike : un démonstrateur par récurrence implicite

Sommaire

Spike [Bouhoula *et al.*1995, Rusinowitch *et al.*2003, Stratulat2008] est un démonstrateur des théorèmes automatique qui peut raisonner par récurrence implicite sur des spécifications conditionnelles. Outre la récurrence implicite, Spike intègre la subsomption et la déduction équationnelle, la réécriture et la surréduction. L'aspect automatique de Spike a permis son utilisation dans des applications de taille industrielle. Il a permis d'établir la preuve de conformité de protocole ABR automatiquement alors qu'elle était auparavant effectuée soit manuellement ou soit par interaction avec PVS [Rusinowitch *et al.*2003]. Spike a également permis de valider automatiquement la plateforme JavaCard qui possède un langage de bas niveau utilisé pour la programmation des cartes à puces [Barthe et Stratulat2003].

Dans ce chapitre, nous présenterons d'abord le système d'inférence de Spike en tant qu'une instance du système abstrait et correct A [Stratulat2000, Stratulat2001]. Ensuite, nous illustrerons par un exemple la construction d'une preuve de Spike pour une spécification donnée. Enfin, nous discuterons la correction des preuves générées par Spike et nous présenterons un exemple de validation d'une preuve.

2.1 Système d'inférence abstrait A

Le système A implémente la récurrence implicite en définissant à chaque étape de la preuve l'information qui peut être utilisée en tant qu'hypothèse de récurrence. Comme, il fait aussi une abstraction des techniques de raisonnement utilisés dans la construction des nouvelles conjectures.

2.1.1 Définition

Le système d'inférence A est réductible et ses règles d'inférence doivent permettre l'utilisation des instances de conjectures déjà traitées dans la preuve. Pour ces raisons, nous :

- adoptons la définition d'une règle d'inférence qui permettra de sauvegarder l'ensemble des conjectures déjà traitées tout au long de la preuve, et
- introduisons la notion de l'ensemble couvrant contextuel qui permettra d'établir la propriété de réductibilité.

Définition 2.1 (Règle d'inférence) *[Stratulat2000] Une règle d'inférence du système A a la forme suivante :*

$$Nom\_règle : (E, H) \vdash^{A} (E', H'), [Conditions].$$

E et E' sont des ensembles de conjectures à prouver et, H et H' sont des ensembles de conjectures déjà traitées appelées prémisses. Les conditions expriment des propriétés sur des conjectures ou des sous-ensembles des conjectures de E, E', H et H'.

Cette définition d'une règle d'inférence permet d'exprimer le fait suivant : pour prouver les conjectures de E il suffit de prouver les conjectures de E' sachant que la preuve de E (respectivement de E') peut utiliser les prémisses de H (respectivement de H').

Un ensemble couvrant contextuel d'une clause égalitaire c est un ensemble fini des clauses égalitaires qui décrivent d'une façon finie toutes les instances closes de c.

Notation 2.2 *Dans la suite nous distinguerons les ensembles de clauses égalitaires suivants :*

- *Ax un ensemble d'axiomes,*

- *L un ensemble de lemmes -des clauses déjà prouvées avant de commencer la preuve-, et*
- *HIs un ensemble d'hypothèses de récurrence -des clauses de la preuve-.*

Dans la suite, nous désignons par la notation suivante $\Phi_{\leq_\phi}$ (respectivement $\Phi_{<_\phi}$) le sous ensemble de l'ensemble des clauses égalitaires de Φ qui contient uniquement les clauses égalitaires qui sont plus petites (respectivement plus petites ou égales) que la clause ϕ.

Définition 2.3 (Contexte et ensemble couvrant contextuel (strict)) *[Stratulat2000]*

*Soient Ax, ζ^1, ζ^2, Φ et Ψ des ensembles de clauses égalitaires de $CE(\Sigma, \mathscr{X})$ et $\leq_c$ un ordre partiel et noethérien sur $CE(\Sigma, \mathscr{X})$ et stable par substitution. On appelle $\zeta := (\zeta^1, \zeta^2)$ un **contexte**. On dit que Ψ **couvre contextuellement** Φ dans le contexte ζ si et seulement si*

$$\forall \phi \in \Phi, \forall \sigma \in Sub(\mathscr{T}(\Sigma)), Ax \cup \zeta^1_{\leq_{\phi\sigma}} \cup \zeta^2_{<_{\phi\sigma}} \cup \Psi_{\leq_{\phi\sigma}} \models_{ini} \phi\sigma.$$

- *Si $\Phi := \{\phi\}$, alors Ψ est un **ensemble couvrant contextuel** de ϕ.*
- *Si on a $\Psi_{<_{\phi\sigma}}$ à la place de $\Psi_{\leq\phi\sigma}$, alors Ψ est un **ensemble couvrant contextuel strict** de ϕ.*
- *Si $\Psi := \emptyset$, alors Ψ un ensemble couvrant contextuel vide.*

La relation de couverture contextuelle est réflexive, transitive et préservée par l'augmentation du contexte. La propriété de transitivité peut être généralisée à des relations de couverture contextuelle définies par des contextes différents.

AddPremise : $(E \cup \{\phi\}, H) \vdash^A (E \cup \Phi, H \cup \{\phi\})$ où :
- $\Phi := \bigcup_{i=1}^n \Phi_i$, où chaque ϕ_i
- Φ_i est un ensemble couvrant contextuel strict de ψ_i dans le contexte $(H, E \cup \{\phi\} \cup (\Phi \backslash \Phi_i))$,
- $\psi_i \in \Psi$ et $\Psi := \cup_{i=1}^n \{\psi_i\}$ est un ensemble couvrant contextuel de ϕ dans le contexte $(H, E \cup \Phi)$.

Simplify : $(E \cup \{\phi\}, H) \vdash^A (E \cup \Phi, H)$ `si` Φ est un ensemble couvrant contextuel de ϕ dans le contexte $(E \cup H, \emptyset)$.

FIGURE 2.1 – Le système d'inférence abstrait A [Stratulat2000]

La règle **AddPremise** remplace une conjecture par son ensemble couvrant contextuel strict. Cet ensemble est calculé en deux étapes : i) on calcule d'abord un ensemble couvrant contextuel de la conjecture courante, qu'on le note Ψ dans la **Figure 2.1**, à partir des axiomes et dans un contexte construit des conjectures à prouver et des prémisses, ii) ensuite on calcule un ensemble couvrant contextuel pour chaque clause de Ψ à partir des axiomes et dans un contexte construit des conjectures à prouver, des prémisses et des ensembles couvrants contextuels des clauses de l'ensemble Ψ calculé pendant l'étape i). Grâce à la généralisation de la propriétés de transitivité, l'ensemble couvrant contextuel d'une clause courante ϕ calculé dans l'étape ii) est un ensemble couvrant contextuel de ϕ. Enfin, la règle **AddPremise** ajoute cette clause à l'ensemble des clauses déjà traitées afin d'être utilisée comme une hypothèses de récurrence. Néanmoins, la

règle **Simplify** remplace une conjecture par son ensemble couvrant contextuel sans la rajouter à l'ensemble des conjectures déjà traitées. Si la règle **Simplify** engendre un ensemble vide alors la clause ϕ est une redondance et elle sera éliminée. Ce cas de la la règle **Simplify** correspond à la règle **Delete** définie comme suit :

Delete : `(Simplify)` $(E \cup \{\phi\}, H) \vdash^A (E, H)$, si Φ est vide.

2.1.2 Correction

Définition 2.4 (théorème de A et A-preuve) *[Stratulat2000] Soient Ax et E^0 deux ensembles de clauses égalitaires. L'ensemble E^0 sont des théorèmes par rapport à l'ensemble d'axiomes Ax s'il existe une A-dérivation finie de la forme :*

$$D := (E^0, \emptyset) \vdash^A (E^1, H^1) \vdash^A \ldots \vdash^A (E^{n-1}, H^{n-1}) \vdash^A (\emptyset, H^n), n \geq 1.$$

La dérivation D est une A-preuve de E^0.

Théorème 2.5 (Correction du système A) *[Stratulat2000] Soient Ax et E^0 deux ensembles de clauses égalitaires avec Ax étant des axiomes et E^0 étant des conjectures à prouver.*

Si $(E^0, \emptyset) \vdash^A (E^1, H^1) \vdash^A \ldots \vdash^A (E^{n-1}, H^{n-1}) \vdash^A (\emptyset, H^n)$ en utilisant Ax alors $Ax \models_{ini} E^0$.

Prouvé correct, le système abstrait A garantit que toute preuve construite par les règles d'inférence du système A est une preuve correcte. Donc, toute instance de A est correcte [Stratulat2000].

2.2 Système d'inférence de Spike

Dans cette sous-section, nous définirons tout d'abord les techniques de raisonnement intégrées dans Spike comme : la réécriture, la surréduction [14], l'analyse par cas, l'élimination par tautologie et par subsomption et la déduction équationnelle. Ensuite, nous présenterons une version simplifiée de système d'inférence de Spike. Enfin, nous discuterons la correction d'une preuve construite avec ce système.

2.2.1 Techniques et raisonnements intégrés dans Spike

A. Réécriture

La réécriture des termes est une branche des sciences informatiques théoriques qui combine des éléments de la logique, de l'algèbre universelle, de la théorie de preuve automatique et de la programmation fonctionnelle. Elle fait partie des outils indispensables dans la construction d'un assistant de preuve. En se basant sur la logique équationnelle, la réécriture des termes devient un moyen très efficace pour effectuer un raisonnement sur des équations. Dans nos travaux, elle nous permettra de décider l'égalité entre deux termes.

Nous considérons qu'une règle de réécriture se définit par l'orientation d'une égalité. Cette orientation s'effectue suivant un ordre de réduction défini sur les termes. Nous considérons le même ordre défini dans la **Section 1.3**.

14. narrowing

Définition 2.6 (Règles de réécriture et système de réécriture conditionnelle)
- *Une* ***règle de réécriture non-conditionnelle*** *ayant la forme $l \to r$, dérive d'une équation non conditionnelle de la forme $l = r$ avec l un terme non variable, $Var(r) \subseteq Var(l)$ et $r <_{rpo} l$.*

- *Une* ***règle de réécriture conditionnelle****, ayant la forme $l_1 = r_1 \wedge \ldots \wedge l_n = r_n \Rightarrow l_{n+1} \to r_{n+1}$, dérive d'une équation conditionnelle de la forme $l_1 = r_1 \wedge \ldots \wedge l_n = r_n \Rightarrow l_{n+1} = r_{n+1}$, avec l_{n+1} un terme non variable, $\forall i \in [1..n]$, $Var(r_{n+1}) \cup Var(l_i) \cup Var(r_i) \subseteq Var(l_{n+1})$, $r_{n+1} <_{rpo} l_{n+1}$ et $\uplus_{i=1}^{n}\{l_i, r_i\} <<_{rpo} \{l_{n+1}\}$.*

Un ***système de réécriture conditionnelle****, noté par R, est un ensemble de règles de réécriture non-conditionnelles ou conditionnelles.*

Un système de réécriture induit une relation de réécriture non-conditionnelle définit comme suit :

Définition 2.7 (Relation de réécriture non-conditionnelle) *Soient R un système de réécriture, $s \to t$ une règle de réécriture non conditionnelle de R et $u[s\sigma]$ un terme qui contient l'instance $s\sigma$ à la position p. L'application de $s \to t$ sur le terme u consiste à remplacer $s\sigma$ par $t\sigma$ à la position p et on écrit :*

$$u[s\sigma]_p \to_R u[t\sigma]_p.$$

Nous notons par $\to_R^*$ la fermeture transitive de $\to_R$. Nous définissons certaines propriétés d'une relation de réécriture telles que la réductibilité, la forme normale, la joignabilité et la terminaison.

Définition 2.8 (Réductibilité, forme normale, joignabilité et terminaison) *Soit R un système de réécriture et s et t deux termes.*
- *t est* ***réductible*** *par R, s'il existe un terme t' tel que $t \to_R t'$.*
- *t est en* ***forme normale****, ou irréductible par R, s'il n'existe aucun terme t' tel que $t \to_R t'$.*
- *s et t sont* ***joignables****, s'il existe un terme z tel que $s \to_R^* z$ et $t \to_R^* z$ et on écrit $t \downarrow_R s$.*
- *$\to_R$ se* ***termine*** *s'il n'existe aucune séquence infiniment décroissante :*
  $t_0 \to_R t_1 \to_R t_2 \to_R \ldots$.

La relation de réécriture $\to_R$ permet de *décider l'égalité entre deux termes en les réduisant en une forme normale unique*. Afin de garantir la décidabilité de l'égalité, la relation de réécriture ainsi le système de réécriture doivent satisfaire certaines critères détaillés dans la suite.

Définition 2.9 (Confluence sur les termes, confluence et convergence sur les termes clos) *Une relation de réécriture $\to_R$ est* ***confluente****, respectivement* ***confluente sur les termes clos****, si et seulement si pour tous termes $t_1, t_2, t_3 \in \mathscr{T}(\Sigma, \mathscr{X})$, respectivement pour tous termes clos $t_1, t_2, t_3 \in \mathscr{T}(\Sigma)$, lorsqu'on a $t_1 \to_R^* t_2$ et $t_1 \to_R^* t_3$ alors $t_2 \downarrow_R t_3$.*
Une relation de réécriture qui est confluente sur les termes clos et qui se termine est une relation de réécriture ***convergente sur les termes clos****.*

La terminaison d'une relation de réécriture garantit l'existence d'une forme normale d'un terme clos. Sa confluence sur les termes clos garantit l'unicité de la forme normale. De ce fait, nous considérons dans la suite que des systèmes de réécriture convergents sur les termes clos.

B. Analyse par cas

L'analyse par cas est une simplification d'une clause égalitaire par un ensemble des clauses égalitaires. Cette technique telle qu'elle est définie dans [Stratulat2000] est combinée avec la réécriture conditionnelle.

Définition 2.10 (Analyse par cas) *[Stratulat2000] Soient un ensemble de clauses égalitaires Ax, R un système de réécriture induit par Ax, $c[s]$ une clause égalitaire et R' un sous-ensemble de règles de réécriture conditionnelles de R.*

$$R' := \cup_{i=1}^{n}\{P_i \Rightarrow l_i \rightarrow r_i\},$$

tel que pour chaque $i \in [1..n]$ il existe une position p_i et une substitution σ_i pour laquelle $s|_{p_i} = l_i\sigma_i$.

*Alors, l'**analyse par cas** de $c[s]$ donne l'ensemble AC_c se définit comme suit :*

$$AC_c = \{P_i\sigma_i \Rightarrow c[s[r_i\sigma_i]_{p_i}], \forall i \in [1..n]\} \text{ si } Ax \models_{ini} \vee_{i=1}^{n} P_i\sigma_i.$$

Notons que pour une clause $c[s]$ donnée, le sous-ensemble des règles de réécriture $R' := \cup_{i=1}^{n}\{P_i \Rightarrow l_i \rightarrow r_i\}$ sont des règles de réécriture de R dont l_i est unifiable avec le sous-terme s pour tout $i \in [1..n]$.

Afin de garantir la complétude de l'analyse par cas, le système de réécriture R doit être fortement complet. Cette propriété est exprimée à l'aide de la complétude suffisante.

Définition 2.11 (terme constructeur et complétude suffisante) *Un **terme constructeur** t est un terme dont tous les symboles de fonctions sont des constructeurs. L'ensemble des termes constructeurs est noté par $T(\Sigma(\mathscr{S}, C), \mathscr{X})$.*

*Un symbole de **fonction** f est **suffisamment complet** par rapport à un système de réécriture R si et seulement si pour tout $t \in \mathscr{T}(\Sigma), t := f(t_1, \ldots, t_n), \{t_1, \ldots, t_n\} \subset \mathscr{T}(\Sigma)$ et s'il existe un **terme constructeur clos** $t' \in T(\Sigma(\mathscr{S}, C))$ tel que $f(t_1, \ldots, t_n) \rightarrow_R^* t'$.*

Maintenant, nous définissons la propriété de complétude forte des systèmes de réécriture.

Définition 2.12 (Complétude forte) *Soient $f \in \mathscr{F}$ un symbole de fonction suffisamment complet et R un système de réécriture déduit à partir des axiomes Ax. On dit que f est **fortement complet** par rapport à R si, pour toutes les règles $P_i \Rightarrow f(t_1, \ldots, t_n) \rightarrow d_i$ dont les membres gauches sont identiques à un renommage près des variables de μ_i, on a $Ax \models_{ini} \vee_i(P_i\mu_i)$. On dit aussi que R est système fortement complet si tous les symboles de fonction de $\mathscr{F}$ sont fortement complets.*

Dans la suite, nous considérons que outre la confluence sur les termes clos les systèmes de réécriture sont fortement complets.

C. Surréduction

La technique de surréduction ou le narrowing est une réécriture des termes basée sur l'unification [Baader et Nipkow1998]. Dans nos travaux cette technique est utilisée pour

calculer un schéma d'instantiation des variables d'une clause. Ce schéma est calculé à partir d'unification d'un sous-terme de la clause avec les règles de réécriture d'un système de réécriture R. Nous supposons que tous les termes de côté gauche de toutes les règles de R sont des **termes de base**, i.e. des termes ayant la forme suivante $f(c_1, \ldots, c_n)$ avec $f \in DF$ et $\forall i \in [1..n], c_i \in T(\Sigma(\mathscr{S}, C), \mathscr{X})$. L'instantiation s'effectue de la façon suivante : soit $e[s[t]_p]$ une clause égalitaire contenant un sous-terme t. Le sous-terme est testé pour unification avec tous les termes de côté gauche de toutes les règles de réécriture de R. À chaque fois le *mgu* est utilisé comme une **substitution test**. Nous avons déjà supposé que le système R est fortement complet donc l'ensemble des substitutions test noté par $TS(c, R)$ est complet. Nous obtenons :

$$Ax \models_{ini} e[s[t]_p] \text{ si } Ax \models_{ini} \bigcup_{\sigma} E_\sigma$$
$$E_\sigma := \{P\sigma \Rightarrow e\sigma[s\sigma[r\sigma]_p], P \Rightarrow l \rightarrow r \in R, \sigma := mgu(l, t)\}.$$

Notons que la surréduction s'effectue en deux étapes. Chacune engendre un ensemble couvrant contextuel :

i) L'unification engendre l'ensemble Ψ qui contient toutes les instances de la clause e obtenues par instanciation de e en utilisant les substitutions test. L'ensemble Ψ est un ensemble couvrant contextuel de e.

ii) La réduction de chaque clause de Ψ engendre un ensemble Φ avec $\forall \phi \in \Phi$ où $\{\phi\}$ est un ensemble couvrant contextuel d'une clause de Ψ.

D. Élimination des redondances

L'ensemble de conjecture à prouver peut contenir des clauses qui sont redondantes. L'élimination de ces clauses n'influence pas la correction de la preuve. Les tautologies sont reconnues d'une façon syntaxique voir la **Définition 1.10**. Les autres redondances sont éliminées à l'aide de la technique de subsomption. Elle consiste à supprimer une clause subsumée par une autre qui est un lemme ou une hypothèse de récurrence.

Définition 2.13 (Sous-clause et subsomption clausale syntaxique) *[Stratulat2000]*
Soient deux clauses c_1 et c_2. On dit que $c_1 := (\{a_1, \ldots, a_n\}, \{b_1, \ldots, b_m\})$ est une sous-clause de $c_2 := (\{a'_1, \ldots, a'_n\}, \{b'_1, \ldots, b'_m\})$ si $\{a_1, \ldots, a_n\} \setminus\setminus \{a'_1, \ldots, a'_n\} := \emptyset$ et $\{b_1, \ldots, b_m\} \setminus\setminus \{b'_1, \ldots, b'_m\} := \emptyset$. On dit que c_1 subsume c_2 s'il existe une substitution σ telle que $c_1\sigma$ est une sous-clause de c_2.

E. Décomposition

La décomposition s'applique lorsque les deux termes d'une égalité ont le même symbole d'entête qui est un constructeur libre.

– *La décomposition positive* :
Soit e une clause ayant la forme suivante :
$$c(s_1, \ldots, s_n) = c(t_1, \ldots, t_n) \vee r, \text{ tel que } c \in C \text{ et libre.}$$
On a $Ax \models_{ini} e$ si pour tout $i \in [1..n]$ on $Ax \models_{ini} (s_i = t_i) \vee r$.

– *La décomposition négative* :
Soit e une clause ayant la forme suivante :
$$\neg(c(s_1, \ldots, s_n) = c(t_1, \ldots, t_n)) \vee r, \text{ tel que } c \in C \text{ et libre.}$$
On a $Ax \models_{ini} e$ si pour tout $i \in [1..n]$ on $Ax \models_{ini} \vee_{i=1}^{n} \neg(s_i = t_i) \vee r$.

2.2.2 Version simplifiée de système d'inférence de Spike

Generate : $(E \cup \{c\}, H) \vdash (E \cup \bigcup\{E_\sigma, \sigma \in TS(c, R)\}, H \cup \{c\})$.

[Unconditional] Rewriting : $(E \cup \{e\}, H) \vdash (E \cup \{e'\}, H)$ `si` $e \rightarrow_{A_x \cup HIs \cup L} e'$, où $HIs := \{e_{hi} | e_{hi} \in E \cup H \wedge e_{hi} <_c e\}$.

Total Case Rewriting : $(E \cup \{c\}, H) \vdash (E \cup AC_c, H)$, où AC_c est l'ensemble de clause retourné par l'analyse par cas en utilisant $R' := \{P_1 \wedge a = b \wedge P_2 \Rightarrow e\}, b \in \{true, false\}$.

Subsumption : $(E \cup \{\phi\}, H) \vdash (E, H)$, où $\exists \phi' \in E \cup H \cup L$ telle que ϕ est subsumée par ϕ'.

Tautology : $(E \cup \{\phi\}, H) \vdash (E, H)$, où ϕ est une tautologie.

Positive decomposition : $(E \cup \{(c(s_1, \ldots, s_n) = c(t_1, \ldots, t_n)) \bigvee r\}, H) \vdash (E \cup \{\bigvee_{i=1}^{n}(s_i = t_i) \bigvee r\}, H)$, où c un constructeur libre.

Negative decomposition : $(E \cup \{\neg(c(s_1, \ldots, s_n) = c(t_1, \ldots, t_n)) \bigvee r\}, H) \vdash (E \cup \{\bigvee_{i=1}^{n} \neg(s_i = t_i) \bigvee r\}, H)$, où c un constructeur libre.

FIGURE 2.2 – Une version simplifiée du système d'inférence de Spike

Soit la version simplifiée du système d'inférence de Spike décrite dans la **Figure 2.2**. La règle Generate remplace d'une part la clause c par son ensemble couvrant contextuel strict contenant toutes les instances de c calculées par la technique de surréduction. D'autre part, elle ajoute cette clause à l'ensemble des conjectures à prouver. La règle Rewriting réécrit une égalité avec des règles de réécriture non-conditionnelle de Ax, des lemmes L ou des hypothèses de récurrence HIs qui résident dans des instances de prémisses plus petites que la conjecture courante. La règle Total Case Rewriting remplace une clause par l'ensemble engendrée par une analyse telle que c'est décrit dans **Définition 2.10**. Notons que dans nos travaux nous considérons la même restriction de [Stratulat2010] : $R' := \{P_1 \wedge a = b \wedge P_2 \Rightarrow e\}, b \in \{true, false\}$. Comme leurs noms indiques, les règles Tautologie et Subsumption permet d'éliminer respectivement des tautologies et les clauses subsumées par d'autres appartenant aux lemmes ou aux hypothèses de récurrence. Les règles Positive decomposition et Negative decomposition effectuent respectivement la décomposition positive et négative.

2.2.3 Correction

La preuve de correction de système présenté dans la **Figure 2.2** consiste à prouver qu'il est une instance du système abstrait correct A. En effet, l'instanciation telle qu'elle présentée dans [Stratulat2000] s'effectue à l'aide des modules de raisonnements. Ces modules mettent en oeuvre

les techniques et les raisonnements présentés auparavant. Dans la suite, nous présenterons le concept de module de raisonnement. Et, nous montrerons la correction du système de la **Figure 2.2** par instanciation du système correct A.

A. Modules de raisonnement

Les modules de raisonnement engendrent des ensembles couvrants contextuels élémentaires.

Définition 2.14 (Modules de raisonnement inconditionnel) *[Stratulat2000] Un* ***module de raisonnement inconditionnel****, $\mathscr{MR}$, est une fonction partielle $G_{\mathscr{MR}}$ ayant le même profil suivant : $L \times (\mathcal{P}(L) \times \mathcal{P}(L)) \rightarrow \mathcal{P}(L)$* [15]*. Elle prend une clause et un contexte et retourne un ensemble de clauses et génère l'ensemble couvrant contextuel de cette clause. On dit que M est applicable à ϕ dans le contexte $\zeta := (\zeta^1, \zeta^2)$*

$$Ax \cup \zeta^1_{\leq \phi\sigma} \cup \zeta^2_{<\phi\sigma} \cup (G_{\mathscr{MR}}(\phi, \zeta))_{\leq \phi\sigma} \models_{ini} \phi\sigma.$$

Exemple 2.15 *Soient les modules de raisonnement correspondants aux techniques et raisonnements définis auparavant.*

`Réécriture` : **([Unconditional] Rewriting)** *soit $\mathscr{MR}_R$ le module de raisonnement de réécriture tel que étant donnés une clause e, un système de réécriture R, un ensemble des lemmes L et un ensemble des hypothèses de récurrence HIs on a :*

$$G_{\mathscr{MR}_R} := \{e'\}, e \rightarrow_{l \rightarrow r \in R \cup L \cup HIs} e'.$$

`Analyse par cas` : **(Total Case Reewriting)** *soit $\mathscr{MR}_{AC}$ le module de raisonnement de l'analyse par cas tel que étant donnés une clause $e[s]$ et $\{P_1 \wedge a = true \wedge P_2 \Rightarrow l \rightarrow r_1, P_3 \wedge a = false \wedge P_4 \Rightarrow l \rightarrow r_2\} \subseteq R$ un système de réécriture on a :*

$$G_{\mathscr{MR}_{AC}} := \{P_1\sigma \wedge a\sigma = true \wedge P_2\sigma \Rightarrow e[r_1\sigma_1], P_3\sigma \wedge a\sigma = false \wedge P_4\sigma \Rightarrow e[r_2\sigma_2]\}$$

tel que σ_1 et σ_2 deux substitutions telles que $s := l\sigma_1$ et $s := l\sigma_2$.

`Surréduction` : **(Generate)** *soit $\mathscr{MR}_N$ le module de raisonnement de la surréduction tel que étant donnés une clause e et un système de réécriture R on a :*

$$G_{\mathscr{MR}_N} := \bigcup_\sigma \{E_\sigma, \sigma \in TS(e, R)\}.$$

Comme nous avons expliqués $G_{\mathscr{MR}_N}$ calculé par surréduction est calculé en deux étapes.

`Subsomption` : **(Subsumption)** *soit $\mathscr{MR}_S$ le module de raisonnement de la subsomption tel que étant donnés deux clause e et e', e est subsumé par e' on a :*

$$G_{\mathscr{MR}_S} := \emptyset.$$

15. $\mathcal{P}(L)$ désigne l'ensemble des parties de l'ensemble des formules L (*Power set*).

`Tautologie :` **(Tautology)** *soit $\mathscr{MR}_T$ le module de raisonnement qui correspond à la tautologie tel que étant donné une tautologie e on a :*

$$G_{\mathscr{MR}_T} := \emptyset.$$

`Décomposition positive` **(Positive Decomposition)** *soit $\mathscr{MR}_{PD}$ le module de raisonnement qui correspond à la décomposition positive tel que étant donné une clause $c(s_1, \ldots, s_n) = c(t_1, \ldots, t_n) \vee r$ on a :*

$$G_{\mathscr{MR}_{PD}} := \{\forall i \in [1..n], (s_i = t_i) \vee r\}.$$

`Décomposition négative` **(Negative Decomposition)** *soit $\mathscr{MR}_{ND}$ le module de raisonnement qui correspond à la décomposition positive tel que étant donné une clause e on a :*

$$G_{\mathscr{MR}_{ND}} := \{\forall i \in [1..n], \neg(s_i = t_i) \vee r\}.$$

B. Spike comme une instance du système A

Proposition 2.16 (Correction de système d'inférence Spike) *Le système d'inférence de Spike de la **Figure 2.2** est une instance du système abstrait A.*

`Preuve :` La preuve s'effectue par analyse de chacune des règles de Spike. Le tableau **Tableau 2.1** récapitule la preuve. Pour chaque règle on précise la règle abstraite, le raisonnement implémenté, le contexte et l'ensemble couvrant contextuel.

Règle de Spike	**Règle abstraite**	$\mathscr{MR}$	$\mathbf{G}_{\mathscr{MR}}$
Generate	AddPremise	$\mathscr{MR}_N$	$\bigcup\{E_\sigma, \sigma \in TS(e, R)\}$
Subsumption	Delete	$\mathscr{MR}_S$	$\emptyset$
Rewriting	Simplify	$\mathscr{MR}_R$	$\{e'\}$
Total Case Rewriting	Simplify	$\mathscr{MR}_{AC}$	$\{e', e''\}$
Tautology	Delete	$\mathscr{MR}_T$	$\emptyset$
Positive decomposition	Simplify	$\mathscr{MR}_{PD}$	$\{e'\}$
Negative decomposition	Simplify	$\mathscr{MR}_{ND}$	$\{e'\}$

TABLE 2.1 – Les règles de système de la **Figure 2.2** en tant qu'instances des règles de A

2.3 Preuves par récurrence implicite par Spike

Dans cette section, nous présenterons un exemple de spécification Spike. Ensuite, nous présenterons la stratégie de Spike pour construire une preuve. Enfin, nous discuterons tous les résultats possibles fournis par le démonstrateur des théorèmes Spike.

2.3.1 Exemple d'une spécification conditionnelle

Une spécification Spike contient principalement deux parties : une description du système (une spécification conditionnelle) et une formalisation des ses propriétés (les conjectures à prouver).

Elle peut également contenir des informations complémentaires qui serviront à la recherche de la preuve (comme la stratégie de preuve) et à la définition de l'ordre de récurrence (à l'aide d'une précédence sur les symboles de fonctions). Chaque spécification dans Spike possède un nom qui permet de gérer sa trace de preuve.

Définition 2.17 (Déclaration de signature)

Sortes : $sorts : Sort_1, Sort_2, \ldots Sort_n$

Fonctions : $nom\_fonction\_\_\ldots\_ : Sort_1 \rightarrow Sort_2 \rightarrow \ldots Sort_n$.
Le nombre des tirets bas suivant le symbole de la fonction correspond à l'arité.

Pour certaines spécifications, Spike peut déduire la précédence sur les symboles des fonctions de manière convenable. Cependant, pour d'autres spécifications, en particulier celles contenant des fonctions mutuellement définies, il est recommandé de fournir la précédence de manière explicite. La déclaration de la précédence regroupe les symboles de fonctions en symboles équivalents et en symboles ordonnés.

L'exemple suivant correspond à la spécification complète de la **Proposition 1** de l'**Introduction**

Exemple 2.18

specification : evenodd.

Cette commande permet d'associer le nom ***evenodd*** *à la spécification.*

sorts : bool nat ;

Cette ligne permet de définir les deux symboles de sortes : bool et nat.

constructors :
$true : \rightarrow bool;$
$false : \rightarrow bool;$
$0 : \rightarrow nat;$
$S\_ : nat \rightarrow nat;$

defined functions :
$even\_ : nat \rightarrow bool;$
$odd\_ : nat \rightarrow bool;$
$add\_\_ : nat\ nat \rightarrow nat;$

Les symboles des fonctions et des constructeurs sont déclarés séparément.

greater : even add S 0 true false ;

Cette commande déclare la précédence suivante : false $<$ *true* $<$ $0 < S <$ *add* $<$ *even.*

equiv : even odd ;

Cette commande déclare l'équivalence entre les symboles suivants : even et odd et permet de déduire la précédence suivante : false < true < 0 < S < add < odd.

axioms :
add (0, x) = x ; $ ***(A1)***
add (S (x), y) = S (add (x, y)) ; $ ***(A2)***

even (0) = true ; $ ***(A3)***
even (S(0)) = false ; $ ***(A4)***
odd (x) = true ⇒ even (S (S (x))) = false ; $ ***(A5)***
odd (x) = false ⇒ even (S (S (x))) = true ; $ ***(A6)***

odd (0) = false ; $ ***(A7)***
even (x) = true ⇒ odd (S (x)) = true ; $ ***(A8)***
even (x) = false ⇒ odd (S (x)) = false ; $ ***(A9)***

A1 *et* ***A2*** *décrivent la fonction* ***add****.* ***A3****,* ***A4****,* ***A5*** *et* ***A6*** *décrivent la fonction* ***even****. Respectivement,* ***A7****,* ***A8*** *et* ***A9*** *définissent la fonction* ***odd****.*

lemmas :
add (x, S (y)) = S (add (x, y)) ;

Ce lemme est nécessaire pour la preuve. Il est défini explicitement par l'utilisateur.

conjectures :
even (add (x, x)) = true ;

Cette conjecture exprime la proposition suivante : l'addition d'un entier à lui-même retourne toujours un naturel pair.

2.3.2 Stratégie de construction des preuves

Un utilisateur de Spike peut exprimer son intuition en proposant une stratégie de recherche de la preuve. La stratégie est une proposition d'une séquence d'application des règles d'inférence de Spike. Tout au long de la recherche de preuve, les règles seront appliquées suivant cette séquence. Spike s'arrête si l'ensemble des conjectures devient vide ou s'il n'existe plus aucune règle applicable. Ce dernier cas correspond à un contre exemple qui sera fourni à l'utilisateur.

Exemple 2.19
Strategy

$tautology\_rule = delete(id, [tautology]);$
$subsumption\_rule = delete(id, [subsumption(L|E)]);$
$decomposition\_rule = simplify(id, [negative\_decomposition]);$
$rewriting\_rule = simplify(id, [rewriting(rewrite, L|R|E, *)]);$
$total\_case\_rewriting\_rule =$
$simplify(id, [total\_case\_rewriting(simplify\_strat, r, *)]);$
$augment\_rule = simplify(id, [augmentation(L)]);$
$total\_case\_rewriting\_add\_premise\_rule =$
$add\_premise(total\_case\_rewriting(simplify\_strat, r, *), [id]);$

```
inst_var_rule = add_premise(generate, [id]);
stra = repeat (try (
                    tautology_rule,
                    decomposition_rule,
                    rewriting_rule,
                    subsumption_rule,
                    total_case_rewriting_add_premise_rule
                    ));
fullind = (repeat(stra, inst_var_rule), print_goals_with_history);
start_with : fullind
```

Les règles *tautology_rule* et *subsumption_rule* sont des instances de la règle abstraite Delete. Les règles *decomposition_rule*, *rewriting_rule*, *total_case_rewriting_rule* et *augment_rule* sont des instances de la règle abstraite Simplify. Et les règles *total_case_rewriting_add_premise_rule* et *inst_var_rule* sont des instances de la règle AddPremise. *stra* est la définition de la séquence d'application des règles qui sera suivie pendant la recherche de la preuve. *fullind* est la stratégie qui répète *inst_var_rule* jusqu'à trouver une preuve et finir par imprimer la trace de la preuve par la commande *print_goals_with_history*.

2.3.3 Résultats fournis par Spike

Spike essaie de trouver une preuve, une dérivation finie, à partir d'un état $(E, \emptyset)$ où E est l'ensemble des conjectures de la spécification. Quatre situations sont possibles :

- une preuve est trouvée ;
- un contre exemple est trouvé ;
- la recherche de la preuve s'arrête sans preuve ni contre-exemple
- la recherche de la preuve ne se termine pas.

Comme exemple, nous présenterons la preuve par récurrence implicite de la conjecture $even(add(x, x)) = true$ telle qu'elle est générée par le démonstrateur des théorèmes Spike.

Exemple 2.20

$\vdash (\{\underline{even(add(x, x)) = true}\}, \emptyset)$

$\vdash_{Generate} (\{\underline{even(0) = true}, even(S(add(x, S(x)))) = true\}, \{even(add(x, x)) = true\})$

$\vdash_{Rewriting} (\{\underline{true = true}, even(S(add(x, S(x)))) = true\}, \{even(add(x, x)) = true\})$

$\vdash_{Tautology} (\{\underline{even(S(add(x, S(x)))) = true}\}, \{even(add(x, x)) = true\})$

$\vdash_{Rewriting} (\{\underline{even(S(S(add(x, x)))) = true}\}, \{even(add(x, x)) = true\})$

$\vdash_{Total\ Case\ Rewriting} (\{\underline{odd(add(x, x)) = false \Rightarrow true = true}, odd(add(x, x)) = true \Rightarrow false = true\}, \{even(add(x, x)) = true\})$

$\vdash_{Tautology} (\{\underline{odd(add(x, x)) = true \Rightarrow false = true}\}, \{even(add(x, x)) = true\})$

$\vdash_{Generate} (\{\underline{odd(0) = true \Rightarrow false = true}, odd(S(add(x, S(x)))) = true \Rightarrow false = true\}, \{even(add(x, x)) = true, odd(add(x, x)) = true \Rightarrow false = true\})$

$\vdash_{Rewriting}$ $(\{\underline{false = true \Rightarrow false = true}, odd(S(add(x, S(x)))) = true \Rightarrow false = true\}, \{even(add(x, x)) = true, odd(add(x, x)) = true \Rightarrow false = true\})$

$\vdash_{Tautology}$ $(\{\underline{odd(S(add(x, S(x)))) = true \Rightarrow false = true}\}, \{even(add(x, x)) = true, odd(add(x, x)) = true \Rightarrow false = true\})$

$\vdash_{Rewriting}$ $(\{\underline{odd(S(S(add(x, x)))) = true \Rightarrow false = true}\}, \{even(add(x, x)) = true, odd(add(x, x)) = true \Rightarrow false = true\})$

$\vdash_{Total\ Case\ Rewriting}$ $(\{\underline{even(S(add(x, x))) = false \Rightarrow false = true \Rightarrow false = true}, even(S(add(x, x))) = true \Rightarrow true = true \Rightarrow false = true\}, \{even(add(x, x)) = true, odd(add(x, x)) = true \Rightarrow false = true\})$

$\vdash_{Tautology}$ $(\{\underline{even(S(add(x, x))) = true \Rightarrow true = true \Rightarrow false = true}\}, \{even(add(x, x)) = true, odd(add(x, x)) = true \Rightarrow false = true\})$

$\vdash_{Negative\ Decomposition}$ $(\{\underline{even(S(add(x, x))) = true \Rightarrow false = true}\}, \{even(add(x, x)) = true, odd(add(x, x)) = true \Rightarrow false = true\})$

$\vdash_{Generate}$ $(\{\underline{even(S(0)) = true \Rightarrow false = true}, even(S(S(add(x, S(x))))) = true \Rightarrow false = true\}, \{even(add(x, x)) = true, odd(add(x, x)) = true \Rightarrow false = true, even(S(add(x, x))) = true \Rightarrow false = true\})$

$\vdash_{Rewriting}$ $(\{\underline{false = true \Rightarrow false = true}, even(S(S(add(x, S(x))))) = true \Rightarrow false = true\}, \{even(add(x, x)) = true, odd(add(x, x)) = true \Rightarrow false = true, even(S(add(x, x))) = true \Rightarrow false = true\})$

$\vdash_{Rewriting}$ $(\{\underline{even(S(S(S(add(x, x))))) = true \Rightarrow false = true}\}, \{even(add(x, x)) = true, odd(add(x, x)) = true \Rightarrow false = true, even(S(add(x, x))) = true \Rightarrow false = true\})$

$\vdash_{Total\ Case\ Rewriting}$ $(\{\underline{odd(S(add(x, x))) = true \Rightarrow false = true \Rightarrow false = true}, odd(S(add(x, x))) = false \Rightarrow true = true \Rightarrow false = true\}, \{even(add(x, x)) = true, odd(add(x, x)) = true \Rightarrow false = true, even(S(add(x, x))) = true \Rightarrow false = true\})$

$\vdash_{Tautology}$ $(\{\underline{odd(S(add(x, x))) = false \Rightarrow true = true \Rightarrow false = true}\}, \{even(add(x, x)) = true, odd(add(x, x)) = true \Rightarrow false = true, even(S(add(x, x))) = true \Rightarrow false = true\})$

$\vdash_{Negative\ Decomposition}$ $(\{\underline{odd(S(add(x, x))) = false \Rightarrow false = true}\}, \{even(add(x, x)) = true, odd(add(x, x)) = true \Rightarrow false = true, even(S(add(x, x))) = true \Rightarrow false = true\})$

$\vdash_{Total\ Case\ Rewriting}$ $(\{\underline{even(add(x, x)) = true \Rightarrow true = false \Rightarrow false = true}, even(add(x, x)) = false \Rightarrow false = false \Rightarrow false = true\}, \{even(add(x, x)) = true, odd(add(x, x)) = true \Rightarrow false = true, even(S(add(x, x))) = true \Rightarrow false = true\})$

$\vdash_{Tautology}$ $(\{\underline{even(add(x, x)) = false \Rightarrow false = false \Rightarrow false = true}\}, \{even(add(x, x)) = true, odd(add(x, x)) = true \Rightarrow false = true, even(S(add(x, x))) = true \Rightarrow false = true\})$

$\vdash_{Rewriting}$ $(\{\underline{true = false \Rightarrow false = true}\}, \{even(add(x,x)) = true, odd(add(x,x)) = true \Rightarrow false = true, even(S(add(x,x))) = true \Rightarrow false = true\})$

$\vdash_{Tautology}$ $(\emptyset, \{even(add(x,x)) = true, odd(add(x,x)) = true \Rightarrow false = true, even(S(add(x,x))) = true \Rightarrow false = true\})$

2.4 Correction des preuves par récurrence implicite de Spike

Le système d'inférence de Spike est proposé comme une instance du système d'inférence abstrait A. Sa preuve de correction s'effectue de la même façon que la **Proposition 2.16**. Néanmoins, l'implémentation de Spike n'est jamais validée. De ce fait, Spike ne possède aucune garantie de la correction de ses preuves.

La taille de code source de Spike dépasse les 20 milles lignes de code [Spike Development Team]. Sa validation manuelle ou automatique sera une tâche difficile et très coûteuse voir impossible. Pour cela, il est plus convenable de valider les résultats de Spike au lieu de son code source.

La validation d'une preuve de Spike consiste à montrer que la preuve est correcte en validant l'application du principe de récurrence implicite sur l'ensemble des clauses égalitaire de la preuve

Exemple 2.21 *La validation de la preuve par récurrence implicite de l'**Exemple 2.20** consiste à vérifier que :*

$$(\forall \psi \in \mathcal{E}, (\forall \delta \in \mathcal{E}, \delta <_c \psi \Rightarrow \delta) \Rightarrow \psi) \Rightarrow \forall \phi \in \mathcal{E}, \phi.$$

Sachant que :

- $\mathcal{E} := \{even(add(x,x)) = true, even(0) = true, even(S(add(x,S(x)))) = true, true = true, even(S(S(add(x,x)))) = true, odd(add(x,x)) = false \Rightarrow true = true, odd(add(x,x)) = true \Rightarrow false = true, odd(0) = true \Rightarrow false = true, odd(S(add(x,S(x)))) = true \Rightarrow falsetrue, false = true \Rightarrow false = true, odd(S(S(add(x,x)))) = true \Rightarrow false = true, even(S(add(x,x))) = false \Rightarrow false = true \Rightarrow false = true, even(S(add(x,x))) = true \Rightarrow true = true \Rightarrow false = true, even(S(add(x,x))) = true \Rightarrow false = true, even(S(0)) = true \Rightarrow false = true, even(S(S(add(x,S(x))))) = true \Rightarrow false = true, false = true \Rightarrow false = true, even(S(S(S(add(x,x))))) = true \Rightarrow false = true, odd(S(add(x,x))) = true \Rightarrow false = true \Rightarrow false = true, odd(S(add(x,x))) = false \Rightarrow true = true \Rightarrow false = true, odd(S(add(x,x))) = false \Rightarrow false = true, even(add(x,x)) = true \Rightarrow true = false \Rightarrow false = true, even(add(x,x)) = false \Rightarrow false = false \Rightarrow false = true, true = false \Rightarrow false = true\}$
- $<_c := <<_{rpo}$, *i.e. l'ordre* $<_c$ *est l'extension multi-ensemble de l'ordre rpo défini sur les termes et défini à partir de la précédence sur les symboles de fonction suivante :*
- $<_F := even \leq odd < add < S < 0 < true < false$.

La validation doit vérifier d'une part que l'ordre est bien fondé sur $\mathcal{E}$ et d'autre par que chaque clause de $\mathcal{E}$ est une conséquence inductive de Ax. L'ensemble $\mathcal{E}$ contient 24 clauses. Cet exemple montre clairement la difficulté de la validation manuelle -déjà c'est un exemple difficile à lire-. Pour cette raison, et afin de traiter des preuves de taille industrielle il est fortement recommandé d'effectuer la validation des preuves par récurrence implicite de Spike automatiquement.

Conclusion

Dans ce chapitre nous avons présenté le démonstrateur des théorèmes Spike. Il implémente le système abstrait A basé sur la récurrence implicite. Montré correct, ce système abstrait A permet de construire des preuves par récurrence implicite correctes. Cependant, son implémentation peut ne pas être correcte. Le constat qui s'impose est qu'une certification automatique des preuves par récurrence implicite de l'assistant de preuve Spike est fortement recommandée.

Chapitre 3

Certification du raisonnement par récurrence sur les formules

Sommaire

En partant de l'**Exemple 2.21**, nous pouvons constater clairement la difficulté ainsi la nécessité de la certification de démonstrateur des théorèmes automatique Spike.

Dans ce chapitre, nous présenterons d'une façon générale la certification formelle en soulignant la différence entre un certificat et une trace de preuve. En partant de cette différence, nous proposons trois solutions générales de certification des preuves générées par Spike.

3.1 Certification formelle

La certification formelle d'un programme consiste à fournir explicitement une preuve qu'un programme possède un certain niveau de sûreté. Cette méthode est une précaution recommandée, voire nécessaire, pendant le développement des systèmes critiques où la moindre erreur peut causer d'énormes dégâts. Pour qu'elle soit considérée comme un certificat formel, la preuve doit être produite et vérifiée indépendamment et d'une façon formelle. La certification est une tâche difficile qui exige des fondements de déduction puissante. De nos jours, elle peut être effectuée par des assistants de preuve qui sont des programmes construits pour supporter la construction et la vérification des preuves pour des assertions mathématiques. Ces outils sont devenus une technologie mûre et indispensable au développement des logiciels. Par conséquent, la liste des démonstrateurs des théorèmes, des techniques et des approches s'est élargie. Par contre, la situation est transformée en un programme qui vérifie un autre programme. Ainsi, la question qui se pose est désormais : `comment faire confiance au premier programme ?` Avant de répondre à cette question, il est nécessaire de faire la différence entre les types de preuves qui peuvent être fournies par un assistant de preuve : preuve, trace de preuve et certification.

3.1.1 Preuve, trace de preuve et certification

La théorie de la preuve a réouvert un ancien débat [Asperti2012, Kerber2010] : `les preuves sont-elles importantes et nécessaires? Quelle est la différence entre un message et une preuve ?` Le débat des mathématiciens autour de la preuve trouve son origine dans la question : `Quel est le rôle joué par une preuve (un message ou un certificat) ?` Pour répondre à cette question, les mathématiciens se divisent en deux écoles :

- D'abord celle qui nie toutes possibilités de déduire la validité d'une preuve comme c'est le cas de G. Hardy. Pour cette école des mathématiciens, la preuve consiste, par exemple, à pointer du doigt un schéma pour expliquer le raisonnement à d'autres personnes [Asperti2012, Hardy1929].
- Par opposition à cet avis, une autre école considère que la preuve, en plus de son rôle de communication, ne peut être vraiment une preuve que lorsqu'elle est une évaluation claire, objective et vérifiable.

De ce fait, la différence entre un message (ou une trace de preuve) et une preuve (une démonstration mathématique) est une différence mathématique qui a également engendré une différence en théorie de la preuve. Donc, une preuve générée par un assistant de preuve ne peut être considérée comme un certificat que si elle établit les mêmes conditions qu'une preuve mathématique, qu'elle soit claire, objective et vérifiable.

A. Preuve

D'une façon très compacte, une preuve est une information suffisante pour décider la correction d'une assertion. En effet, elle est une séquence de raisonnements permettant de décider - à partir d'un ensemble d'assertions - si une proposition est vraie ou fausse. Ces instructions

peuvent utiliser un ensemble d'assertions qui peut contenir des axiomes (des assertions supposées initialement vraies) ou/et d'autres obtenues à partir de ces axiomes suivant de différents raisonnements mathématiques ou logiques. En théorie de la preuve, une preuve peut avoir deux formes possibles : un message ou un certificat. Un assistant de preuve peut générer une de ces deux formes ou les deux en même temps.

B. Message

Une preuve est considérée comme une trace de preuve ou un message si elle servira uniquement à la communication entre l'assistant de la preuve (un mathématicien ou un programme) et le lecteur de la preuve. Elle doit contenir une explication formée par deux informations nécessaires : *a*) une description des raisons de la validité d'une assertion donnée en fournissant l'intuition du démonstrateur, et *b*) une indication sur la ligne de raisonnement et les techniques utilisées pour conjecturer le résultat.

C. Certificat

Un certificat (ou une assurance) fournit une ligne de raisonnement bien précise. Un certificat de preuve doit être re-vérifiable d'une façon objective, automatique et indépendante de sa construction. L'argument de sa validation doit être suivi et vérifié sans même comprendre le sens de la preuve [Asperti2012]. Son rôle est de fournir un objet de preuve et non pas sa trace.

3.1.2 Méthodes et approches générales

Les approches de certification varient largement, de la révision manuelle de code à une vérification complètement formelle et automatique. Le niveau de confiance accordée à une approche augmente relativement avec sa formalité (son emploi des notions formelles). Le plus haut niveau de confiance est accordé souvent aux approches basées sur des méthodes formelles, des notions logiques et mathématiques, ainsi que des techniques de la théorie de la preuve. À ce stade, nous pouvons citer les méthodes suivantes :

1. `Certification par plusieurs démonstrateurs` : c'est l'une des approches adoptées par certaines autorités de certification à l'instar de l'Autorité de Certification de Logiciels CAST [16] qui est affiliée à la FAA [17]. Elle consiste à effectuer la vérification d'une seule preuve par plusieurs assistants. Les différents résultats générés par les assistants utilisés construisent un certificat s'ils sont tous validés. L'assurance de cette approche est l'absence de la contradiction entre les différents assistants. Cette méthode est très coûteuse et ne peut être généralisée.
2. `Révision manuelle:` la révision manuelle du code source ou des preuves consiste à effectuer cette tâche par l'homme. Cette certification est compliquée [Klein *et al.*2010], voire quasiment impossible pour des preuves de grande taille.
3. `Certification par un environnement certifié` : la communauté scientifique accorde le plus haut niveau de confiance à certains environnements de preuve certifiés. Ces outils possèdent une architecture spéciale qui sépare la construction de la preuve de la validation. Souvent, ces environnements sont interactifs.

16. *Certification Authorities Software Team*
17. *Federal Aviation Administration*

4. `Re-vérification des preuves générées par un démonstrateur automatique` : les assistants de preuve automatiques sont un moyen très puissant pour traiter des preuves de grande taille. Néanmoins, ce sont des programmes de grande taille dont la certification est fastidieuse. Pour qu'elle soit validée, il est possible de revérifier la preuve à l'aide d'autres assistants de preuve certifiés. L'approche consiste à coupler l'assistant de preuve automatique avec un environnement de preuve certifiée.

3.1.3 Environnement de preuve certifiée

un environnement de preuve certifiée est un outil d'interprétation de langage mathématique d'ordre supérieur vers un ensemble d'instructions de niveau moins élevé dont la correction peut être automatiquement vérifiée par la machine. L'efficacité d'un tel environnement est assurée par la séparation entre la construction de la preuve et sa validation d'un côté, et la certification de la composante de validation d'un autre côté. Ce type d'assistant de preuve est souvent un démonstrateur de théorème interactif [18].

A. Architecture d'un environnement de preuve certifiée

Pour limiter le risque des bugs dans la conception d'un tel environnement, son architecture doit contenir deux composantes principales : un moteur de preuve et un noyau de vérification. Le moteur de preuve regroupe tous les outils permettant la construction de la preuve. Afin de la certifier, la preuve sera par la suite validée par le noyau. Ces deux composantes doivent être clairement séparées de telle façon que la vérification d'une preuve s'effectue indépendamment et en ignorance complète de sa construction.

A.a. Moteur de preuve Il est le composant qui permet la construction de la preuve. Il contient des implémentations de différents raisonnements mathématiques et logiques. La majorité des assistants utilisent comme outil de construction de la preuve les tactiques qui sont introduites pour la première fois dans [Gordon *et al.*1979]. Une tactique est un moyen de simplification d'une assertion (ou but) par la recherche de sa preuve en fonction des preuves des sous assertions (ou sous-buts). Elle est le moyen d'interaction entre l'environnement de preuve et l'utilisateur. Certaines tactiques sont complètement automatiques, i.e. elles permettent la construction d'une preuve en une seule étape.

A.b. Noyau de vérification Il est le composant axial dans un environnement de preuve dans la mesure où il assure la vérification de la correction des théories formelles et y inclus les définitions et les preuves. En l'appliquant sur un objet de preuve, le noyau vérifie que ce dernier est construit à partir des étapes logiques élémentaires valides qui font des utilisations correctes des suppositions. En effectuant cette étape en indépendance de la construction de la preuve, la correction de la preuve dépend seulement de la cohérence de noyau. Ce dernier ayant souvent une petite taille, il est possible de le vérifier et d'établir sa preuve de cohérence. Étant une image de sa théorie, l'implémentation du noyau possède une petite taille qui peut être vérifiée et lui accorder un niveau élevé de confiance.

B. Exemples d'environnement de preuve certifiée

Parmi les environnements de preuve certifiés nous pouvons citer :

18. *Interactive Theorem Prover*

Coq [Coq development team] est un environnement de preuve basé sur la théorie des types. Il possède un *langage formel* unifié permettant l'expression des objets, des propriétés et des preuves en même temps et sont tous représentés avec la même structure générique syntaxique des λ-termes. Coq est muni également d'un vérificateur de preuve qui vérifie la validité des preuves. En effet, une preuve p est un λ-terme de type t, et sa vérification consiste à vérifier que le terme construit pour la preuve p est bien typé et de type t par rapport à l'ensemble de ses règles de typage. La certification de Coq est due à la certification de son noyau de typage.

ISABELLE [The ISABELLE development team] est un environnement de preuve interactif qui offre un environnement logique permettant de spécifier une théorie quelconque, sa syntaxe, ses axiomes et ses règles. Parmi les théories en question, nous pouvons citer la théorie de construction des types de Martin-Löf, des logiques de premier ordre, la théorie des ensembles de Zermelo-Fraenkel et la logique d'ordre supérieur. Ce démonstrateur de théorèmes est muni de différents moteurs de preuves génériques pouvant être instanciés par de différentes logiques comme un moteur de réécriture, un démonstrateur à base de tableaux etc.

HOL [The HOL development team] est un environnement de preuve interactif basé sur la logique d'ordre supérieur. Cet environnement permet en même temps de prouver des théorèmes et d'implémenter des outils de preuve. Ils permet aussi l'appel des moteurs de preuve externe tels que SMT [19] [Barrett *et al.*2009] et BDD [20] [Akers1978, Gordon2000]. Il possède un haut degré de programmabilité par le Méta-Langage ML.

3.1.4 Certification d'un assistant de preuve automatique

La certification des preuves critiques et de grande taille nécessite des démonstrateurs automatiques d'un niveau de performance élevé. Comme tout autre programme, la certification d'un assistant de preuve automatique ATP [21] permet d'augmenter sa sûreté de fonctionnement, ainsi que de garantir la correction de ses preuves. Par contre, cette tâche devient très coûteuse et fastidieuse, vu la grande taille des assistants automatiques qui sont souvent basés sur des calculs compliqués et des structures de données avancées et qui ont des implémentations optimisées. Donc, une vérification formelle du code source de tels programmes est fastidieuse voire impossible.

Certaines autorités prétendent qu'il existe des ATPs certifiés. Par contre, cette classification reste relative et sociale car elle ne possède pas de forts arguments. Par exemple, la classification de la CASC [22] [23] de 1999 avait sélectionné un certain nombre d'ATP comme assistants certifiés, pour que par la suite deux d'entre eux s'avèrent non cohérents [Sutcliffe2000]. Donc, il est souvent critique de considérer qu'un démonstrateur automatique est correct, surtout s'il est en cours de développement. On peut dire que le niveau de sûreté est accordé à une version de cet ATP, et non pas à l'ATP lui-même. Sa nouvelle version peut remettre en question la validation de l'ATP.

Cette discussion nous amène à une nouvelle approche qui est une alternative de la vérification formelle des ATPs. Elle consiste à valider indépendamment les preuves générées en construisant

19. *Satisfiability Modulo Theories*
20. *Binary Decision Diagram*
21. *Automatic Theorem Prover*
22. *CADE ATP System Competition*
23. La CASC est organisée par la CADE : *International Conference on Automated Deduction*

leurs certificats qui sont indépendamment re-vérifiables à l'aide d'un outil externe. Cette approche ne peut jamais prétendre la certification complète d'un ATP, mais se limite à la certification de certaines propriétés. La création des environnements de preuve certifiés a ouvert la porte à une approche de certification des ATPs qui devient de plus en plus utilisable et qui consiste à utiliser un ITP pour certifier un ATP. De cette approche dérivent différentes techniques prometteuses.

Parmi ces techniques, nous citons la certification de l'implémentation d'un ATP. En effet, les développeurs des ATPs fondent généralement leurs assistants sur des calculs corrects et des théories solides qui sont déjà validées par la communauté scientifique. Cependant, la grande taille des ATPs et la complexité de leurs architectures peut empêcher la validation de la conformité de ces ATPs par rapport à leurs bases théoriques. Et même si nous imaginons une certification d'une implémentation d'un ATP, nous allons nous contenter d'une certification de sa conception, d'une partie de son code ou de l'une de ses versions (notamment si l'ATP est en cours de développement).

En revanche, nous pouvons penser à la certification des preuves d'un ATP au lieu de la certification de son implémentation. La certification des preuves consiste principalement à fournir un certificat tel que nous l'avons défini auparavant, et qui affirme que cette preuve est correcte. En termes de théorie de la preuve, si la preuve est construite dans un système formel cohérent, on peut considérer que celle-ci est correcte et que son objet construit est un certificat. De ce fait, nous pouvons utiliser un ITP certifié dont le système formel est considéré comme correct pour valider les preuves d'un ATP. La validation consiste à reconstruire ces preuves dans un ITP certifié, i.e si elles sont acceptées par ce dernier, alors elles sont correctes. Cette approche devient de plus en plus utilisée et elle est la base de plusieurs travaux. Nous citons deux façons possibles de valider les preuves d'un ATP en utilisant un ITP certifié : i) traduire ces preuves en des scripts vérifiables par un ITP et dans ce cas le certificat est la traduction de la preuve, ou ii) intégrer son raisonnement directement dans le moteur de preuve de l'ITP afin d'obtenir des certificats sous la forme d'objets de preuve directement dans l'ITP.

Ces techniques présentent un certain niveau de difficulté. En effet, les ATPs ne fournissent que des traces de preuve d'où la nécessité de créer un moyen qui permet de transformer les preuves de l'ATP en des scripts ITP. Ces scripts peuvent être de grande taille dans la mesure où les ITPs sont souvent basés sur des constructions interactives des preuves. Ainsi, il est souvent nécessaire d'intégrer des nouvelles définitions, des notions supplémentaires et des raisonnements automatiques dans les ITPs et les valider par la suite.

Dans ce qui suit, nous discuterons des techniques de validation des preuves dans le cas d'un ATP qui génère des preuves basées sur la récurrence sur les formules.

3.2 Certification d'un ATP basé sur la récurrence sur les formules

Nous considérons qu'il existe un ATP qui implémente le raisonnement par récurrence sur les formules. Nous discuterons dans cette section des techniques de certification de preuves de l'ATP en utilisant un ITP certifié, comme la fusion de deux environnements ou l'intégration du premier dans le moteur de preuve du deuxième.

3.2.1 Fusion d'un ATP basé sur la récurrence sur les formules et un ITP certifié

A. Approche générale

Algorithm 3.1 Fusion entre un ATP et un ITP

```
1: procedure FUSIONNER_ATP_ITP
2:     (theoreme_courant, environnement) := recuperer_environnement_preuve()
3:     specification := specifier_ATP(theoreme_courant, environnement)
4:     si specification extraite alors
5:         preuve := appeler_ATP(specification)
6:         si preuve existe alors
7:             script_preuve := traduire_preuve_ATP(preuve)
8:             si traduction preuve réussie alors
9:                 noyau_ITP(script_preuve)
10:                si validation réussie alors
11:                    preuve est correcte.
12:                sinon
13:                    échec de la validation de la preuve.
14:                fin si
15:            sinon
16:                échec de la traduction de la preuve.
17:            fin si
18:        sinon
19:            échec de la génération de la preuve.
20:        fin si
21:    sinon
22:        échec de l'extraction de la spécification.
23:    fin si
24: fin procedure
```

La fusion entre un ITP et un ATP consiste à considérer que l'ATP est le générateur des preuves et que l'ITP est leur vérificateur. Cette technique permet la validation des preuves par récurrence basée sur les formules, et ce par traduction des preuves de l'ATP en script ITP.
Cette approche nécessite la création de deux composantes automatiques dont la première est dans l'ITP qui appelle l'ATP pour construire la preuve, et la deuxième dans l'ATP qui traduit la preuve dans un script ITP pour valider la preuve. La procédure générale de cette technique de certification est représentée formellement par l'**Algorithme 3.1**. En effet, en partant de l'environnement de preuve de l'ITP, la procédure commence par extraire la spécification de la proposition du théorème à prouver dans le langage de spécification de l'ATP. Cette spécification contient également toutes les informations nécessaires pour commencer une preuve, comme par exemple le contexte (les structures de données, les fonctions, les axiomes et les assomptions). L'ATP génère en même temps la preuve et sa traduction en script ITP. Ce script peut également contenir toutes les propriétés et les propositions nécessaires à la validation d'une preuve de l'ATP et leurs preuves de correction. Si ce script qui est supposé équivalent à la preuve établie par l'ATP est validé par le noyau de l'ITP, alors il est considéré comme un certificat de la preuve de l'ATP.

B. Exemples

La fusion entre un ATP et un ITP est une approche qui a été utilisée dans plusieurs occasions pour certifier des preuves basées sur de différents raisonnements. Nous citons dans la suite certains travaux s'inscrivant dans ce cadre :

Combinaison de théorie de la preuve automatique et interactive dans la théorie des types [Kanso et Setzer2010, Kanso2012] : c'est une approche d'intégration d'ATP dans l'ITP Agda [Bove *et al.*2009] basée sur la théorie des types de Martin-LÖf. Elle implémente des cas particuliers de solveurs SAT/SMT et des outils de validation de modèles de type CTL [24] [Clarke *et al.*2001]. Cette approche a été utilisée pour vérifier la correction des systèmes de verrouillage des chemins de fer.

Intégration de Gandalf [Tammet1997]dans HOL [Hurd1999] : Gandalf est un démonstrateur de théorèmes équationnels portant sur des clauses de grande taille dans un temps d'exécution optimisé. *Gandalf_tac* est la tactique dans HOL qui permet d'appeler Gandalf, de traduire sa preuve et de vérifier sa cohérence à l'aide de HOL.

Intégration de Jprover dans des ITPs [Schmitt *et al.*2001] : Jprover est un démonstrateur de théorèmes en logique de premier ordre à la fois intuitionniste et classique, basé sur la méthode de connexion. Il est couplé à des mécanismes d'intégration dans le démonstrateur Nuprl [Kreitz2002] qui est un système de développement de programmes et de preuves, et MetaPRL [Hickey et Nogin2000] qui est un environnement de preuve.

Construction automatique de preuve dans la théorie des types en utilisant la résolution [Bezem *et al.*2002] : Bliksem [Bliksem development team] est un démonstrateur de théorèmes basé sur la résolution. L'approche consiste à identifier des étapes non triviales dans Coq et qui correspondent à des tautologies d'ordre premier. Ces tautologies seront exportées vers Bliksem pour chercher la preuve de résolution qui est convertie dans la théorie des types, puis importée dans Coq.

Elan pour des raisonnements équationnels dans Coq [Alvarado et Nguyen2000] : Elan [Borovanský *et al.*1998] est un système qui offre un environnement pour la spécification et le prototypage des systèmes de déduction avec un langage basé sur des règles de réécriture. Il est fusionné avec Coq à l'aide d'une tactique qui permet la communication entre les deux assistants via des sockets TCP/IP de telle sorte qu'Elan joue le rôle du serveur, et que Coq joue le rôle d'un client. Le processus de Coq génère et lance un moteur de réécriture en lui fournissant les termes à réécrire accompagnés des stratégies de réduction. Enfin, on récupère la trace de calcul d'Elan. Cette méthode permet de certifier des traces d'Elan.

Recherche d'une preuve avec la masse [25] dans ISABELLE : Sledgehammer [Böhme et Nipkow2010] peut être appelée dans n'importe quel point de la preuve pour fonctionner en arrière plan. L'idée générale consiste à transformer le but initial en une clause pour la fournir à des ATPs avec une collection de centaines de clauses et de lemmes qui représentent des faits extraits de la théorie d'ISABELLE. La preuve automatique peut être cherchée par une liste d'ATPs en parallèle avec sa construction interactive, la première arrivée étant celle considérée. La reconstruction de la preuve s'effectue avec la méthode générale de Metis [Hurd2003].

24. *Computation Tree Logic*
25. *Sledgehammer*

C. Motivation

Le but majeur de la fusion entre deux assistants est la certification des preuves de l'ATP. L'utilisation d'un ITP certifié accomplit parfaitement ce but en fournissant un certificat de correction de la preuve construite par l'ATP. Ainsi, quand ils se sont développés la première fois, les ITPs n'ont pas eu de moyens automatiques pour gérer des preuves de grande taille. L'intégration des composantes automatiques permet de bénéficier des procédures automatiques des ATPs pour résoudre des problèmes en un nombre réduit d'étapes, voire en une seule étape.

D. Limites

D'après la procédure générale décrite dans l'**Algorithme 3.1**, il existe quatre cas d'échec possibles. La ligne 22 peut s'exécuter à cause d'une différence entre les deux langages de spécification des deux assistants. Par exemple, dans le cas de l'intégration des raisonnements équationnels par ELAN dans Coq, l'interface ELAN/Coq fonctionne uniquement pour des systèmes de réécriture des termes de premier ordre multi-sortés [Alvarado et Nguyen2000]. De ce fait, l'extraction ne se fait que pour une sous-classe de spécifications de Coq. La ligne 19 peut être atteinte suite à une erreur de génération de la preuve due à l'incapacité de l'ATP pour trouver une preuve. Par instance, la tactique **Gandalf_tac** échoue certainement si **Gandalf** ne retourne pas une preuve [Hurd1999]. La ligne 16 peut être invoquée par une erreur de traduction de la preuve. La reconstruction de 10% des preuves en ISABELLE en utilisant Metis échoue pour des preuves correctes dans les ATPs utilisés dans [Böhme et Nipkow2010] à cause des longs temps d'exécution. La ligne de code 13 indique que la preuve construite n'est pas correcte. Le rejet de la preuve par l'ITP ne signifie pas la non validité de la conjecture, mais plutôt la non correction de la preuve générée par l'ATP.

3.2.2 Construction automatique d'une preuve par récurrence sur les formules à l'ATP directement dans un ITP certifié

A. Approche générale

La certification des preuves d'un ATP par construction directe dans un ITP se base sur l'intégration des raisonnements automatiques à l'ATP dans l'ITP. Cette intégration peut engendrer la définition des raisonnements de l'ATP et la création d'outils qui permettent de rejouer ses stratégies et ses règles d'inférence dans l'ITP. Pour cette méthode, une seule composante automatique dans l'ITP suffit. Cette composante effectue une ou plusieurs étapes préparatoires et des prétraitements invisibles pour trouver la preuve de la même façon que l'ATP. Ensuite, elle les valide et les réutilise pour résoudre un théorème dans l'ITP. Par exemple, dans notre cas où l'ATP implémente le raisonnement par récurrence basée sur les formules, le pré-traitement consiste à construire une pré-preuve.

Une preuve par récurrence basée sur les formules peut employer des hypothèses de récurrence qui correspondent à des formules non encore prouvées. Pour cela, nous proposons de séparer l'étape de la construction de la pré-preuve de l'étape de la validation. Dans un premier temps, nous créerons une pré-preuve à l'aide d'outils à l'ATP. Ensuite, nous validerons les hypothèses de récurrence. La validation consiste à certifier l'application de la récurrence noethérienne sur l'ensemble des formules de la pré-preuve.

Définition 3.1 (Pré-preuve par récurrence basée sur les formules) ***Une pré-preuve basée sur les formules*** *est toute preuve construite par un ATP dans un ITP et qui contient des hypothèses de récurrence non validées.*

La procédure générale de la méthode est décrite formellement par l'**Algorithme 3.2**.

Algorithm 3.2 Construction automatique des preuves par récurrence sur les formules à l'ATP dans un ITP

```
1: procedure CONSTRUCT_PROOF_ATP_ITP
2:     si strategie existe alors
3:         strategie := recuperer_startegie(environnement_courant)
4:     sinon
5:         strategie := strategie_predefinie()
6:     fin si
7:     but := recuperer_but(but_courant)
8:     contexte := recuperer_contexte(environnement_courant)
9:     si récupération contexte et but réussis alors
10:        ensemble_formules := construire_pre_preuve(strategie, but, contexte)
11:        si construction de la prépreuve réussit alors
12:            valider_preuve(ensemble_formules)
13:            si validation de la preuve réussit alors
14:                preuve est correcte.
15:            sinon
16:                échec lors de la validation de la preuve
17:            fin si
18:        sinon
19:            échec lors de la construction de la pré-preuve
20:        fin si
21:    sinon
22:        échec lors de récupération du contexte
23:    fin si
24: fin procedure
```

B. Exemples

Certains des travaux basés sur la fusion des assistants de preuve cités dans le paragraphe précédent sont passés de la fusion à l'intégration des ATPs dans les ITPs comme dans [Bezem *et al.*2002, Hurd2003]. La technique a eu un succès chez les environnements interactifs parce qu'elle a permis d'améliorer considérablement leur automatisation [Meng *et al.*2006]. Les listes de contributions des ITPs certifiés contiennent souvent plusieurs exemples, parmi lesquels nous citons ici :

Des tactiques de premier ordre dans des démonstrateurs des théorèmes en logique d'ordre supérieur : Hurd propose dans [Hurd2003] une architecture générale pour l'intégration des procédures basées sur la logique de premier ordre en tant que tactiques pour prouver des buts dans des ITPs basés sur la logique d'ordre supérieur.

Raisonnement de premier ordre dans le calcul de constructions inductives : Corbineau dans [Corbineau2003] propose une procédure de recherche de preuve implémentée en tant que tactique dans Coq. Elle se base sur un calcul de séquents sans contraction qui est une extension du calcul LJT [Dyckhoff1992] de Dyckhoff et qui vérifie les propriétés de contraction et d'élimination des coupures.

Une construction automatique de preuve dans la théorie des types en utilisant la résolution : Dans [Bezem *et al.*2002], on propose une technique d'intégration de la résolution logique basée sur des égalités dans la théorie des types. Cette méthode a permis de profiter de la puissance de la résolution dans un environnement de preuve interactif. Elle se base sur la traduction des preuves de résolution dans des $\lambda-$termes afin qu'ils soient vérifiés.

C. Motivation

Dans le même cadre de la méthode précédente, la construction automatique de la preuve par récurrence basée sur les formules dans un ITP a pour but principal la certification des preuves de l'ATP. Par contre, elle réduit le nombre d'échecs en évitant l'étape de la traduction et évite l'utilisation d'outil externe tel que l'ATP. Ce dernier sera présent en tant que stratégie générale de recherche de la preuve et non plus en tant que générateur de la preuve. Les procédures de l'ATP permettent uniquement de guider la construction de la preuve. Cependant, la vérification est effectuée par l'ITP qui garantit la correction de la preuve construite.

Dans notre cas, cette approche permet tout d'abord d'effectuer la récurrence paresseuse et mutuelle dans l'ITP. En plus, elle permet d'utiliser des raisonnements et des calculs existant dans l'ITP. Et enfin, elle offre une traduction raffinée des preuves par récurrence basée sur les formules.

D. Limites

Le langage de spécification d'un ITP est très expressif et permet des spécifications que l'ATP peut ne pas traiter. Ce cas correspond à la ligne de code 22 de l'**Algorithme 3.2**. L'échec de la construction de la prépreuve de la ligne de code 19 de l'**Algorithme 3.2** est dû aux procédures de l'ATP qui essaient de chercher une preuve, mais cette recherche a divergé ou échoué. Cet échec ne signifie pas nécessairement la non validité de la conjecture à prouver. Comme toute preuve construite dans l'ITP, elle se termine par la vérification de correction de la preuve. Si cette étape qui correspond à la ligne de code 16 échoue, alors la preuve construite à l'ATP est fausse et son calcul est non cohérent. Dans notre cas (l'intégration du raisonnement à la Spike dans Coq), l'échec pourrait être dû à une utilisation non justifiée d'une hypothèse de récurrence.

3.2.3 Validation interactive d'un théorème par récurrence sur les formules à l'ATP dans un ITP certifié

A. Approche générale

Un théorème énoncé dans un ITP certifié peut être prouvé interactivement par des tactiques qui implémentent le raisonnement par récurrence basée sur les formules. La particularité d'une telle preuve est l'utilisation des formules de la preuve qui ne sont pas encore validées. De ce fait, sa validation nécessite une étape supplémentaire qui consiste à vérifier la correction de l'application du raisonnement par récurrence sur l'ensemble des formules de la preuve. L'**Algorithme**

3.3, respectivement l'**Algorithme 3.4**, explique l'application d'une tactique qui implémente la récurrence sur les formules, respectivement la validation d'une preuve qui utilise une telle tactique.

Algorithm 3.3 Application d'une tactique basée sur récurrence sur les formules

```
1: procedure APPLICATION_TACTIQUE_RECURRENCE_FORMULES
2:     but := recuperer_but(but_courant)
3:     contexte := recuperer_contexte(environnement_courant)
4:     premisses := recuperer_premises(environnement_courant)
5:     si récupération contexte, prémisses et but réussit alors
6:         tactique := appliquer_regle_inference(but, contexte, premisses)
7:         application tactique
8:         si l'application de la règle d'inférence réussit alors
9:             nouveaux buts introduits
10:        sinon
11:            échec de l'application de la tactique
12:        fin si
13:    sinon
14:        échec de récupération du contexte, prémisses et but réussit
15:    fin si
16: fin procedure
```

Algorithm 3.4 Certification d'un théorème dans un ITP par récurrence sur les formules

```
1: procedure VALIDATION_THEOREM_RECURRENCE_FORMULES
2:     contexte := recuperer_contexte(environnement_courant)
3:     ensemble_formules := recuperer_ensemble_formules(environnement_courant)
4:     si récupération contexte et ensemble des formules de la preuve réussit alors
5:         valider_preuve(ensemble_formules)
6:         si validation de la preuve réussit alors
7:             théorème est valide
8:         sinon
9:             échec lors de la validation de la preuve
10:        fin sip
11:    sinon
12:        échec de récupération du contexte et de l'ensemble des formules de la preuve
13:    fin si
14: fin procedure
```

B. Motivation

L'avantage important de cette approche est la possibilité d'utiliser d'autres tactiques mis à part celles qui implémentent la récurrence sur les formules. En plus, elle permet à l'utilisateur de guider la construction de la preuve et de rester dans le même style interactif des ITPs. L'interaction avec ces outils permet d'éviter l'inconvénient majeur des approches automatiques, à savoir leur divergence.

C. Limites

L'inconvénient majeur de l'implémentation de cette approche est le décalage de l'étape de la validation jusqu'à la fin de la preuve. En effet, la validation de la preuve par récurrence consiste principalement à vérifier que toutes les hypothèses de récurrence employées au cours de la preuve respectent les contraintes d'ordre de récurrence. Il suffit de trouver une seule hypothèse de récurrence qui viole l'ordre pour rejeter une preuve. Dans le cas des preuves de grande taille, cette vérification qui s'effectue à la fin de la preuve peut être décevante pour l'utilisateur.

Conclusion

Dans cette chapitre, nous avons présenté trois façons possibles pour certifier les preuves par récurrence générées par un ATP ou construites à l'ATP à l'aide d'un ITP. Ces trois approches diffèrent au niveau de la construction de la preuve : grâce à un outil externe, automatiquement ou interactivement. Néanmoins, elles nécessitent une étape supplémentaire de validation par rapport aux autres preuves construites dans un ITP. Dans ce qui suit, nous discuterons de cette validation qui prend en considération si la preuve est construite d'une façon réductible ou non.

Spike est un ATP qui permet de générer des preuves par récurrence implicite. Coq est un environnement de preuve certifiée et basé sur le calcul des constructions. Nous adoptons les trois approches proposées dans ce chapitre pour certifier les preuves de Spike en utilisant Coq :

- La fusion entre Spike et Coq ;
- L'intégration du raisonnement automatique à la Spike dans Coq ;
- Construction interactive des preuves par récurrence implicite à la Spike dans Coq.

Dans ce qui suit, nous présenterons Coq et ces trois approches.

Chapitre 4

Coq : un environnement de preuve certifiée

Sommaire

Coq [Coq development team, Bertot et Castéran2004] est un environnement qui permet de développer des programmes ayant un haut niveau de confiance. Cet assistant de preuve permet de spécifier formellement un programme et de prouver qu'il est cohérent avec sa spécification. Ayant un petit noyau basé sur la théorie des types et validé par la communauté scientifique, Coq assure la construction des preuves certifiées. Son langage GALLINA est considéré comme une extension du calcul des constructions inductives (CIC [26]) [Coquand et Huet1988, Paulin-Mohring1993]. Et ses preuves sont représentées sous la forme des séquents. La liste des exemples de l'utilisation de Coq est innombrable. Elle varie entre la logique, les automates, la syntaxe et la sémantique des langues naturelles, l'algorithmique, etc, et porte sur des différents domaines comme les télécommunications, la sécurité des transports, l'énergie, la cryptologie, etc [Coq development team].

Coq se base sur une théorie riche et possède un langage très expressif. Dans ce chapitre, nous nous intéressons uniquement aux notions qui permettent de définir une spécification à la Spike, de construire une preuve par récurrence basée sur les clauses égalitaires et de certifier cette preuve.

4.1 Spécification GALLINA

Le langage GALLINA permet d'une part de spécifier formellement un programme, y compris ses structures de données et ses fonctions. D'autre part, il permet d'exprimer les propriétés de ce programme et de construire ses preuves de correction. Chaque spécification exprimée dans GALLINA ne contient que des expressions bien formées suivant des règles de constructions [Bertot et Castéran2004]. Coq possède une structure générique syntaxique unique qui permet de définir tout terme GALLINA [27]. Coq possède des structures générales qui permettent de gérer les différentes définitions et déclarations. Nous présenterons dans la suite ces structures ainsi les définitions, les déclarations et les structures syntaxiques générales des termes que nous utiliserons dans la suite.

4.1.1 Section, environnement et contexte

Tout objet dans Coq possède un **identificateur** [28] unique. La **déclaration** est la création d'un nouveau identificateur auquel on associe un type. La **définition** est la création d'un nouveau identificateur auquel on associe une valeur à partir de laquelle est déterminé son type. D'une façon générale, un objet dans Coq peut être global ou local. Une **section** est une structure de Coq qui regroupe un bloc de développement permettant de gérer les déclarations et les définitions locales. À n'importe quelle étape du développement, il existe un environnement courant et un contexte courant. L'**environnement** contient des déclarations et des définitions globales. Le **contexte** contient des déclarations et des définitions locales.

Notation 4.1 *Ces notations sont les mêmes que celles dans [Bertot et Castéran2004]. Dans ce qui suit, nous notons l'environnement par E et le contexte par Γ. Le contenu de chacun est noté comme suit : $[t_1 : type_1; \ldots; t_n := val_n : type_n; \ldots]$, t_1 et t_n sont respectivement une déclaration et une définition. Si le contenu est vide, on le note par $[]$. Nous notons ainsi l'ajout d'une déclaration $t' : type'$ (respectivement d'une définition $t' := val' : type'$) à un contenu d'un environnement ou*

26. *Calculus of Inductive Constructions*
27. La structure générale d'un terme Coq est fournie dans le premier chapitre de son manuel [Coq development team].
28. Techniquement, l'identificateur est un symbole ou une chaîne de symboles séparés par des points -.- qui indiquent ainsi l'emplacement physique de cet objet.

d'un contexte comme suit : $[t_1 : type_1; \ldots; t_n := val_n : type_n; \ldots] :: (t' : type')$ *(respectivement* $[t_1 : type_1; \ldots; t_n := val_n : type_n; \ldots] :: (t' := val' : type')$*). Nous utilisons* $\cup$ *pour exprimer l'union de deux contenus.*

Exemple 4.2

`Section` *exemplesection.*

(* $E := []$ *et* $\Gamma := []$. *)

`Variable` $v : nat$.

(* $E := []$ *et* $\Gamma := [v : nat]$. *)

`Definition` $add : nat \to nat \to nat$.

(* $E := [add : nat \to nat \to nat]$ *et* $\Gamma := [v : nat]$. *)

`Definition` *add1 (u1 u2 : nat) :=* $add\ u1\ u2$.

(* $E := [add : nat \to nat \to nat; add := fun\ u1\ u2 : nat \Rightarrow add\ u1\ u2 : nat \to nat \to nat]$ *et* $\Gamma := [v : nat]$. *)

`End` *exemplesection.*

(* $E := [add : nat \to nat \to nat; add := fun\ u1\ u2 : nat \Rightarrow add\ u1\ u2 : nat \to nat \to nat]$ *et* $\Gamma := []$.

4.1.2 Terme GALLINA

Coq se base sur le calcul des constructions inductives, un λ-calcul enrichi avec des types inductifs, polymorphes, de niveau supérieur, dépendants et primitifs. Tout objet dans Coq - fonction, structure de données ou preuve- est un terme qui habite (ou appartient à) un type. Contrairement à d'autres théories de types, le terme et le type dans Coq ont la même structure syntaxique. L'unicité de la structure permet d'une part de définir les termes et les types d'une façon récursive et mutuelle, et d'autre part d'effectuer les mêmes constructions sur les deux. Dans la suite, nous présenterons uniquement les structures générales de différents termes GALLINA que nous utiliserons dans nos travaux.

A. Type, sorte et proposition

Définition 4.3 (Typage et type habité) *Soit* t *un terme de* ***type*** T *dans un environnement* E *et un contexte* Γ*. On note la règle de typage correspondante par* $E, \Gamma \vdash t : T$*. Et on dit que le type* T *est* ***habité*** *s'il existe un terme* t *tel que* $E, \Gamma \vdash t : T$ *est valide.*

Notons que $E, \Gamma \vdash t : T$ est valide si elle dérive de l'une des règles de typage de Coq citées dans [Coq development team] ou en détail dans [Bertot et Castéran2004]. Ces règles de typage sont en même temps les règles de construction des termes. Ce qui permet de construire toujours des termes bien typés. Toute expression dans Coq possède un type déterminé suivant ces règles et ne peut être acceptée que si ce type est bien typé.

Exemple 4.4 *Soit la règle de typage qui correspond aux variables :*

$$\textbf{\textit{Var}}\ \frac{(x, A) \in E \cup \Gamma}{E, \Gamma \vdash x : A}\ \textit{[Bertot et Castéran2004].}$$

Cette règle permet de vérifier le typage pour une variable.

Dans le Calcul des Constructions (ainsi dans Coq), on appelle une **sorte** le type d'un type qui est considéré comme un terme. Soient t' un type de sorte s et t un terme de type t'. On dit donc que t est de sorte s. Soient les sortes `Set` et `Prop` deux sortes prédéfinies et le plus abstraites dans Coq. D'une façon générale, les sortes se définissent suivant une hiérarchie bien-formée, infinie et définie avec des univers de la façon suivante :

Définition 4.5 (Univers) *[Bertot et Castéran2004] L'**univers** est la famille des termes de type :* `Type(i)` *qui satisfait la relation suivante :*

`Set` *:* `Type`*,*

`Prop` *:* `Type` *et*

`Type(i)` *:* `Type(i+1)`*,* $i \in \mathbb{N}$.

Le type le plus abstrait est l'univers de premier niveau `Type` et les types de base sont `Prop` et `Set` ayant comme type `Type`. Notons que le type `Prop` est l'univers de toutes les propositions logiques et les preuves, et le type `Set` est l'univers de tous les spécifications et les programmes.

Définition 4.6 (Proposition et preuve) *Une **proposition** est tout terme p de type Prop. Tout terme t habitant ce type p est un **terme de preuve** ou tout simplement une **preuve**, de la proposition exprimée par p.*

En se basant sur l'isomorphisme de Curry-Howard [Howard1980], d'une part une proposition p est prouvée si on trouve un terme t bien typé qui l'habite et d'autre part les règles d'inférence sont en même temps les règles de construction de ce terme t. Ces règles sont les règles de typage pour cette raison elles résultent un terme t bien typé de type p. De ce fait, la construction d'un terme avec ces règles assure qu'il est une preuve -assure qu'il est bien typé et qu'il est de même type que la proposition-.

4.1.3 Type inductif

Parmi les points forts du Coq nous citons les types inductifs. Ces types sont utilisés pour décrire des structures des données et des prédicats. Dans nos travaux, nous les utilisons pour spécifier des structures des données récursivement définies telles que les naturelles, les listes, etc. Les types inductifs sont des sortes, i.e. des types appartenant à l'univers `Set`. La commande vernaculaire pour la définition d'un type inductif est *Inductive*.

La forme syntaxique des types inductifs est la suivante :

$ident_{IT} : Set \quad :=$

$\quad |ident_0 : type_0$

$\quad | \ldots$

$\quad |ident_n : type_n$

$ident_{IT}$ est l'identificateur d'un nouveau type inductif et $ident_0 \ldots ident_n$ sont les identificateurs de ses constructeurs auxquels on associe les types respectifs $type_0 \ldots type_n$.

A. Variables

Les variables dans Coq peuvent être des déclarations ou des définitions globales ou locales, ou des variables liées à une expressions. Dans nos travaux, nous nous intéressons uniquement à ce dernier cas. Des telles varaibles sont définies en utilisant la structure syntaxique les *binders*.

D'une façon générale, une variable se définit par son identificateur, son type et éventuellement une valeur. Un identificateur ou un type peut être remplacé par le symbole "_" (appelé *joker*). Les jokers correspondent à des méta-variables dont Coq associe un identificateur et un type -le type est déduit de type de l'expression où il a apparu ce joker-. La partie de la structure syntaxique générale *term* qui correspond à la définition des binders est la suivante -notons ici que *ident* représente un identificateur- :

name ::= *ident* | _

binder ::=

| *name*

| (*name* ... *name* : *term*)

| (*name* [: *term*] := *term*)

binders ::= *binder* ... *binder*

Exemple 4.7 *Dans le terme suivant* $(u_1\ u_2\ \_ : nat) \Rightarrow expr$*, le binder* $(u_1, u_2 : nat)$ *permet d'exprimer que les variables* u_1 *et* u_2 *sont des varaibles liées à l'expression expr. Ainsi, expr possède une méta-variable de type nat.*

B. Abstraction et produit

Une $\lambda-$abstraction (ou une abstraction) est un terme Coq ayant la forme suivante :

`fun` *binders* `=>` *term*

Ce terme permet de définir l'abstraction des variables définies par *binders* sur le terme *term*. Cette abstraction définit une fonction paramétrée par des variables.

Exemple 4.8 *fun x y ⇒ add x (S y) = S (add x y)*

Le produit est un terme de la forme suivante :

`forall` *binders*`,` *term*

Ce terme permet d'exprimer le produit des variables sur un terme *term*.

Exemple 4.9 *∀ x y, add (S x) y = S(add x y)*

En effet, ce type permet de définir des variables universelles d'une expression, sinon, il représente une proposition d'implication logique telle que $A \rightarrow B$ peut être représenté par le produit suivant $\forall(\_ : A), B.$, ou un type d'une fonction que nous traiterons dans la définition des fonctions.

Techniquement, chacun de ces deux structures -l'abstraction et le produit- peuvent se décomposer en une paire constituée par une liste des varaibles et un terme.

Exemple 4.10 *La décomposition de :*

- *∀ x y, add (S x) y = S(add x y) est ([x : nat ; y : nat], add (S x) y = S(add x y)), et*
- *∀ x, odd (add(x,x)) = false → true = true est ([x : nat ; H : odd (add(x,x)) = false], true = true).*

C. Filtrage (Pattern matching)

Le pattern matching est une structure de Coq qui permet d'effectuer un filtrage[29] sur un terme de type inductif. Sa forme générale est la suivante :

match_item : := *term* [`as` *name*] [`in` *term*]

pattern : :=

| *qualid pattern* ... *pattern*

| *pattern* `as` *ident*

| *pattern* `%` *ident*

mult_pattern : := *pattern* `,` ... `,` *pattern*

equation : := *mult_pattern* `|` ... `|` *mult_pattern* `=>` *term*

`match` *match_item* `,` ... `,` *match_item* [*return_type*] `with`

[`|` *equation* `|` ... `|` *equation*] `end`

Le *qualid* au début de *pattern* est un constructeur de type inductif de *match_item*. Le filtrage s'effectue par cas, i.e. chaque branche correspond à un cas.

Nous distinguons un cas particulier de pattern matching. C'est le filtrage effectué sur une valeur booléenne qui correspond au terme :

`if` *term* `then` *term* `else` *term*

4.1.4 Fonction récursive

Une fonction récursive est une expression de la forme suivantes :

`fix` *fix_bodies*

fix_bodies : :=

| *fix_body*

| *fix_body* `with` *fix_body* `with` ... `with` *fix_body* `for` *ident*

fix_body : := $ident_0$ *binders* [`{ struct` $ident_1$ `}`] [`:` *term*] `:=` *term*

Cette structure permet de définir des fonctions récursives et mutuelles. L'identificateur $ident_0$ est le nom de la fonction récursive. Comme tout définition, le type d'une fonction peut être défini implicitement par Coq ou explicitement par l'utilisateur. L'identificateur $ident_1$ est l'argument décroissant de la fonction qui peut être déterminé par Coq ou défini explicitement.

4.2 Construction des preuves certifiées

Une preuve dans Coq commence par énoncer une proposition qui est appelé un but initial. Pour simplifier sa preuve, on la diviser par des tactiques en sous-buts dont les solutions construiront la solution du but initial.

29. Filtrage est la traduction de pattern matching selon la version française de [Bertot et Castéran2004].

4.2.1 But et séquent

A. Séquent

Les buts dans Coq sont présentés par des séquents. Introduit pour la première fois par [Gentzen1935, Szabo1969], le calcul des séquents est considéré comme le système le plus élégant et souple pour écrire des preuves [Buss1998b]. Il est adopté ensuite par des différents assistants de preuve tels que Coq [Coq development team], ISABELLE [The ISABELLE development team], Nuprl [Kreitz2002], PVS [The PVS development team].

Définition 4.11 (Séquent) *Un séquent est une expression de la forme :*

$$\Gamma \vdash \Sigma$$

avec Γ *est un ensemble de formules appelés antécédents et* Σ *est une formule appelée la conclusion.*

Dans une preuve construite dans le calcul de séquent chaque étape de la preuve est un séquent.

B. But

L'énoncé d'une proposition doit être un terme bien typé de type *Prop*. Tout énoncé d'une proposition qu'on souhaite prouver est introduit par l'une des commandes vernaculaires suivantes : *Theorem*, *Lemma*, *Goal* ou *Definition*. Si l'énoncé de preuve n'est pas bien typé alors la preuve ne peut pas être commencée. Le but se représente sous la forme d'un séquent de la façon suivante :

Définition 4.12 (But sous forme d'un séquent dans Coq) *Un but dans Coq est un* λ*-terme. Quand il est ouvert (non encore prouvé), on le trouve sous la forme d'un séquent contenant deux parties :*

- *le contexte* Γ *qui est*
$$\Gamma := [v_1 : T_1; \ldots; v_m : T_m] :: [h_{m+1} := B_{m+1} : T_{m+1}; \ldots; h_n := B_n : T_n]$$
- *la conclusion* c *qui est un terme.*

Soit la présentation générale d'un séquent d'un but décrit dans la **Définition 4.12**.

$v_1 : T_1$
$\vdots$
$v_m : T_m$
$h_{m+1} := B_{m+1} : T_{m+1}$
$\vdots$
$h_n := B_n : T_n$
==========
c

où $v_1, \ldots, v_m$ sont des variables, $h_{m+1}, \ldots, h_n$ sont des hypothèses et c est une conclusion.

FIGURE 4.1 – La présentation générale d'un séquent dans Coq

Exemple 4.13 `Theorem` *even_xx :* $\forall$ *x, even (x + x) = true.*

Ce théorème énonce la proposition de notre exemple : l'addition d'un naturel à lui même donne un naturel pair.

Dans Coq, une preuve d'une proposition énoncée par un λ-terme bien typé p est le résultat de la recherche d'un λ-terme bien typé dont le type est p. La preuve se termine quand il n'y a plus de sous-buts ouverts. Si elle est terminée et son terme est bien typé et de même type que son énoncé, elle peut être enregistrée sous un identificateur qui correspond au nom du théorème. La construction de la preuve se fait à l'aide des tactiques.

C. But et variable existentielle

Dans nos travaux, nous avons besoin de récupérer des buts de la preuve. D'où nous nous intéressons à la façon avec laquelle Coq représente les buts. En effet, au cours d'une preuve, Coq donne un identificateur à chaque sous-but. Cet identificateur, appelé **uid**, est un entier naturel qui indexe une variable existentielle dans Coq et qui a comme type la proposition de ce sous-but. En appliquant une tactique à un sous-but, Coq attribue une valeur à cette variable existentielle. Cette valeur doit avoir le même type que celui de la variable existentielle.

Ces variables existentielles appartiennent à l'environnement local d'un sous-but mais elles n'appartiennent pas à son contexte. Elles peuvent être visualisées, avec leurs types qui correspond aux propositions des sous-buts ouverts, par la commande `Show` *Existentials*.

Exemple 4.14 *Nous considérons le théorème de l'* ***Exemple 4.13****. En commençant la preuve de ce théorème, Coq crée la variable existentielle qui correspond à ce but.*
La commande -`Show` Existentials.- donne

Existential 1 = ?12 : [x : nat |- even(x+x)=true].

La variable existentielle d'un but résolu ne peut plus être visible, cependant, elle est conservée dans l'environnement courant. Elle est utilisée par Coq pour calculer le terme de preuve.

4.2.2 Construction des preuves (Tactiques)

Le concept de tactique a été introduit pour la première fois dans [Gordon *et al.*1979]. Ensuite, il a été adopté par des environnements de preuve interactifs comme HOL [The HOL development team] et ISABELLE [The ISABELLE development team]. D'une façon générale, les tactiques sont les commandes des assistants de preuve pour prouver interactivement des théorèmes. Elles permettent de simplifier le but initial en des sous-buts. Il est possible d'appliquer une succession de tactiques pour résoudre une proposition jusqu'à ce que toutes les conjectures soient trivialement prouvées. Le choix des tactiques ainsi que leur ordre dépendent de l'intuition de l'utilisateur et du type du terme engendré par l'application de la tactique. Toute application d'une tactique doit respecter les règles de typage du noyau de Coq. Soit g le but courant, t une tactique et $g_1, g_2, \ldots, g_n$ les nouveaux sous buts engendrés par t appliquée à g. Chaque tactique possède une fonction associée qui permet de construire la solution du but g à partir des solutions des sous-buts $g_1, g_2, \ldots, g_n$.

Trois types de tactiques sont distinguées :
- *primaires* : qui sont un ensemble de tactiques prédéfinies ;
- *complexes* : qui sont des moyens de réduction d'un but de preuve dans le but de simplifier la recherche d'une preuve ;

– *automatiques* : qui sont des tactiques permettant de trouver la preuve en une seule étape en cachant tous les détails de la recherche. Elles peuvent utiliser le contexte (les définitions existantes), les tactiques primaires ou complexes ainsi que d'autres outils et moyens externes.

A. Tactiques primitives

Nous présentons certaines tactiques primitives de [Coq development team] que nous utilisons ultérieurement. Dans ce qui suit, nous expliquerons le comportement de chaque tactique.

Notation 4.15 *Le séquent seq correspond à la formule :*

$$\phi_{seq} := \forall (x_1 : A_1) \ldots (x_n : A_n), B_1 \to \ldots \to B_m \to C.$$

A.a. intro, intros et revert La tactique `intro` permet de transformer le séquent $seq_1 : \Gamma \vdash \forall (x : A), B$ en le séquent $seq_2 : \Gamma :: \{(x : A)\} \vdash B$. Cette tactique correspond à l'introduction de la variable x dans le contexte courant et applicable également dans le cas d'une implication logique (x n'apparaît pas dans B). Les séquents seq_1 et seq_2 ont la même formule. La tactique `intros` répète `intro` tant que c'est possible et ne cause jamais un échec.
Contrairement à `intro`, la tactique `revert` permet de déplacer une hypothèse du contexte vers le but. Coq assure que ce déplacement respecte la dépendance entre les hypothèses. En effet, l'application de `revert` sur le séquent $seq_1 : \Gamma_1 :: (H : A) :: \Gamma_2 \vdash B$ et sur l'hypothèse H donne l'un des deux séquents suivants : i) $seq_2 : \Gamma_1 :: \Gamma_2 \vdash \forall (H : A), B$ (si H apparaît dans B et n'apparaît pas dans aucune hypothèse de Γ_2), ou ii) $seq_3 : \Gamma_1 :: \Gamma_2 \vdash A \to B$ (si H n'apparaît pas ni dans B ni dans une hypothèse de Γ_2).

A.b. refine La tactique `refine` prend comme argument un terme qui peut contenir des trous (des jokers) représentant les nouveaux sous-buts.

A.c. cut La tactique `cut` implémente la règle de Modus ponens. L'application de le script suivant `cut` A au but courant B génère les deux buts suivants : $A \to B$ et A. Cette application correspond à un raffinement en utilisant le terme de preuve suivant : $t := ((fun(H : A) \to \_H)\_)$. Le premier trou, noté par $\_$, correspond au but suivant $A \to B$, i.e ce but est de même type que le but initial, mais avec une nouvelle assomption qui correspond à A. Le deuxième trou est un sous-but dont la proposition est A. Soit $\forall (x_1 : A_1), \ldots, (x_n : A_n), C$ la formule qui correspond au séquent initial B. Les deux nouvelles formules qui correspondent aux nouveaux séquents issus de l'application de `cut` sont : $\forall (x_1 : A_1) \ldots (x_n : A_n), A \to C$ et A.

A.d. assert La tactique `assert` est une variante de `cut` qui inverse l'ordre des preuves des nouveaux sous-buts introduits et qui personnalise l'identificateur de l'assomption introduite par la tactique. L'application de cette tactique se fait sous la forme suivante : `assert` $(H : T_H)$ sur un but courant B. Le terme de preuve correspondant est $t := ((fun(H : T_H) \to \_)\_)$. Le premier trou correspond à un nouveau sous-but de type T_H, et le deuxième correspond à un nouveau sous-but de type $\forall (x_1 : A_1) \ldots (x_n : A_n), B_1 \to, \ldots, \to B_m \to T_H \to C$. Dans le deuxième séquent, T_H est introduit comme hypothèse.

A.e. rewrite La tactique `rewrite` s'applique au but courant ou l'une d'hypothèses du contexte courant en utilisant des termes de la forme :

`forall (x`$_1$`:A`$_1$`) ... (x`$_n$`:A`$_n$`)eq` $term_1$ $term_2$, avec `eq` étant l'égalité dans Coq.

La tactique cherche le premier sous-terme dans le but courant qui correspond à $term_1$ et remplace toutes ses occurrences avec $term_2$. Les variables x_i seront résolues par unification, ou déclarées comme des nouveaux sous-buts.

A.f. admit La tactique `admit` permet d'échapper au but courant en le résolvant par un axiome. Elle déclare cet axiome qui a la même proposition que celle du but courant, et le nom de la preuve courante suivi de "_admitted" .

A.g. apply La tactique `apply` prend en argument un terme t, et essaie de faire correspondre le but courant à la conclusion de ce terme. Si ces derniers sont du même type ou unifiables, `apply` remplace le but courant par des buts correspondant aux prémisses non dépendant de type t. Le terme t peut être un axiome, une variable ou un théorème.

A.h. autorewrite La tactique `autorewrite` est une tactique automatique. Elle permet de réécrire avec une base de réécriture. Cette base correspond à un système de réécriture que la tactique utilise pour réécrire le but courant. `autorewrite` s'arrête si le but ne change plus.

A.i. inversion et inversion_clear La tactique `inversion` peut être appliquée sur une assomption qui appartient au contexte courant et qui a la forme suivante : $c_1(x_1, \ldots, x_n) = c_1(y_1, \ldots, y_n)$ avec c_1 est un constructeur d'un type inductif. L'application de `inversion` à une telle assomption permet de générer toutes les conditions nécessaires pour que cette assomption soit valide. La tactique `inversion_clear` supprime l'assomption après l'application de `inversion`.

B. Tacticielles

Les tacticielles[30] sont des tactiques composées qui permettent de réduire les scripts des preuves afin de les rendre plus courts et lisibles. Elles peuvent être utiles pour l'automatisation de la résolution des buts, mais leur rôle reste limité.

C. Création des nouvelles tactiques

Généralement, il existe deux façons de créer de nouvelles tactiques dans Coq : en Ltac ou en OCAML.

C.a. Ltac Ltac [Delahaye2000, Chlipala2011] est le langage de tactiques de Coq. Il permet de construire des tactiques paramétrées et récursives en utilisant des analyses par cas effectués sur le but ou un élément de contexte courants. Une tactique Ltac se définit en fonction des tactiques primitives et des tacticielles. Un script Coq est souvent construit par des commandes Ltac. C'est un langage de spécification très puissant et très utile. On peut même dire qu'il est un moyen indispensable pour créer de nouvelles tactiques dans Coq. La construction des tactiques

30. Le terme tacticielles (*tacticals*) est introduit dans [Bertot et Castéran2004].

ainsi que leurs applications s'effectuent suivant les règles de typage de Coq. Par contre, il existe des cas où écrire une tactique en Ltac devient une tâche longue et trop compliquée. Cette tâche peut être impossible si on souhaite utiliser des outils externes à Coq.

C.b. OCAML Une deuxième approche très efficace pour créer de nouvelles tactiques consiste à utiliser le même langage qui a permis le codage de Coq : OCAML. Cette approche offre tous les avantages de ce langage de programmation. D'une part, elle permet d'utiliser des outils externes à Coq, comme Spike dans notre cas. D'autre part, elle facilite la création des tactiques automatiques. Cette approche se base sur une interface existante entre Coq et OCAML. La correction des preuves construites par des tactiques développées en OCAML est toujours garantie par le noyau de typage de Coq. Ce noyau reste toujours intouchable et s'occupe de la validation du terme de preuve construit. Un exemple tel que celui présenté dans [Braibant2012] peut exprimer concrètement l'intérêt de l'utilisation de OCAML pour créer des nouvelles tactiques. Par contre, cette méthode souffre d'une absence de documentation, ce qui rend la tâche de l'implémentation fastidieuse.

4.2.3 Certification

Coq est un assistant de preuve certifié. Son architecture sépare clairement la partie moteur de preuve de la partie noyau. Néanmoins, les deux se basent sur la même théorie. En effet, comme nous avons mentionné auparavant, les termes et les types ont la même structure syntaxique, les même règles de construction. Ces règles permettent de construire et de vérifier les types en même temps.

Définition 4.16 (Certificat) *Pour une preuve d'une proposition p, un certificat dans Coq est un terme de preuve bien-typé de type p.*

Les moyens de construction des preuves sont les tactiques. En outre le respect de typage pendant la construction d'un terme de preuve, Coq effectue une vérification de ce terme à la fin de la preuve. Une preuve se terminer par la commande vernaculaire `Qed`. Cette commande construit le terme de la preuve qui correspond à la séquence des tactiques employées, vérifie que le type de ce terme est la proposition initiale et enregistre le théorème courant comme une définition qui relie l'identificateur de ce théorème à sa proposition et à son terme de preuve. La vérification du type s'effectue suivant des règles de typage. La relation entre ces règles et les tactiques est détaillée dans [Bertot et Castéran2004].

4.3 Raisonnement par récurrence noethérienne

Coq intègre la récurrence noethérienne sous sa forme structurelle. Nous présenterons dans la suite la définition d'une relation noethérienne qui permet de définir un ordre noethérien. Et, nous présenterons des différentes façons de la définition d'un schéma de récurrence.

4.3.1 Relation noethérienne

Une relation R définie sur un type A est noethérienne si tout élément du type A est accessible. L'accessibilité d'une relation est définie de la façon suivante :

```
Inductive Acc (A : Type) (R : A -> A -> Prop) (x : A) : Prop :=
    Acc_intro : (forall y : A, R y x -> Acc R y) -> Acc R x
```

```
For Acc: Argument A is implicit
For Acc_intro: Arguments A, R are implicit
For Acc: Argument scopes are [type_scope _ _]
For Acc_intro: Argument scopes are [type_scope _ _ _]
```

Cette définition permet d'exprimer qu'il n'existe aucun élément de A qui fait partie d'une séquence infiniment décroissante établie par R.

En se basant sur la définition `Acc`, Coq définie la relation noethérienne notée par *well_founded* de la façon suivante :

```
well_founded =
fun (A : Type) (R : A -> A -> Prop) => forall a : A, Acc R a
     : forall A : Type, (A -> A -> Prop) -> Prop
Argument A is implicit
Argument scopes are [type_scope _]
```

L'argument implicite A correspond à l'ensemble d'éléments sur lesquels nous souhaitons définir une relation noethérienne R. Cette définition vérifie que R est bien fondée si tous les éléments de A sont accessibles.

4.3.2 Récurrence structurelle

La récurrence structurelle se définit dans Coq par un schéma engendré par une définition récurrente d'une sorte ou d'une fonction. Le principe de récurrence sur un type inductif T est constitué par :

($\mathcal{A}$) un en-tête contenant des quantifications universelles et une quantification sur un prédicat P portant sur tous les éléments de T,

($\mathcal{B}$) des implications dont les prémisses sont appelées des prémisses principales,

($\mathcal{C}$) une conclusion exprimant le fait que la propriété P est vérifiée pour tout élément de T.

Exemple 4.17 *Soit le schéma de récurrence qui dérive de la définition inductive de la sorte nat.*
nat_ind : ∀ P : nat → Prop ($\mathcal{A}$ *),*
P 0 → (∀ n : nat, P n → P (S n)) ($\mathcal{B}$ *)*
→ ∀ n : nat, P n ($\mathcal{C}$ *)*

Cette définition exprime dans Coq le `Principe de Peano` introduit auparavant. Elle dérive de la définition de type nat.

Ainsi, nous pouvons définir un schéma de récurrence à partir d'un fonction.

Exemple 4.18
Fixpoint f (u1 : nat) {struct u1} : nat :=
match u1 with
| 0 ⇒ 0
| (S 0) ⇒ 0
| (S (S u1)) ⇒ 0
end.

```
Functional Scheme f_ind := Induction for f Sort Prop.
```
Le schéma de récurrence obtenu est le suivant :
```
f_ind : ∀ P : nat → nat → Prop (* A *) ,
        (∀ u1 : nat, u1 = 0 → P 0 0) →
        (∀ u1 n : nat, u1 = S n → n = 0 → P 1 0) →
        (∀ u1 n : nat,
          u1 = S n → ∀ _x : nat, n = S _x → P (S (S _x)) 0) (* B *)
        → ∀ u1 : nat, P u1 (f u1) (* C *)
```

Nous pouvons aussi définir le schéma de récurrence en variant aussi l'ordre de récurrence. Cette définition s'effectue à l'aide de la commande `Function` [Balaa et Bertot2000, Barthe *et al.*2006, Coq development team].

Exemple 4.19 *Soit wff un ordre de récurrence.*
```
Require Import Wf_nat.
Require Import Recdef.
Require Import Inverse_Image.

Function g (x : nat) {wf lt x } :=
  match x with
    | 0 ⇒ True
    | S x1 ⇒ g x1
  end.
```
Cette définition engendre les deux obligations de preuve suivantes qui consistent à la vérification de l'emploi de l'hypothèse de récurrence et à la vérification de la propriété noethérienne de l'ordre :

```
2 subgoals, subgoal 1 (ID 77)

  ============================
   ∀ x x1 : nat, x = S x1 → x1 < S x1

subgoal 2 (ID 78) is :
 well_founded lt
```

Conclusion

Dans ce chapitre, nous avons présenté l'environnement de preuve certifiée Coq. Son noyau jouera le rôle d'un vérificateur des preuves par récurrence implicite dans nos travaux. La récurrence noethérienne est déjà définie dans cet environnement. Dans ce qui suit, nous utiliserons des instances de cette définition pour valider des preuves par récurrence implicite générées par Spike ou construites à la Spike. Nous considérons le langage OCAML comme langage de développement des nos tactiques.

Chapitre 5

La tactique `Spike`

Sommaire

Notre première approche de certification des preuves par récurrence implicite est la fusion entre le démonstrateur des théorèmes Spike et l'environnement de preuve Coq. Elle consiste à créer une tactique automatique dans Coq qui permet en même temps de générer une preuve par récurrence implicite et de la certifier. Le générateur de la preuve est Spike et le vérificateur de la preuve est Coq. Le script Coq devient un certificat de la preuve de Spike s'il est validé par Coq.

Dans ce chapitre, nous montrerons tout d'abord nos motivations pour cette approche. Ensuite, nous présenterons le processus général de notre tactique `Spike`. À l'aide de la spécification GALLINA qui correspond à l'exemple de l'**Introduction**, nous illustrons chaque étape de la tactique.

5.1 Présentation générale de la tactique `Spike`

5.1.1 Motivations

Cette approche nous ne permettra pas uniquement de certifier des preuves par récurrence implicite, mais elle nous permettra aussi d'exécuter ce raisonnement -la récurrence implicite- dans l'environnement de preuve certifiée Coq. En effet, Spike et Coq sont deux assistants de preuve dont les types de raisonnement, les ordres de la logique et les natures des preuves construites sont largement différents. Pour certifier des preuves par récurrence à la Spike dans Coq, nous devons penser à : i) une représentation des spécifications conditionnelles égalitaires et leurs propriétés, et ii) une reconstruction de leurs preuves basées sur la récurrence implicite générées par Spike, dans Coq.

Des différents éléments de cette approche existent déjà :

- un générateur des preuves par récurrence implicite : Spike [Stratulat2008],
- un traducteur des preuves de Spike en script Coq : une bibliothèque dans Spike [Stratulat2010, Stratulat et Demange2011], et
- un vérificateur des preuves : le noyau de l'environnement de preuve certifiée Coq [Coq development team].

Nos travaux consistent premièrement à assembler les différentes pièces du puzzel afin de construire une tactique qui permet de prouver par récurrence implicite un théorème dans Coq. Deuxièmement, nous définirons la structure générale d'une spécification conditionnelle égalitaire dans Coq. Troisièmement, nous envisagerons la création des pièces manquantes telles que : une procédure d'extraction d'une spécification Spike à partir d'un script Coq et une procédure qui permet la réutilisation de script traduisant une preuve de Spike pour valider un théorème dans Coq. Notons que nous adapterons la traduction des preuves de Spike en script Coq en apportant des changements à la reconstruction des preuves de Spike dans Coq effectuée dans [Stratulat et Demange2011].

Enfin, nous souhaiterons automatiser tout le processus. En effet, l'intérêt de la récurrence implicite et de Spike est leur efficacité importante en raisonnement automatiquement sur des preuves de taille industrielle. Pour cette raison, il est commode de développer cette tactique qui implémente la récurrence implicite d'une façon automatique.

5.1.2 Schéma général

La fusion entre les deux assistants peut être effectuée par la création d'une nouvelle tactique appelée `Spike` dans Coq, et qui permet d'appeler le démonstrateur Spike. Le schéma fonctionnel général de la tactique `Spike` est représenté dans la **Figure 5.1**. Cette tactique peut être complètement automatique, i.e. elle peut résoudre le but dans une seule étape et sans précision

des arguments. En effet, elle permet, à partir de l'environnement Coq, de générer une spécification Spike, d'appeler ce dernier pour générer une preuve par récurrence implicite, de traduire cette preuve en script Coq, et enfin de valider ce script par le noyau de Coq. Dans le cas de sa validation, ce script peut être réutilisé pour résoudre un but dans Coq.

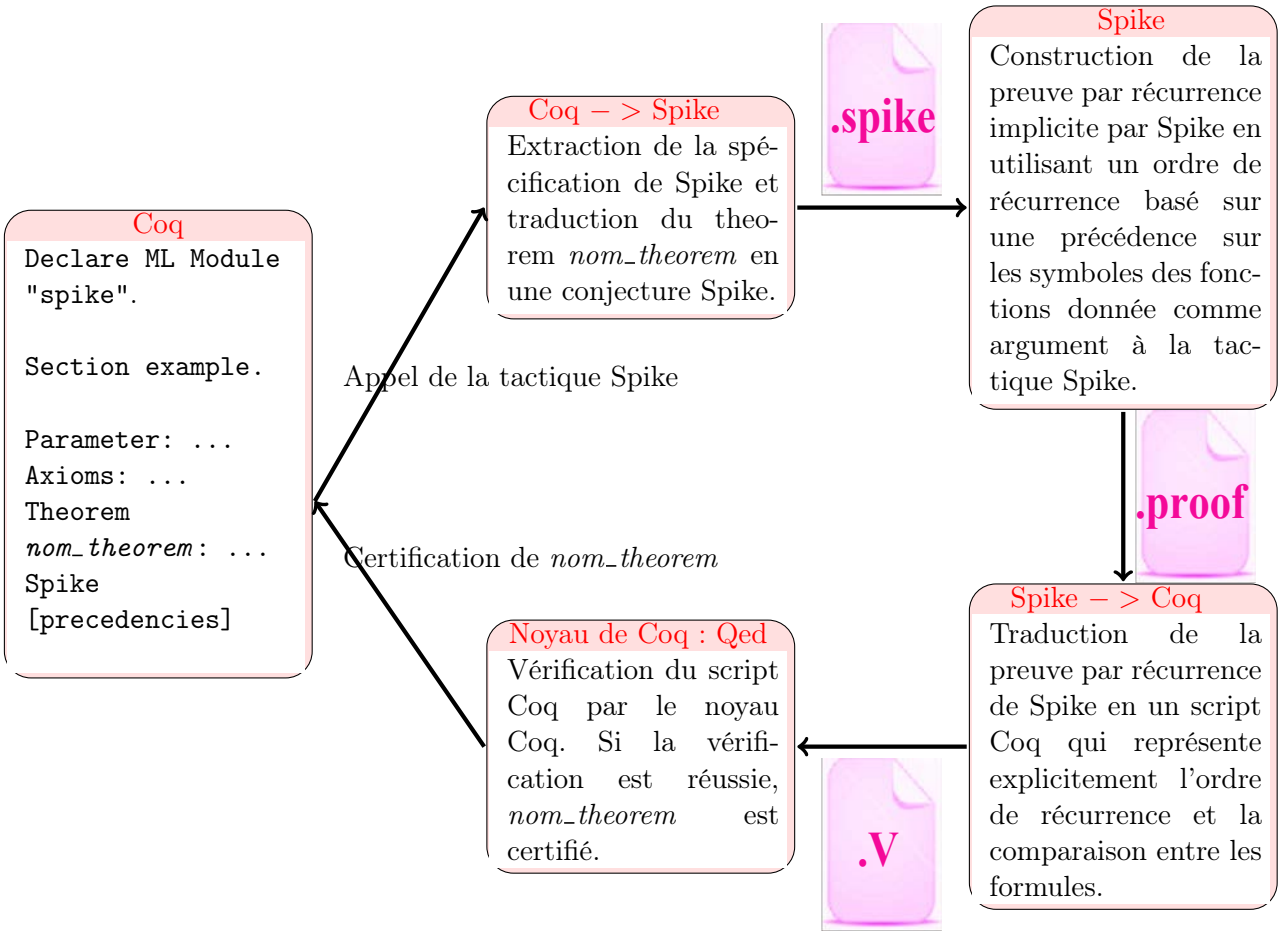

FIGURE 5.1 – Le schéma général de la tactique **Spike**

5.1.3 Description syntaxique

La tactique **Spike** est implémentée en OCAML et existe en quatre variantes :

-Spike.

-Spike equiv S_{equiv}**.**

-Spike greater $S_{greater}$**.**

-Spike equiv S_{equiv} **greater** $S_{greater}$**.**

Notons que S_{equiv} est une liste d'identificateurs Coq et $S_{greater}$ une liste des listes d'identificateurs Coq, sachant que nous adapterons la même définition que Coq pour la liste et que les identificateurs sont des symboles de fonctions. Ces arguments de la tactique serviront pour la construction de la précédence sur les symboles de fonction détallée dans la suite.

5.2 Extraction de la spécification Spike à partir d'une spécification GALLINA

La tactique commence par l'extraction de la spécification Spike à partir d'un script Coq. Une analyse du contexte courant et des énoncés des théorèmes est suffisante pour remplir toutes les informations nécessaires à Spike.

Les symboles de fonctions et les axiomes sont déterminés lors d'un seul parcours de tous les axiomes du script Coq qui sont enregistrés dans une seule structure contenant toutes les déclarations d'une section. Pour chaque axiome, on construit la clause égalitaire correspondante. Tout nouveau symbole rencontré sera rajouté à l'alphabet. Pour faciliter ultérieurement la traduction des preuves de Spike en script Coq, il existe des lignes de script Coq à fournir dans la spécification Spike. Contrairement aux travaux précédents [Stratulat2010, Stratulat et Demange2011], ces lignes sont fournis automatiquement par la tactique `Spike`. Plus de détails sur la nature de ces lignes sont développés dans le **Paragraphe 5.4**.

Seuls les théorèmes et les lemmes sous la forme des clauses égalitaires sont transformés. Tout autre forme peut engendrer l'échec de la preuve. Ceux qui sont prouvés appartiendront aux lemmes dans Spike, et ceux qui ne le sont pas appartiendront aux conjectures. Cette différenciation s'effectue dans Coq par la vérification que ce théorème (éventuellement ce lemme) soit bien défini dans la section courante. De même, l'alphabet se met à jour à chaque fois qu'on rencontre un nouveau symbole.

Enfin, la tactique peut déterminer à partir des arguments de la tactique -la précédence sur les symboles de fonction fournie par l'utilisateur- l'ordre de récurrence. Dans le cas où la tactique est appelée sans arguments, Spike définit sa propre précédence.

Si la spécification est construite, Spike est appelé pour construire une preuve par récurrence implicite comme c'est détaillé dans le **Chapitre 2**. En cas de succès de la preuve, Spike peut fournir, en plus de sa trace, sa traduction en script Coq dans un fichier .v. Ce fichier porte le même nom de la section concaténé au nom de théorème auquel on ajoute le suffixe "_complet".

5.2.1 Exemple d'une spécification GALLINA accepté par `Spike`

Dans ce paragraphe, nous mettons l'accent sur les commandes et les outils que nous allons utiliser pour représenter une spécification conditionnelle égalitaire, telle qu'une spécification Spike, dans Coq. Nous restons le plus fidèle à la description d'une spécification de Spike. Nous poursuivons avec le même exemple de l'**Introduction** dont nous présenterons le script Coq de l'**Exemple 5.1**.

Notons que la preuve de la conjecture de *even_xx* nécessite un lemme qui correspond à *addS1* qui doit être prouvé précédemment. Dans la suite, nous justifierons chaque commande de ce script et nous commencerons par la première ligne qui consiste à importer du module OCAML de la tactique "`Spike`".

Exemple 5.1

Declare ML `Module` *"spike".*

`Section` *evenoddintroduction.*

`Parameter` *add : nat → nat → nat.*
`Parameter` *even odd : nat → bool.*

`Axiom` *add1 :* $\forall$ *x, add 0 x = x.*
`Axiom` *add2 :* $\forall$ *x y, add (S x) y = S(add x y).*

Axiom even1 : even 0 = true.
Axiom even2 : even (S 0) = false.
Axiom even3 : ∀ x, odd x =false → even (S (S x))=true.
Axiom even4 : ∀ x, odd x =true → even (S (S x))=false.

Axiom odd1 : odd 0 = false.
Axiom odd2 : ∀ x, even x =true → odd (S x) = true.
Axiom odd3 : ∀ x, even x =false → odd (S x) = false.

Theorem addS1 : ∀ x y, add x (S y) = S (add x y).
Spike equiv [[even, odd]] greater [[even, add, S, 0, true, false], [add, S, 0]].
Qed.

Theorem even_xx : ∀ x, even (add x x) = true.
Spike equiv [[even, odd]] greater [[even, add, S, 0, true, false], [add, S, 0]].
Qed.

End evenoddintroduction.

*L'intégralité de ce script est fournie manuellement par l'utilisateur et il correspond à la spécification décrite dans l'**Exemple 2.18**.*

5.2.2 Spécification conditionnelle

Comme nous avons défini dans le **Chapitre 2**, une spécification conditionnelle égalitaire contient une signature -sortes et fonctions- et un ensemble d'axiomes. Dans la suite, nous monterons comment les déterminer à partir d'un script Coq tel qu'il est décrit dans l'**Exemple 5.1**.

A. Sortes

Comme nous avons présenté dans le **Chapitre 4**, les types inductifs sont utilisés pour spécifier des structures de données récursivement définies. Dans une spécification Spike, toute sorte se définit par : un symbole et un ensemble de constructeur. Donc, à partir de la forme syntaxique des types inductifs définie dans la **Sous-section 4.1.3**, l'identificateur du type inductif $ident_{IT}$ est un symbole de sorte et les identificateurs $\{ident_0, \ldots, ident_n\}$ sont des constructeurs. Le profils et l'arité de chaque constructeur sont déterminés à partir de son type dans la définition du type inductif $ident_{IT}$. En effet, la définition du profil et de l'arité des constructeurs s'effectuent de la même façon que toutes les fonctions. Donc, ils seront détaillés dans le paragraphe suivant.

Exemple 5.2 *Soient les définitions de deux types inductifs pédifinis dans Coq nat et bool suivantes : Inductive bool : Set :=*
| true : bool
| false : bool.
Inductive nat : Set :=
| O : nat
| S : nat → nat.

À partir des identificateurs de deux types inductifs nat et bool, la tactique construit l'ensemble des sortes suivant : $\mathscr{S} := \{nat, bool\}$ *et l'ensemble de constructeurs* $\mathscr{C} := \{0, S, true, false\}$.

B. Fonctions

Afin d'avoir une représentation axiomatique proche de celle de Spike, nous déclarons les symboles de fonctions en tant que paramètres, i.e. on ne précise pas des valeurs. Cette déclaration en utilisant la commande `Parameter` définit uniquement le profil d'une fonction. L'identificateur est le symbole de fonction et le type est son profil. En effet, la tactique détermine l'ensemble des symboles de fonctions d'une spécification au cours de la récupération des axiomes, des conjectures et des lemmes afin de décharger la preuve des symboles de fonction qui ne seront pas utilisés.

Les arités sont calculées à partir des types des fonctions. Sachant que les types des fonctions dans Coq sont des produits. Et tel que nous avons décrit dans le **Chapitre 4**, la décomposition d'un produit donne une paire qui contient une liste des types et un type. L'arité d'une fonction est la longueur de cette liste. Contrairement à Spike, nous nous considérons que des notations préfixes des fonctions.

Exemple 5.3 *Soient les deux lignes suivantes du script Coq de l'**Exemple 5.1** :*

```
Parameter add : nat → nat → nat.
Parameter even odd : nat → bool.
```

Donc, la tactique quand elle rencontre les symboles add, even ou odd, dans les définitions des théorèmes, elle détermine leurs profils qui correspondent aux types. La décomposition de produit (nat → nat → nat) qui correspond au type de add donne la paire ([nat ; nat], nat). L'arité de add est 2 *qui correspond à la longueur de la liste [nat ; nat]. Donc, la tactique définit add dans Spike comme suit : add__ : nat nat → nat*

C. Axiomes

Une spécification peut s'exprimer en style axiomatique, tel que le script de l' **Exemple 5.1**. Deux raisons nous avons fait penser à utiliser les axiomes. Premièrement, pour cette approche nous souhaitons rester le plus fidèle à une spécification conditionnelle égalitaire de Spike. Deuxièmement, certaines spécifications ne sont pas acceptées à cause d'un problème de définition du critère de récurrence dans le cas d'une définition en style fonctionnel, i.e. en utilisant `Fixpoint`. Parmi les exemples rejetés nous citons l'exemple suivant qui est mentionné dans [Henaien et Stratulat2013] :

Exemple 5.4

```
Fixpoint even (x : nat) :bool :=
match x with
| 0 ⇒ true
| S 0 ⇒ false
| S (S n) ⇒ even1 ( S ( S n))
end
with even1 (x : nat) : bool :=
match x with
| 0 ⇒ true
| S 0 ⇒ false
| S (S n) ⇒ if odd n then false
else even n
end
```

```
with odd (x : nat) : bool :=
match x with
| 0 ⇒ false
| S 0 ⇒ true
| S (S n) ⇒ odd1 ( S ( S n))
end
with odd1 (x : nat) : bool :=
match x with
| 0 ⇒ false
| S 0 ⇒ true
| S (S n) ⇒ if even n then odd n
else true
end.
```

Cette fonction est une autre version de la définition de la parité et l'imparité des naturels. Nous pouvons remarquer que les fonctions even, even1, odd et odd1 ont un niveau de mutualité plus élevé. Malheureusement, cette spécification n'est pas acceptée dans Coq, i.e. elle génère le message d'erreur suivant :

Recursive definition of even is ill-formed.

Notons qu'étant donnée cette spécification mise sous forme d'une spécification conditionnelle égalitaire Spike construit une preuve par récurrence implicite pour la conjecture $even(add(x, x)) = true$.

La vérification de la correction des axiomes par rapport à leur fonction est effectuée manuellement et séparément de la preuve. Sachant que souvent ses preuves sont relativement triviale.

Exemple 5.5 *Pour les fonctions even et odd de l'**Exemple 5.1** on a :*

```
Fixpoint even ( x : nat) : bool :=
  match x with
    |0 ⇒ true
    | S 0 ⇒ false
    | S (S x) ⇒ if (odd x) then false
                  else true
  end
```

```
with odd ( x : nat) : bool :=
  match x with
    | 0 ⇒ false
    | S x ⇒ if (even x ) then true
              else
                false
  end.
```

```
Theorem theven3 : ∀ x, odd x =false → even (S (S x))=true.
intro x.
simpl even.
destruct odd; auto.
Qed.
```

Dans cette partie de nos travaux, l'ensemble des axiomes est fournie manuellement. Et donc, la propriété de complétude forte n'est pas vérifiée automatiquement. Nous laissons à l'utilisateur de s'assurer que le système de réécriture dérivant des axiomes est fortement complet. Une façon simple permet de s'assurer que cet ensemble est complet est d'écrire les fonctions en format fonctionnel dans Coq. Si la définition est acceptée par Coq alors la propriété de complétude est assurée.

Nous avons choisi de séparer les axiomes par rapport à leurs preuves afin d'avoir une description de spécification conditionnelle égalitaires claire, séparée de reste de développement.

5.2.3 Conjectures et lemmes

Les conjectures, respectivement les lemmes, correspondent aux théorèmes non encore prouvés, respectivement aux théorèmes déjà prouvés. Les lemmes doivent être fournis manuellement toutefois ils peuvent être prouvés avec la tactique `Spike` ou d'autres tactiques de Coq.

Exemple 5.6 *Dans un premier temps dans la spécification **Exemple 5.1**, la conjecture à prouver est (∀ x y, add x (S y) = S (add x y)) et il n'existe pas des lemmes. Ensuite, pendant la preuve de even_xx la conjecture à prouver est (∀ x, even (add x x) = true) et il existe un seul lemme qui correspond à la conjecture de addS1.*

Les lemmes sont spécifiés manuellement comme les axiomes et ils sont aussi prouvés avant de commencer la preuve. Néanmoins, nous avons choisi de séparer les lemmes et les axiomes et séparer les preuves des axiomes et la preuve courante afin d'éviter toutes sortes de confusion entre les lemmes et les axiomes pendant l'extraction de la spécification. Surtout que les axiomes dans notre cas jouent un rôle très particulier qui consiste à la définition des fonctions définies.

5.2.4 Précédence sur les symboles de fonctions

Telle que nous avons détaillé auparavant, une spécification Spike peut contenir une définition de précédence sur les symboles de fonctions afin de définir l'ordre de la récurrence. Cette précédence est fournie comme argument de la tactique. Dans la description de la tactique, la liste des identificateurs S_{equiv} est de la forme $[symb_1, \ldots, symb_n]$ où $symb_1, \ldots, symb_n$ sont des symboles de fonctions équivalents. Ainsi, la liste est de la forme $[S_1, \ldots, S_n]$ où chaque S_i $(i \in [1..n])$ est une liste de la forme $[symb_1, symb_2, \ldots, symb_n]$. Le symbole $symb_1$ est plus grand que tous les symboles de l'ensemble $\{symb_2, \ldots, symb_n\}$. Ces symboles peuvent être des symboles de fonctions ou de constructeurs.

Exemple 5.7 *Dans l'exemple de l'**Introduction**, nous n'avons que les symboles even et odd qui sont équivalents, et le symbole even (resp. add) qui est plus grand que true, false, S, 0, add (resp. S et 0).*

5.3 Construction de la preuve

La construction de la preuve s'effectue comme nous avons détaillé dans le **Chapitre 2**. Une stratégie prédéfinie est utilisée. Nous considérons dans la suite la même stratégie définie dans l'**Exemple 2.19**. Le démonstrateur des théorèmes Spike fournit un trace de la preuve qui montre le raisonnement employé.

Exemple 5.8 *Soit la portion de la trace de preuve fournie pour la preuve l'**Exemple 2.20** suivante :*
«REWRITING : simplify by rewriting
$[229] even(add(u4, u4)) = false \Rightarrow false = true$ *;*
- rewriting at the position false/1/[1] :
$even(add(u4, u4))$
is simplified by : $even(add(u1, u1)) \rightarrow true$ *(from [50] of* C_1*)*
with substitution : <! u1, u4 !> into true
$[233] true = false \Rightarrow false = true$ *;»*

Ce message montre une étape de récurrence implicite où Spike effectue une réécriture de la conjecture courante ayant le numéro 229 *avec une hypothèse de récurrence, une instance de la règle* $even(add(u1, u1)) \rightarrow true$*. Cette règle est une conjecture de la preuve ayant le numéro* 50 *-cette information indiqué par le message (from [50] of* C_1*)-. Spike fournit aussi la substitution, <! u1, u4 !>, la position de terme à réécrire, false*/1/[1]*, et le terme obtenu par la réécriture, true. Enfin, il fournit le résultat qui consiste à la clause ayant le numéro* 223*.*

5.4 Traduction de la preuve Spike en script Coq

Pour traduire le raisonnement par récurrence implicite de Spike dans Coq, il est nécessaire de traduire les différentes étapes de la preuve, mais aussi l'ordre de la récurrence utilisé. La définition de l'ordre permet de valider les comparaisons effectuées sur les clauses égalitaires d'une preuve de Spike. En effet, ces clauses seront représentées par des termes Coq et donc nous devons définir leur abstraction pour avoir une représentation syntaxique. Cette abstraction s'effectue en utilisant une algèbre de termes fournie par COCCINELLE [Contejean *et al.*2007]. La structure générale de cette bibliothèque est détaillée dans [Contejean2005]. C'est en utilisant cette algèbre que nous définissons l'ordre de la comparaison. La suite correspond à l'automatisation et à la généralisation des travaux publiés dans [Stratulat2010, Stratulat et Demange2011].

Dans la suite, nous présenterons tout d'abord la bibliothèque COCCINELLE. Ensuite, nous expliquerons les trois éléments principaux de la traduction d'une preuve Spike :

1. la définition d'un ordre noethérien,
2. la définition de l'ensemble des clauses de la preuve, et
3. l'application du principe de récurrence noethérienne sur l'ensemble des clauses de la preuve.

5.4.1 COCCINELLE

COCCINELLE est une bibliothèque de Coq créée pour produire des preuves en Coq pour le démonstrateur automatique CiME [CiME Development Team]. Elle contient des modules mathématiques nécessaires pour effectuer la réécriture tels que : l'algèbre de termes, la réécriture générique, la théorie équationnelle AC et générique, ainsi que l'ordre RPO avec statut. Parmi les avantages de cette bibliothèque, nous mentionnons le fait de contenir les preuves des propriétés de toutes ces notions citées [Contejean *et al.*2007]. Nous utiliserons dans nos travaux les deux modules suivants :

1. Term qui définit une algèbre de termes à partir d'une signature donnée, et
2. RPO qui définit un ordre de récurrence sur les termes et fournit les preuves des propriétés de monotonicité et stabilité par instanciation.

Notons que toutes les informations nécessaires pour instancier ces modules sont déterminées automatiquement par notre tactique dont les détails sont fournis dans la suite.

5.4.2 Définition de l'ordre noethérien sur les clauses égalitaires

A. Définition de l'algèbre de termes

Le module RPO de COCCINELLE définit une algèbre de termes opérationnelle à partir d'une signature et d'un ensemble de symboles avec égalité décidable. En effet, notre tactique définit un module général qui s'appelle *Specif* et qui contient toutes les informations nécessaires à la construction du module RPO de COCCINELLE. Parmi ces informations, nous spécifions les symboles, leurs arités et leurs statuts.

Ensemble de symboles Il peut être défini comme une énumération des symboles des fonctions de la spécification de Spike. Pour éviter toute redondance d'un symbole avec des symboles prédéfinis dans Coq, on rajoute à chaque symbole le préfixe *id_*.

Exemple 5.9

```
Inductive symb : Set :=
```

```
| id_true
| id_false
| id_0
| id_S
| id_odd
| id_even
| id_add.
```

symb *est l'ensemble des symboles pour la spécification de notre exemple.*

Signature La définition de la signature commence par la définition de l'arité qui est une fonction qui associe à chaque symbole le nombre des paramètres et qui précise s'il est libre ou non. Dans COCCINELLE, un symbole peut être libre, AC (associatif commutatif) ou C (commutatif).

Exemple 5.10

```
Definition arity (f :symb) :=
  match f with
  | id_true ⇒ term_spec.Free 0
  | id_false ⇒ term_spec.Free 0
  | id_0 ⇒ term_spec.Free 0
  | id_S ⇒ term_spec.Free 1
  | id_odd ⇒ term_spec.Free 1
  | id_even ⇒ term_spec.Free 1
  | id_add ⇒ term_spec.Free 2
  end.
```

Status Nous considérons que tous les symboles de fonctions ont un statut multi-ensemble.

Exemple 5.11

```
Definition status (f :symb) :=
    match f with
  | id_true ⇒ rpo.Mul
  | id_false ⇒ rpo.Mul
  | id_0 ⇒ rpo.Mul
  | id_S ⇒ rpo.Mul
  | id_odd ⇒ rpo.Mul
  | id_even ⇒ rpo.Mul
  | id_add ⇒ rpo.Mul
end.
```

Précédence sur les symboles (Index) Chaque symbole de fonction f a un index qui est un naturel n supérieur à tous les indexes des symboles de fonctions plus petit que f et égale à tous les indexes des symboles de fonctions équivalent à f.

Exemple 5.12

```
Definition index (f :symb) :=
    match f with
  | id_true ⇒ 2
```

```
| id_false ⇒ 3
| id_0 ⇒ 4
| id_S ⇒ 5
| id_odd ⇒ 10
| id_even ⇒ 10
| id_add ⇒ 7
end.
```

Remarquons dans cet exemple que les symboles id_even et id_odd ont le même indice ce qui s'explique par le fait qu'ils sont équivalents.

Algèbre de termes Une algèbre de termes dans COCCINELLE est définie à partir d'une signature qui correspond au module SYMBOLS et un ensemble de variables qui correspond au NAT. La signature SYMBOLS est définie à partir de la section Specif qui contient **symb**, **arity**, **status** et **index**. Dans notre cas, les variables dans Spike sont toujours codées sur des naturels afin de garantir la décidabilité de la comparaison. Donc, les variables dans la définition d'algèbre de termes sont instanciées par le module NAT qui se base sur la sorte **nat** et des définitions nécessaires qui permettent de décider la comparaison sur les naturels.

Une fois que toutes les définitions sont bien spécifiées par la tactique, nous pouvons instancier le module TERMS de COCCINELLE par la commande suivante :

```
Module TERMS <: TERM_SPEC.TERM := TERM.MAKE' SYMBOLS NAT.
```

Cette définition permet de définir la structure élémentaire **terme** et nous obtenons :

```
Inductive term : Set :=
  | Var : variable → term
  | Term : symbol → list term → term.
```

Les variables abstraites variable et symbol correspondent respectivement à l'ensemble de variables et l'ensemble de symboles de fonction. La commande précédente d'instanciation de l'algèbre de termes permet de les instancier respectivement par l'ensemble de variables de l'algèbre de terme et l'ensemble de symboles de fonction.

B. Abstraction des termes Coq

Tout terme de Coq peut être abstrait par des fonctions de modélisation automatiquement définies par notre tactique et qui ont la même structure que celles des travaux de [Stratulat et Demange2011].

Exemple 5.13

```
Fixpoint model_bool (v : bool) : term :=
match v with
| (true) ⇒ (Term id_true nil)
| (false) ⇒ (Term id_false nil)
end.

Fixpoint model_nat (v : nat) : term :=
match v with
| (O) ⇒ (Term id_0 nil)
```

```
| (S x0) ⇒ let r0 := model_nat x0 in (Term id_S (r0:: nil))
end.
```

model_nat et model_bool sont les deux fonctions d'abstraction des termes associées à notre exemple.

L'abstraction s'effectue récursivement sur les termes comme suit :
- Un terme variable $x \in \mathscr{X}_s$, $s \in \mathscr{S}$ se transforme en $(model\_s\ x)$ où $model\_s$ est la fonction d'abstraction associée à s
- Un terme non-variable de la forme $f(t_1, \ldots, t_n)$ avec f étant un symbole de fonction se transforme en $(Term\ id\_f\ [t'_1; \ldots; t'_n])$ où $t'_1, \ldots, t'_n$ sont respectivement les abstractions des termes $t_1, \ldots, t_n$.

C. Ordre de récurrence sur des clauses égalitaires dans Coq

Afin de définir l'ordre sur les clauses égalitaires, la tactique utilise :
- une précédence sur les symboles des fonctions,
- un ordre *rpo*,
- une représentation multi-ensemble des poids des clauses égalitaires, et
- une extension multi-ensemble de l'ordre *rpo*.

La précédence sur les symboles de fonction est la définition *index* définie auparavant. Le reste des définitions sont définies par la suite.

C.a. Ordre *rpo* La bibliothèque COCCINELLE modélise RPO d'une façon générique en utilisant la précédence et le statut de chaque symbole de fonction. Donc, la tactique instancie le modèle général RPO de COCCINELLE par l'algèbre de termes déjà définie et via la commande suivante :

`Module` R := RPO.Make T1.,

où T1 est l'instance de module générique Terme de COCCINELLE pour notre spécification. Le module RPO de COCCINELLE établit toutes les propriétés fondamentales d'un ordre RPO telle que la réductible. Par conséquent, le module R établit cette propriété aussi.

C.b. Poids d'une clause égalitaire Nous associons à chaque formule F son poids W_F : (F, W_F). Nous considérons qu'un multi-ensemble est représenté par une liste dans Coq. Donc, le poids d'une formule est une liste des termes COCCINELLE correspondant aux abstractions des termes de cette formule. Afin de garantir la propriété de stabilité par substitution entre une formule et son poids, nous représentons chaque formule par le terme fonctionnel de Coq suivant : fun $(x_1, \ldots, x_n) \Rightarrow (F, W_F)$, où $(x_1, \ldots, x_n)$ est un vecteur de variables qui contient les variables de F.

Exemple 5.14
Definition F_50 := (fun u1 ⇒ ((even (add u1 u1)) = true, (Term id_even ((Term id_add ((model_nat u1) : : (model_nat u1) : :nil)) : :nil)) : :(Term id_true nil) : :nil)).

F_50 est la fonctionnelle associée à la formule (even (add u1 u1)) = true.

C.c. Extension multi-ensemble de l'ordre *rpo* Dans nos travaux, nous considérons l'ordre **rpo_mul** qui consiste à une extension multi-ensemble de l'ordre *rpo* défini sur des clauses égalitaires. Cet ordre est défini en utilisant la précédence sur les symboles de fonction *index* -rappelons que tous les symboles des fonctions ont le status multi-ensemble- et l'ordre *rpo*. Dans la suite, nous notons cet ordre par less. Cet ordre présenté dans [Stratulat2010, Stratulat et Demange2011] est prouvé qu'il est un ordre noethérien et stable sur par substitution dans Coq.

5.4.3 Définition de l'ensemble des clauses d'une preuve Spike

Afin de vérifier la correction d'une preuve par récurrence implicite générée par Spike dans Coq, il faut vérifier la correction de l'application du principe sur l'ensemble des formules de la preuve. Pour cela, il faut tout d'abord réaliser une traduction -par les fonctions introduites auparavant- de toutes les formules de Spike dans Coq. Ensuite, il faut effectuer un regroupement - dans une liste- de toutes les formules de la preuve. Néanmoins, une liste ne contient que des éléments de même type, c'est pour cette raison il faut mettre les fonctionnels sous une forme standard unique pour toutes les formules de la preuve.

A. Définition du type abstrait des fonctionnelles

Exemple 5.15 *Soit la preuve générée par Spike de la conjecture du théorème even_xx. Son ensemble des clauses est décri dans de l'**Exemple 2.21**. Nous avons par exemple les deux conjectures suivantes :*

- *even*(0) $=$ *true dont la définition est ((even 0) = true, (Term id_even ((Term id_0 nil) : :nil)) : :(Term id_true nil) : :nil).*
  , et
- *even*$(add(x, x))$ $=$ *true dont la définition est (**fun** u1* $\Rightarrow$ *((even (add u1 u1)) = true, (Term id_even ((Term id_add ((model_nat u1) : : (model_nat u1) : :nil)) : :nil)) : :(Term id_true nil) : :nil)).*

*Ces deux définitions se mettront dans la même liste, une liste qui contiendra toutes les fonctionnelles associées à toutes les clauses de la preuve citées dans l'**Exemple 2.21**. Donc, toutes les fonctionnelles doivent avoir le même type abstrait. Nous remarquons que toutes les clauses de l'**Exemple 2.21** ont au plus une variable de type nat. Donc, le type abstrait aura la forme* $fun(x : nat) \Rightarrow (F, W_F)$*, où* F *est une formule quelconque de la preuve et* W_F *est son poids. Nous obtenons dans Coq :*

Definition type_LF_50 := $nat \rightarrow$ *(Prop* $\times$ *(List.list term)).*

Dans le cas de $even(0) = true$*, la variable ne joue aucun rôle.*

B. Abstraction des termes et des clauses de Spike

Nous considérons tout d'abord que les symboles de sortes, des variables et des fonctions sont les mêmes dans Spike et Coq. L'abstraction des termes et des clauses s'effectue récursivement de la façon suivante :

1. Un terme variable dans Spike $x \in \mathscr{X}_s$, $s \in \mathscr{S}$ se transforme en un terme x dans Coq de type s, où $s : Set$ est un type inductif dans Coq qui correspond à la sorte s.
2. Un terme non-variable dans Spike de la forme $f(t_1, \ldots, t_n)$, $f : s_1, \ldots, s_n \rightarrow s_{n+1}, f \in \mathscr{F}$ et $t_1, \ldots, t_n$ sont des termes respectivement de sortes $s_1, \ldots, s_n$ se transforme en un terme $f(t'_1, \ldots, t'_n)$ dans Coq où $(t'_1, \ldots, t'_n)$ sont des termes Coq traduisant les termes Spike $(t_1, \ldots, t_n)$.

3. Une égalité dans Spike définie par la paire de deux termes (t_1, t_2) se transforme en un terme $t'_1 = t'_2$ dans Coq, où t'_1 et t'_2 deux termes Coq traduisant t_1 et t_2.
4. Une clause égalitaire dans Spike de la forme $e_1, \ldots, e_n \Rightarrow e$ se transforme en un terme $\forall (x_1 : s_1), \ldots, (x_m : s_m), e'_1 \rightarrow \ldots e'_n \rightarrow e'$ dans Coq , où $(x_1 : s_1), \ldots, (x_m : s_m)$ sont toutes les variables de la clause et $e'_1, \ldots, e'_n, e'$ sont des termes Coq traduisant les égalités $e_1, \ldots, e_n, e$.
5. Une clause égalitaire dans Spike de la forme $e_1, \ldots, e_n \Rightarrow$ se transforme en un terme $\forall (x_1 : s_1), \ldots, (x_m : s_m), e'_1 \rightarrow \ldots e'_n \rightarrow$ *False* dans Coq, où $(x_1 : s_1), \ldots, (x_m : s_m)$ sont toutes les variables de la clause et $e'_1, \ldots, e'_n$ sont des termes Coq traduisant les égalités $e_1, \ldots, e_n$.

C. Définition de l'ensemble des clauses d'une preuve

L'ensemble des clauses d'une preuve générée par Spike se traduit par une liste des fonctionnelles.

Exemple 5.16 *Soit la liste LF_50 de type abstrait : list type_LF_50. Cette liste contient toutes les fonctionnelles qui correspond à toutes les clauses de la preuve de Spike citées dans l'**Exemple 2.21**.*

`Definition` LF_50 := [F_50, F_61, F_70, F_67, F_73, F_83, F_87, F_102, F_111, F_108, F_114, F_134, F_130, F_135, F_157, F_166, F_163, F_169, F_197, F_193, F_198, F_224, F_228, F_229, F_233].

*Notons que F_50 est la fonctionnelle qui correspond à la clause $even(add(x, x)) = true$, F_61 est la fonctionnelle qui correspond à la clause $even(0) = true$ -qui est définie dans l'**Exemple 5.15**-, etc. Donc, la LF_50 est la traduction de l'ensemble des clauses de la preuve générée par Spike de l'**Exemple 2.20**.*

5.4.4 Traduction du principe de la récurrence implicite

A. Principe de la récurrence noethérienne

De même que [Stratulat2010], le principe de récurrence noethérienne se représente dans Coq comme un théorème dans une section appelée wf_subset et qui possède les paramètres suivants : quatre variables et deux hypothèses. Cette section permet de définir le schéma de récurrence sur un ensemble d'éléments ordonnés suivant une relation bien fondée.

```
Section wf_subset.
  Variable T : Type.
  Variable R : T → T → Prop.
  Hypothesis wf_R : well_founded R.
  Variable S : T → Prop.
  Variable P : T → Prop.
  Hypothesis S_acc : ∀ x, S x → (∀ y, S y → R y x → P y) → P x.
  Theorem wf_subset : ∀ x, S x → P x.
  Proof.
  intro z ; elim (wf_R z).
```

```
intros x H1x H2x H3x.
apply S_acc ; trivial.
intros y H1y H2y ; apply H2x ; trivial.
Qed.
End wf_subset.
```

La variable T est le type générique de toutes les formules de la preuve. La variable R définit l'opérateur de comparaison entre deux poids de formules. La relation représentée par P, respectivement S, donne la formule, respectivement le poids d'une formule, à partir d'une fonctionnelle. Le sens du théorème est le suivant : toute formule de ϕ qui peut être récupérée par P est valide si pour toute instance ψ plus petites que ϕ, par rapport à R, on a ψ est valide. L'hypothèse *wf_R* permet de donner l'argument justifiant que l'ordre de la récurrence utilisé est bien fondé. *S_acc* est l'argument de correction de l'application du principe de la récurrence noethérienne.

B. Théorème de l'application du principe de la récurrence implicite et sa preuve

L'application du principe de récurrence dans chaque preuve de récurrence implicite se traduit par un théorème principal appelé *main_num*. Ce théorème énonce que toutes les formules employées dans la preuve sont correctes en supposant que pour toute formule on peut utiliser des formules plus petites.

Exemple 5.17 *Soit le théorème qui correspond à notre exemple even_xx :*

Lemma main_50 : ∀ F, In F LF_50 → ∀ u1, (∀ F', In F' LF_50 → ∀ e1, less (snd (F' e1)) (snd (F u1)) → fst (F' e1)) → fst (F u1).

La preuve de *main_50* est principalement une analyse par cas sur toutes les formules de *LF_50*. Chaque formule est validée suivant les mêmes étapes de résolution correspondantes de Spike. Pour chaque application d'une règle d'inférence dans la preuve de récurrence implicite, la bibliothèque de traduction dans Spike génère un script Coq équivalent permettant d'effectuer le même raisonnement dans Coq.

C. Traduction de l'application des règles d'inférence en script Coq

Nous effectuons la traduction des règles d'inférence une à une. Nous essayons toujours de rester conforme avec la preuve de Spike. En effet, chaque traduction d'une règle d'inférence doit générer dans Coq les mêmes clauses générées par la même règle dans Spike. Cette condition garantit la cohérence entre la preuve de Spike et sa traduction. Ainsi, elle diminue les échecs de la tactique causés par des échecs de la traduction de la preuve. Nous ne pouvons pas prétendre que la méthode de traduction proposée est optimale, i.e. qu'elle garantit la cohérence entre toutes les preuves de Spike et leurs traductions. Néanmoins, nous avons apporté des différentes amélioration aux travaux proposés dans [Stratulat et Demange2011].

Dans nos travaux, nous avons remplacé la définition des fonctions suivant le style fixpoint par la définition axiomatique. Ceci nous a permis d'améliorer la traduction des applications des règles de Spike, en particulier les règles traitant des clauses égalitaires.

Dans les travaux précédents [Stratulat et Demange2011], la méthode d'identification des axiomes non-conditionnels utilisés par **Rewriting** de Spike consiste au déploiement "**unfolding**" de la définition de `fixpoint` de symbole de la fonction correspondante après avoir choisi la branche de la définition qui valide la condition de l'axiome dans le contexte courant de la preuve.

L'opération **unfolding** ne suit pas exactement le raisonnement Spike, i.e. il se peut qu'elle génère pas les mêmes clauses que dans la preuve de Spike. Une telle situation engendre certainement l'échec de la validation de la preuve et par suite l'échec de la tactique.

Exemple 5.18 *Soient la définition fonctionnelle de add, la conjecture* $\forall$ *y, add 0 (S y) = S (add y 0) et une opération de réécriture non conditionnelle en utilisant l'axiome add1. Si l'opération de réécriture est effectuée avec* `unfold` *alors nous obtenons :*

```
Fixpoint add (u1 u2 : nat) :=
  match u1 with
    | 0 ⇒ u2
    | S x ⇒ S ( add x u2)
  end.
Axiom add1 : ∀ x : nat, add 0 x = x.
Axiom add2 : ∀ x y : nat, add (S x) y = S
(add x y).
Goal ∀ y, add 0 (S y) = S (add y 0).
  intros.
  unfold add.
```

```
y : nat
============================
 S y =
 S
  ((fix add (u1 u2 : nat) {struct u1} :
    nat :=
      match u1 with
      | 0 ⇒ u2
      | S x ⇒ S (add x u2)
      end) y 0)
```

*L'opération de réécriture de l'**Exemple 5.18** peut être effectuer de la façon suivante :*

```
  Goal ∀ y, add 0 (S y) = S (add y 0).
  intros.
  rewrite add1.
```

```
y : nat
============================
 S y = S (add y 0)
```

*Cette traduction est plus conforme à une preuve Spike que celle de l'**Exemple 5.18**.*

Grâce à la représentation axiomatique, la traduction des opérations de réécriture est améliorée. La traduction s'effectue de la façon suivante :

i) La réécriture non-conditionnelle du but courant : tout axiome non-conditionnel peut être ajouté à la base de réécriture. Par exemple : le script suivant ajoute les axiomes non conditionnels de notre exemple précédent à la base de réécriture *rewrite_axioms*

Exemple 5.19
```
Hint Rewrite add1 add2 even1 even2 odd1 : rewrite_axioms.
Ltac rewrite_ax := autorewrite with rewrite_axioms.
```

Dans la traduction, l'application d'une règle de réécriture non-conditionnelle est remplacée par *rewrite_ax*.

ii) La réécriture conditionnelle du but courant : la traduction de l'application de la règle de réécriture à une conjecture avec un axiome non-conditionnel est traduite en `rewrite` *name_of_conditional_axiom*.

iii) La réécriture non-conditionnelle d'une hypothèse H du but courant : elle est traduite dans *normalize* `with` *rewrite_axioms* `in` *H*, où *normalize* est une tactique qui permet de réécrire H avec la base *rewrite_axioms*.

5.5 Validation du théorème initial

Si les étapes précédentes : extraction de spécification, construction et traduction de la preuve se terminent avec succès. Alors, on obtient un théorème dans la traduction de la preuve qui a le même énoncé que le théorème initial (à un renommage près). Ce théorème équivalent est prouvé par récurrence implicite et sa preuve est générée et traduite en script Coq par un bibliothèque de Spike. Le fichier contenant le script Coq est validé par son noyau avec la commande *coqc*. Si la validation se termine correctement, il sera importé dans la section courante. Le théorème initial peut être prouvé dans une seule étape en appliquant ce théorème équivalent et en effectuant la correspondance entre les symboles de fonctions de deux sections. Cette correspondance est nécessaire parce que les deux théorèmes existent dans deux sections différentes. Ainsi, les symboles de fonctions, même s'ils portent des noms identiques, sont considérés différents dans Coq.

Exemple 5.20 `Require Import` *evenoddintroduction_even_xx_complet.*
`apply` *true_91 ;* `repeat (apply` *even* | `apply` *odd* | `apply` *add).*
evenoddintroduction_even_xx_complet est le fichier généré par Spike et qui contient le théorème true_91. Ce théorème est équivalent au théorème initial even_xx. La preuve de true_91 est la traduction de sa preuve par récurrence implicite construite par Spike. Ce fichier est validé par le compilateur de Coq : ***coqc****.*

5.6 Résultats et limites

Ces travaux ont fait le sujet de la publication [Henaien et Stratulat2013]. Et ils sont utilisés pour valider des conjecture de la preuve d'un algorithme de conformité du protocole ABR.

5.6.1 Validation d'un algorithme de conformité du protocole ABR

Notre intérêt pour cet exemple réside dans le fait qu'il a déjà été traité par Spike [Rusinowitch *et al.*2003, Stratulat et Demange2011]. Nous allons automatiser la certification de certaines preuves de conjectures qui font partie de la validation d'un algorithme de conformité du protocole ABR [31] dans Coq. La certification manuelle de ces preuves, qui sont déjà construites par Spike et qui sont de grande taille s'avère fastidieuse. Pour cette raison, cette tâche doit se faire avec le minimum d'intervention de la part de l'utilisateur.

A. Protocole ABR

ABR est un protocole des réseaux ATM [32] qui assure une bonne gestion des débits de données entre des applications partageant simultanément un lien physique. L'ABR offre à un utilisateur un débit minimal qui peut changer au cours du temps suivant la charge de réseau et la disponibilité de la ressource partagée. Cette flexibilité est la propriété la plus importante de ce protocole, sa gestion est cependant un défit pour le fournisseur. L'utilisateur reçoit toujours la nouvelle valeur du débit pour adapter ses applications à cette valeur. Cependant, le fournisseur vérifie si le débit d'un utilisateur est conforme à une valeur de conformité qui est variable au cours du temps, et qui se calcule suivant un algorithme de conformité. Cet algorithme s'exécute sur une machine entre l'utilisateur et le fournisseur. Une nouvelle valeur de débit est envoyée du fournisseur à

31. *Available Bit Rate*
32. *Asynchronous Transfert Mode*

la machine via des cellules RM [33]. Le débit, le temps d'arrivée des cellules RM et la valeur de conformité sont enregistrées dans un buffer. A un instant donné, la valeur de conformité dépend du débit et le temps d'arrivée des cellules RM enregistrées.

B. Acr et Acr1 : deux algorithmes de conformité

Parmi les différents algorithmes de conformité, nous nous intéressons à Acr et Acr1. Ces deux protocoles [Stratulat et Demange2011] enregistrent toutes les valeurs des cellules RM et offrent un contrôle de conformité optimal du côté des utilisateurs. En effet, un utilisateur est informé du changement de la valeur de débit le plus tôt dès son augmentation, et le plus tard dès son diminution. Selon le forum d'ATM, Acr est considéré comme un algorithme standard de référence par rapport aux autres algorithmes de conformité. Toutefois, Acr1 est plus efficace puisque qu'il donne une valeur de conformité d'une façon instantanée. Contrairement à Acr, Acr1 prévoit une planification dans le futur du débit des cellules RM. Ce qui permet d'effectuer la plupart des calculs quand les cellules RM arrivent à la machine et non pas quand la valeur de conformité est demandée.

C. Extensions et statistiques de la tactique `Spike`

Dans le but de traiter des preuves plus complexes, nous proposons les deux extensions de la tactique `Spike`.

C.a. Extensions

SpikeWithFullind_aug [des contraintes d'ordre] qui intègre la technique d'augmentation dans la stratégie de la preuve. Cette technique présentée dans [Boyer et Moore1988b] et implémentée dans Spike [Armando *et al.*2002] permet d'augmenter la base factuelle de contexte des conjectures de simplification.

SpikeWithIndPriorities [stratégie de récurrence] [des contraintes d'ordre] qui décide quels sont les termes d'une conjecture qui peuvent être instanciés. Donnée comme argument de la tactique, la stratégie de récurrence établit la priorité entre les symboles de fonctions qui peuvent apparaître à la racine de ces termes.

C.b. Statistiques Nous avons utilisé la tactique `Spike` et ses extensions pour traiter des conjectures de [Stratulat2010, Stratulat et Demange2011]. Le **Tableau 5.1** illustre le nombre des lemmes, le nombre des hypothèses ainsi que la tactique employée avec son état de paramétrage et son temps d'exécution. `Aug` et `Ind` sont les abréviations respectives de `SpikeWithFullind_aug` et `SpikeWithIndPriorities`. Ces 33 conjectures font environ 50 % des conjectures de la preuve de validation d'un algorithme de conformité du protocole ABR. La tactique `Spike` est complètement automatique pour 60% de ces conjectures (21 conjectures) comme montre la colonne `Paramètres` de **Tableau 5.1**. Dans le reste des cas, il faut définir la précédence sur les symboles de fonction pour la tactique `Spike`.

5.6.2 Limites

Cette approche est une implémentation de l'**Algorithme 3.1** ayant Spike comme ATP et Coq comme ITP. Par conséquent, elle possède les limites déjà citées dans le **Chapitre 3** telles

33. Resource Management

que :

- `L'échec de l'extraction d'une spécification Spike à partir d'un script Coq :`
  Ce cas est dû à la différence entre les langages de spécification de Spike et de Coq. En effet, GALLINA est un langage très expressif. Néanmoins, notres approche est limitée à des spécifications GALLINA transformables en spécifications conditionnelles égalitaires. Pour l'instant, il existe certaines spécifications acceptées par Spike mais qui ne sont pas traitées par notre tactique telles que les spécifications avec des conditions non booléennes.
- `L'échec de la génération de la preuve :`
  Ce cas est engendré par une erreur pendant la construction de la preuve par Spike. Comme nous avons expliqué dans **Chapitre 2**, la recherche peut échouer, peut aboutir à un contre exemple ou peut ne pas se terminer. Ces trois cas engendrent l'échec de la tactique. Nous constatons que le cas de la non terminaison de la recherche de la preuve est le plus rencontré.
- `L'échec de la traduction de la preuve :`
  Ce cas correspond à un problème de traduction de la preuve. Nous ne fournissons aucune preuve formelle qui montre la cohérence entre toute preuve de Spike et sa traduction. Ainsi, cette étape est relativement longue par rapport à l'étape de la construction de la preuve. De plus, il existe des raisonnements dans Spike qui ne possèdent pas des traductions tels que le raisonnement arithmétique dans Spike. Ce raisonnement dans Spike se base sur des procédures décisionnelles compliquées [Armando *et al.*2002] et il est déjà intégré dans Coq à l'aide de la tactique `omega`.
- `L'échec de la validation de la preuve :`
  Cet échec peut être causé par deux raisons : un échec de la traduction ou un théorème non valide. Notre tactique `Spike` peut distinguer ces deux cas. Néanmoins, nous ne pouvons pas affirmer le cas de la non validation d'un théorème.

Nous notons aussi que pour cette approche la propriété de la complétude forte du système de réécriture induit par l'ensemble d'axiomes n'est pas établie automatiquement. Ainsi, les preuves de la validation des axiomes sont effectuées manuellement.

De point de vue pratique, cette approche est relativement coûteuse en termes de temps d'exécution. L'étape de la validation est l'étape la plus longue.

Conclusion

Dans ce chapitre, nous avons présenté un premier essai de l'intégration du raisonnement par récurrence implicite dans Coq. La fusion de Coq et Spike nous a permis de certifier certaines conjectures de validation d'un algorithme de conformité du protocole ABR. Ces travaux ont fait le sujet de la publication [Henaien et Stratulat2013].

#	Nom	Lemmes	Hypothèses	Tactique	Paramètres	Temps (s)
1.	firstat_timeat	1	0	Spike	non	3.021
2.	firstat_progat	1	0	Spike	non	3.159
3.	sorted_sorted	0	0	Spike	non	2.295
4.	sorted_insat1	4	0	Aug	oui	14.233
5.	sorted_insin2	4	0	Aug	oui	15.400
6.	sorted_e_two	0	0	Spike	non	2.039
7.	member_t_insin	2	0	Spike	oui	8.558
8.	membert_insat	3	0	Spike	non	11.317
9.	member_firstat	2	0	Spike	non	8.471
10.	timel_insat	0	0	Spike	non	2.402
11.	erl_insin	0	0	Spike	oui	2.512
12.	erl_insat	0	0	Spike	non	2.488
13.	erl_prog	2	0	Spike	oui	11.768
14.	time_progat_er	1	0	Spike	non	4.238
15.	timeat_tcrt	0	0	Spike	non	3.225
16.	timel_timeat_max	2	1	Aug	oui	9.191
17.	null_listat	1	0	Spike	non	4.143
18.	null_listat1	0	0	Spike	non	1.958
19.	cons_insat	0	0	Spike	non	1.987
20.	cons_listat	0	0	Spike	non	2.004
21.	progat_timel_erl	3	0	Aug	oui	11.621
22.	progat_insat	3	1	Aug	oui	44.422
23.	progat_insat1	3	0	Aug	oui	17.055
24.	timel_listupto	0	0	Spike	non	2.295
25.	sorted_listupto	3	0	Aug	oui	13.042
26.	time_listat	1	0	Spike	non	5.720
27.	sorted_cons_listat	3	1	Ind	oui	16.699
28.	null_wind2	0	1	Spike	non	3.382
29.	timel_insin1	1	0	Spike	oui	4.456
30.	null_listupto1	0	0	Spike	non	1.949
31.	erl_cons	0	0	Spike	non	2.415
32.	no_time	1	1	Spike	non	7.129
33.	final	2	1	Spike	non	8.330

TABLE 5.1 – Statistiques sur les preuves ABR.

Chapitre 6

Certification automatique des preuves par récurrence implicite à la Spike dans Coq

Sommaire

Notre deuxième approche de certification des preuves par récurrence implicite de Spike est l'intégration directement de ce raisonnement dans le moteur de preuve de Coq. Elle consiste à la création d'une tactique automatique qui permet de construire une preuve par récurrence implicite et de la certifier. La construction de la preuve s'effectue à la Spike. Contenant des hypothèses de récurrence non validées, le terme de preuve obtenu n'est pas valide. Pour cela, il est nécessaire d'effectuer une étape de validation de la preuve de la même façon que la tactique `Spike`. Le certificat dans ce cas est le terme de preuve finale accepté par Coq.

Dans ce chapitre, nous montrons tout d'abord nos motivations dont certaines résolvent des limites de notre première approche. Ensuite, nous présenterons la procédure générale de notre tactique. Ainsi, nous expliquerons le passage dans le deux sens entre un environnement basé sur la logique égalitaire celui de Spike et un environnement basé sur la logique d'ordre supérieur celui de Coq. Comme, nous mettrons l'accent sur la traduction des règles de Spike qui implémentent la récurrence implicite. Enfin, nous montrerons comment le théorème initial est validé par récurrence implicite.

6.1 Présentation générale de la tactique `StrategieSpike`

6.1.1 Motivations

Pour cette partie de travail, notre objectif au départ était l'élimination de l'utilisation de l'outil externe Spike. Cet objectif a entraîné l'intégration de raisonnement par récurrence implicite dans le moteur de preuve de Coq. Conformément à ce raisonnement, il faut un moyen qui permettra l'utilisation des hypothèses de récurrence qui sont des assertions non encore prouvées au cours d'une preuve dans Coq et de les valider à la fin.

Nous envisagerons aussi d'améliorer et d'adopter la traduction des règles d'inférence pour la nouvelle approche. En effet, l'apport par rapport à la tactique `Spike` est le fait que la preuve par récurrence implicite sera construite et validée dans le même environnement Coq. Pour chaque clause de la preuve, nous pouvons utiliser la même tacticielle pour la prouver pendant la construction et pour la valider pendant la certification de la preuve. Cette proposition permet de garantir la cohérence entre la preuve et sa traduction. En effet, dans cette approche nous pouvons vérifier à chaque étape de la construction de la preuve que les clauses générées à la Spike correspondant exactement aux sous-buts générés dans Coq. Il suffit de comparer les clauses à la Spike aux celles de nouveaux sous-buts dans Coq. Néanmoins, nous voyons que cette vérification est inutile parce que le cas où la procédure à la Spike ne génère pas les mêmes clauses que Spike ne causera plus d'échec, i.e. tant qu'on utilise pour chaque clause la même tacticielle pendant la construction et la validation.

6.1.2 Schéma général

Dans la **Figure 6.1**, nous présentons le schéma général de la méthode de certification des preuves par récurrence à la Spike directement et automatiquement dans Coq. En effet, cette méthode consiste à simuler une stratégie à la Spike dans Coq. En partant de l'environnement Coq, nous fournissons l'environnement, le contexte et la clause de séquent courant à une procédure Spike qui correspond à sa stratégie générale. Cette procédure est développée avec les mêmes procédures correspondant aux règles d'inférence de Spike et appartenant à son code source. Elle essaye de résoudre le but courant de la même façon dont Spike traite une clause. La procédure retourne une tacticielle qui traduit une règle d'inférence de Spike et qui s'applique au but courant. Chaque règle d'inférence possède une tacticielle générique équivalente à elle

dans Coq. L'application de cette tacticielle sur le but courant engendre sa simplification et la génération des nouveaux sous-buts calculés à la Spike. Ce processus se répète jusqu'à la fin de la preuve dans Coq, i.e. jusqu'à ce qu'il ne reste aucun sous-but ouvert. À ce niveau, une pré-preuve basée sur la récurrence implicite est construite. Nous considérons que la pré-preuve est construite par récurrence implicite parce qu'elle est construite par une stratégie composée des règles d'inférence de Spike qui implémentent la récurrence implicite, comme nous l'avons montré auparavant. Dans, ces travaux nous gardons la même stratégie prédéfinie définie dans le **Chapitre 2**.

La pré-preuve est validée de la même façon que celle proposée dans le **Chapitre 5**. Nous commençons tout d'abord par la création de l'algèbre de termes et l'ordre sur les termes correspondant. Ensuite, nous cherchons un type générique des paires de toutes les clauses de la pré-preuve et leurs poids, et nous créons la liste de toutes les fonctionnelles auxquelles nous appliquons le principe de récurrence noethérienne sur les clauses. Dans ce cas, la liste contiendra les fonctionnelles correspondant à toutes les clauses de la pré-preuve basée sur récurrence implicite. Enfin, le théorème initial est prouvé comme une conséquence inductive de cet ensemble de clauses.

Cette méthode nécessite d'effectuer les trois étapes suivantes :

1. La traduction dans les deux sens entre Coq et la procédure à la Spike en se limitant à la sous-classe des termes de GALLINA acceptés par Spike ;
2. L'association à chaque règle d'inférence de sa tacticielle équivalente en prenant en considération le cas des règles qui introduisent des hypothèses de récurrence ; et
3. La validation de la preuve en justifiant toute hypothèse de récurrence en définissant un ordre de récurrence pour effectuer la comparaison.

Dans la suite, nous détaillons chacune de ces étapes.

6.2 Traduction dans les deux sens entre Coq et Spike

Dans notre cas, la certification nécessite la réalisation d'une traduction dans les deux sens entre un terme de Coq et une clause de Spike. La traduction de Coq vers Spike permet d'utiliser des procédures à la Spike. La traduction inverse permet la reproduction du calcul à la Spike dans Coq.

Pour lancer la procédure de la stratégie Spike, il faut fournir une signature, des axiomes, des conjectures, des lemmes et des hypothèses de récurrence.

Comme nous l'avons déjà présenté dans le **Chapitre 4**, un environnement courant dans Coq peut contenir une grande quantité de définitions et de déclarations. Un passage de cet environnement entier à la stratégie Spike peut causer l'échec de construction de la preuve ou sa divergence. Pour cela, nous effectuons le filtrage nécessaire.

Dans la suite, nous expliquerons le passage entre un terme de Coq et une équation conditionnelle Spike.

6.2.1 De Coq vers Spike

A. D'un terme Coq vers une clause de Horn à la Spike

Parmi les termes définis par la structure syntaxique générique dans Coq, nous ne considérons qu'un terme qui permet de représenter une clause égalitaire. Pour cela, nos procédures de tra-

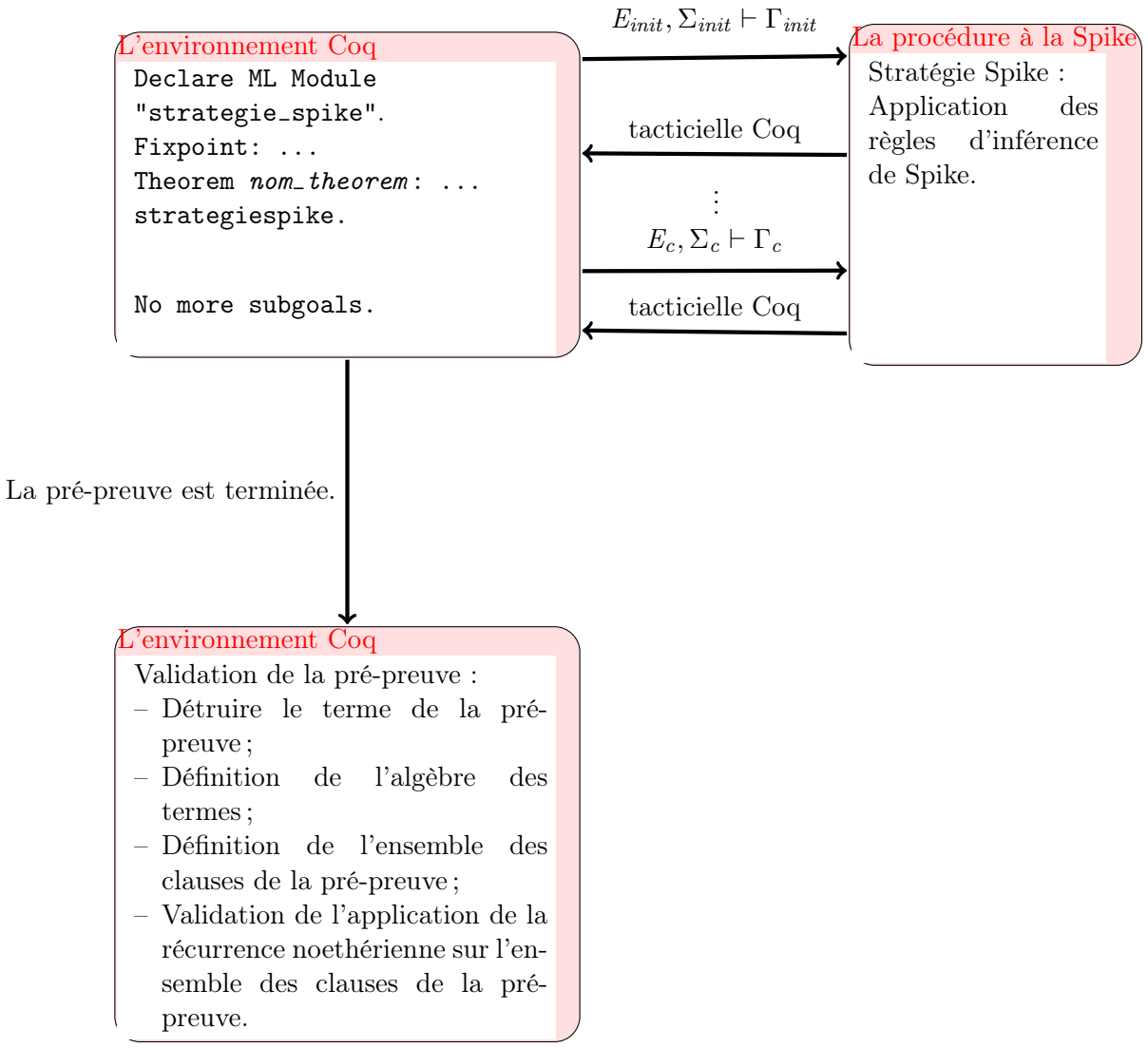

FIGURE 6.1 – Le schéma général de la tactique `StrategieSpike`

duction seront partielles par rapport aux termes GALLINA seuls les termes ayant des variables quantifiées universellement et qui ont la forme d'une égalité ou d'une égalité conditionnelle seront traduits.

Définition 6.1 (Termes d'égalité et d'égalité conditionnelle)

– *La forme générale de* ***terme d'égalité*** *est la suivante :*

$$\forall x_1 : T_1, \ldots, x_n : T_n, e = t$$

– *La forme générale de* ***terme d'égalité conditionnelle*** *est la suivante :*

$$\forall x_1 : T_1, \ldots, x_n : T_n, e_1 = t_1 \rightarrow \ldots \rightarrow e_m = t_m \rightarrow e = t.$$

Comme nous l'avons déjà évoqué dans le **Chapitre 4**, les variables universelles et les implications logiques ont la même représentation logique correspondant au produit. Pour cette raison, nous pouvons avoir la représentation suivante illustrant des termes d'égalités conditionnelles :

$$\forall x_1 : T_1, \ldots, x_n : T_n, H_1 : e_1 = t_1, \ldots, H_m : e_m = t_m, e = t$$

Sachant que pour tout $i \in [1..m]$ il n'y a aucun H_i qui apparaît dans l'un des termes suivants $e_1, t_1, \ldots, e_m, t_m, e, t$.

La traduction d'un terme t d'égalité ou d'égalité conditionnelle dans Coq en une égalité conditionnelle s'effectue par l'**Algorithme 6.5**. Coq possède une procédure appelée *decompose_prod* qui permet de décomposer un produit que nous utilisons pour séparer la liste des assomptions (les variables et les conditions) de la conclusion. Cette séparation nous permet de vérifier si le terme t est sous la forme d'un terme d'égalité ou d'égalité conditionnelle. Les variables d'un terme peuvent être déterminées par la procédure *decompose_prod_assum*. Une nouvelle clause dans Spike se construit à partir d'un ensemble d'atomes qui correspondent aux conditions (éventuellement vide dans le cas d'égalité non conditionnelle), et d'un atome conclusion.

En effet, une conjecture, un axiome, un lemme ou une hypothèse sont tous des égalités ou des égalités conditionnelles. La conjecture correspond à un but dans Coq, cependant, le reste se trouve dans l'environnement ou le contexte courant. Dans ce qui suit, nous détaillerons chacun de ces cas.

Algorithm 6.5 Spécification d'une égalité

```
1:  procedure SPECIF_EGALITE(terme)
2:      (liste_assum, terme_conclusion) := decompose_prod(terme)
3:      variables := decompose_prod_assum(terme)
4:      atome_conclusion := specif_atom(variables, terme_conclusion)
5:
6:      si non_conditionel(liste_assum) alors
7:          atomes_conditions := ∅
8:
9:      sinon
10:         atomes_conditions := liste_des_atomes(variables, liste_assum)
11:
12:     fin si
13:     retourner nouvelle_clause(atomes_conditions, atome_conclusion)
14: fin procedure
```

B. D'un environnement Coq vers un environnement à la Spike

Un environnement de preuve dans une procédure à la Spike se construit à partir des éléments suivants :

- des dictionnaires,
- une stratégie de preuve,
- un ensemble d'axiomes,
- un ensemble de lemmes,

– un ensemble de conjectures, et
– un ensemble d'hypothèses de récurrence.

Un seul parcours de l'environnement courant dans Coq suffit pour définir tous ces éléments, à l'exception des conjectures qui correspondent aux buts ouverts.

B.a. Les dictionnaires Les dictionnaires dans Spike sont des structures générales permettant de collecter tous les symboles d'un langage de premier ordre : dictionnaire de variables, dictionnaire de sortes et dictionnaire de fonctions. Afin d'optimiser le temps d'exécution de la procédure, les dictionnaires sont définis en parallèle avec le reste de l'environnement. Au cours de son exécution et de sa rencontre avec un nouveau symbole, la procédure *specif_atom* permettant de spécifier un atome met à jour le dictionnaire.

B.b. Les axiomes Dans un environnement Coq, les axiomes de l'environnement E sont des **définitions** appelées constantes et appartenant à la catégorie *assomption*. Dans notre cas, nous considérons que la forme générale des axiomes s'écrit de la façon suivante :

$a := B$ avec B est un terme d'égalité ou d'égalité conditionnelle et du type B : *Prop*.

L' **Algorithme 6.6** : *specif_axioms*, filtre l'environnement courant E dans Coq et récupérer uniquement les déclarations et les définitions nécessaires telles que les axiomes en mettant à jour les dictionnaires. Les procédures *is_constant*, *is_assomption* et *constant_body* sont des procédures de Coq permettant respectivement de vérifier si un élément de l'environnement est une constante ou non, s'il est une assomption ou non, et de fournir sa définition sous la forme d'un terme. La procédure *is_egalite* vérifie si un terme est un terme d'égalité ou d'égalité conditionnelle.

L'ensemble des axiomes est déduit automatiquement à partir de la définition des symboles de fonctions. Cette définition facilite l'automatisation des preuves des axiomes.

B.c. Les lemmes Ils correspondent aux théorèmes déjà prouvés desquels nous récupérons les propositions en vérifiant que celles-ci sont des termes d'égalité ou d'égalité conditionnelle.

B.d. Les conjectures Elles correspondent aux buts ouverts dans Coq. Du fait que nous traitons les buts un par un, nous ne considérons que le premier but. En effet, un but dans Coq est un contexte Γ et une conclusion telle que nous l'avons définie dans la **Définition 4.12**. Afin de mettre le but sous une forme standard ayant un contexte vide $\Gamma := \emptyset$ et une conclusion sous la forme d'un terme d'égalité ou d'égalité conditionnelle, nous effectuerons des prétraitements qui consistent à appliquer la tactique `revert_all` correspondant à la tacticielle suivante :

`intros` ; `revert_conditions` ; `revert_variables`.

Appliquée sur un but, `revert_all` permet de mettre toutes les variables et les conditions dans le contexte en tant que produit dans la conclusion. Tout d'abord, elle commence par la construction du produit des conditions, et ensuite par la construction du produit des variables. Notons que `revert_conditions`, respectivement `revert_variables`, applique la tactique `revert` tant qu'il y a encore des conditions, respectivement des variables, dans le contexte courant. Enfin,

Algorithm 6.6 Spécification des axiomes

```
1: procedure SPECIF_AXIOMS(E)
2:     axioms := list_of_axioms(E)
3: fin procedure
1: procedure LIST_OF_AXIOMS (E)
2:     si E ≠ ∅ alors
3:         soit e ∈ E
4:         si is_constant(e) et is_assomption(e) alors
5:             body := constant_body(e)
6:             si is_egalite(body) alors
7:                 eq_axiom := specif_egalite(body)
8:                 retourner ({eq_axiom} ∪ (list_of_axioms(E\{e})))
9:             sinon
10:                retourner (list_of_axioms(E\{e})
11:            fin si
12:        sinon
13:            retourner (list_of_axioms(E\{e})
14:        fin si
15:    sinon
16:        retourner (∅)
17:    fin si
18: fin procedure
```

nous pouvons appliquer la procédure *specif_egalite* à la conclusion obtenue pour déterminer la clause Spike correspondante.

Exemple 6.2 *Soit le but suivant :*

$x : nat$
$H1 : odd(x) = true$
==========
$even(S\ (S\ (x))) = false$

L'application de `revert_all` *nous permet d'avoir le but suivant :*

==========
$forall\ x : nat,\ odd(x) = true \rightarrow even(S\ (S\ (x))) = false$

B.e. Les hypothèses de récurrence Dans la récurrence implicite telle qu'elle est implémentée dans Spike, les hypothèses de récurrence sont des clauses de la preuve déjà traitées. Dans notre cas, ces clauses correspondent à des buts déjà simplifiés dont nous enregistrons les identificateurs, i.e. leurs *uid*s des variables existentielles associées. Comme nous l'avons détaillé dans le **Chapitre 4**, les sous-buts dans Coq sont des variables existentielles dont les types sont leurs propositions logiques. Pour fournir à la procédure Spike la liste d'hypothèses de récurrence, il suffit de transformer tous les types de toutes les variables existentielles qui correspondent aux sous-buts de la pré-preuve par récurrence implicite. Bien évidemment, il est vérifié que leurs

types sont des types d'égalité ou d'égalité conditionnelle. Sur le plan pratique, afin d'éviter le recalcul des hypothèses de récurrence à chaque étape de la preuve, leurs clauses sont enregistrées et identifiées par leurs *uid*s.

6.2.2 De Spike vers Coq

L'application de certaines règles d'inférence peut engendrer la traduction d'une ou de plusieurs égalités conditionnelles ou non conditionnelles qui peuvent jouer le rôle d'un nouveau sous but ou d'une hypothèse de récurrence. En effet, Coq possède des procédures qui sont habituellement utilisées par son analyseur syntaxique, et qui permettent d'interpréter des termes construits syntaxiquement. Nous effectuons une construction syntaxique des termes et nous appelons la procédure de Coq, *check*, pour effectuer la vérification de typage de ces termes. Spike représente une égalité conditionnelle ou non conditionnelle en tant que paire de deux listes des atomes :

$$egalite := (premises, conclusions)$$

L'ensemble de *premises* peut être vide et dans ce cas l'égalité est une égalité non-conditionnelle, et *conclusions* peut être vide et dans ce cas l'égalité correspond à l'égalité $premises \Rightarrow False$. Nous supposons que la liste *conclusions* contient au plus un atome. La procédure générale de traduction est présentée dans l' **Algorithme 6.7**. La procédure *creer_produit* fournie par Coq permet la création du produit de l'implication logique et du produit des variables universellement quantifiées. En prenant une liste de définitions et un terme, *creer_produit* retourne le terme produit correspondant.

6.3 Représentation générale des règles d'inférence de Spike

L'**Algorithme 6.8** explique d'une façon générale l'application d'une règle d'inférence quelconque de la stratégie générale Spike dans Coq. Nous proposons cet algorithme d'une façon générique pour qu'il soit adapté pour toutes les règles et pour qu'il soit réutilisé dans le cas des tactiques interactives. Dans ce qui suit, la règle d'inférence, la procédure de son application et sa tactique sont toutes confondues. Comme nous avons expliquer dans les motivations, cette fusion est possible parce que une règle est traduit par la même tacticielle pendant la construction et la validation.

L'application d'une règle commence par mettre à jour tous les éléments de l'environnement à la Spike. La structure générale *env_Spike* contiendra tous les éléments d'un environnement Spike comme c'est détaillé auparavant. La procédure associée à *regle*, la règle d'inférence de Spike, s'exécute sur la clause de but courant comme si elle était traitée dans Spike. Le résultat *solution* contient les nouvelles conjectures qui correspondent aux nouveaux sous-buts. En cas de réussite de la règle d'inférence, la procédure *calcul_tactique* calcule la tacticielle qui permettra de rejouer la règle d'inférence dans Coq. Pour chaque règle, il existe une tacticielle générique comme nous le détaillerons dans la suite.

Algorithm 6.7 Traduction d'une égalité Spike en un terme Coq

```
1: procedure TRADUCTION_EGALITE(egalite_Spike)
2:     (premises, conclusion, variables) := egalite_Spike
3:     si premises == ∅ alors
4:         termes_premises := ∅
5:         si conclusions == ∅ alors
6:             échec
7:         sinon
8:             terme_atome := creer_terme(conclusions)
9:         fin si
10:    sinon
11:        pour tout atom dans premises faire
12:            terme_atome := creer_terme(atom)
13:            termes_premises := ajuter_termes_atomes(terme_atome)
14:        fin pour
15:        si conclusions == ∅ alors
16:            terme_conclusion := False
17:        sinon
18:            terme_conclusion := creer_terme(conclusions)
19:        fin si
20:    fin si
21:    terme := creer_produit(termes_premises, terme_conclusion)
22:    retourner creer_produit(variables, terme)
23: fin procedure
```

Algorithm 6.8 Application d'une règle d'inférence

```
1: procedure APP_REGLE(regle)
2:     env_Spike := mettre_jour_environnement_Spike(env_Coq, but_courant)
3:     si env_Spike est déterminé alors
4:         solution := appliquer_regle_inference(regle, env_Spike)
5:         si solution existe alors
6:             tac := calcul_tactique(regle, solution)
7:             retourner tac
8:         sinon
9:             règle non applicable
10:        fin si
11:    sinon
12:        échec règle
13:    fin si
14: fin procedure
```

6.4 Intégration du raisonnement par récurrence implicite

La récurrence implicite est présente dans certaines règles d'inférence de Spike telles que : Subsumption et Rewriting.

6.4.1 Subsumption

La règle d'inférence Subsumption du système d'inférence de Spike décrit dans la **Figure 2.2** permet de supprimer les redondances dans une preuve. Une conjecture peut être supprimée si elle est subsumée par une autre clause de la preuve ou des lemmes.

Dans Coq, la traduction de Subsumption distingue les deux cas suivants :
- La subsomption par un lemme : elle correspond à une application de la tactique `apply` en utilisant ce lemme ;
- La subsomption par une autre clause : elle consiste à appliquer de la tactique `apply` en utilisant une clause ϕ_{sub} qui correspond à un sous-but dans la preuve. Cette opération ne peut être acceptée par Coq que si ϕ_{sub} est correctement déclarée dans le contexte courant du but courant. Pour ajouter ϕ_{sub} au contexte, nous utilisons la tactique primitive `assert`. Dans un premier temps, pendant la construction de la pré-preuve, le sous-but associé à la preuve ϕ_{sub} est admis, mais il est enregistré en tant qu'hypothèse de récurrence à valider.

Toute hypothèse de récurrence introduite dans la preuve est marquée syntaxiquement en lui associant un nom significatif (préfixé par la chaîne de caractères Subsumption et post-fixé par le numéro du but correspondant). Ce marquage facilite leur capture pendant la validation de la preuve.

Exemple 6.3 *Soient les deux conjectures* $c_1 := \forall x : nat, x + 0 = x$ *et* $c_2 := \forall y : nat, y + 0 = y$. *La clause de* c_2 *est la clause d'un sous-but dans la preuve de* c_1. *Nous supposons que* c_2 *peut être subsumée par* c_1 *avec la substitution* $\theta = \{x \leftarrow y\}$. *On peut appliquer la règle Subsumption avec le script Coq suivant :*
`assert` *(H_Subsumption :* $\forall$ *x : nat , x + 0 = x); [admit;* `apply` *H_Subsumption y].*

La décision qu'une clause peut être subsumée par une autre est réalisée par une procédure Spike qui vérifie l'ordre et détermine la substitution. À ce stade de la preuve, nous n'effectuons pas la validation de la comparaison parce qu'elle se fait automatiquement pendant l'étape de la validation de la preuve.

Dans l'**Exemple 6.3**, l'hypothèse de récurrence est admise, i.e. le but correspondant est résolu par la tactique `admit`. Cette tactique engendre la création d'un axiome qui a le même type que l'hypothèse de récurrence. Dans nos travaux, nous souhaitons éviter la déclaration des tels axiomes pendant l'application de la tactique `subsumption`. Donc, nous créons notre propre terme de preuve qui contient l'hypothèse de récurrence mais qui ne contient pas sa preuve. Ce terme est utilisé par la suite pour raffiner le but courant. Toutefois, il est un terme qui n'est pas bien typé. Donc, naturellement l'utilisation de la tactique `refine` avec ce terme engendre une erreur. Pour cette, raison nous avons utilisé la procédure de raffinement dans Coq de bas niveau appelée `refine_no_check` qui appartient à son code source. Cette procédure force le raffinement du but courant avec ce terme même s'il n'est pas bien typé. Donc, la tactique `subsumption` ne respecte pas les règles de typage de Coq. Néanmoins, la correction de l'approche générale est toujours conservée par l'étape de la validation qui s'effectue à la fin de la preuve.

Exemple 6.4 *Par exemple, le script de l'* ***Exemple 6.3*** *peut être remplacé par*
`refine_no_check` *(*`fun` *H_Subsumption :* $\forall$ *x : nat,* $x + 0 = x \Rightarrow$ *H_Subsumption y).*

La procédure `refine_no_check` permet de raffiner le but courant avec un terme sans vérifier que ce dernier est bien typé.

Définition 6.5 (Subsumption) *Soient deux conjectures $c_1 \in H \cup L$ et $c_2 \in E$ ayant respectivement les deux ensembles des variables $Var(c_1) = \{x_1, \ldots, x_n\}$ et $Var(c_2) = \{y_1, \ldots, y_n\}$. S'il existe une substitution θ telle que $c_2 = \theta c_1$ et $Image(\theta) = \{t_1, \ldots, t_n\}$, alors la traduction de la règle Subsumption :*

- $(E \cup \{c_2\}, H) \rightarrow (E, H), c_1 \in L$ *est :*
  apply *nom_lemme_c_1* ***with*** $(x_1 := t_1), \ldots, (x_n := t_n)$.
  Sachant que nom_lemme_c_1 est le nom de lemme qui correspond à la conjecture c_1 dans l'environnement courant du séquent dans Coq ;
- $(E \cup \{c_2\}, H \cup \{c_1\}) \rightarrow (E, H \cup \{c_1\})$ *est :*
  refine *(****fun*** *H_Subsumption_uid :* $\forall$ $y_1 : S_1$, ... , $y_n : S_n$, $c_1 \Rightarrow$ *H_Subsumption_uid* $t_1, \ldots, t_n$*).*
  Sachant que le uid dans le nom de l'hypothèse de récurrence, H_Subsumption_uid, est l'uid de but qui correspond à la conjecture c_1.

Nous proposons de mettre en oeuvre la tactique `subsumption` en suivant l' **Algorithme 6.9**. Notons que la structure *historique* est utilisée pour enregistrer toutes les informations de la preuve, et est détaillée dans la **Section 6.6**. Ainsi, R est l'ensemble d'axiomes et L est l'ensemble de lemmes.

Algorithm 6.9 Application de la règle Subsumption

```
1: procedure CALCUL_SUBSUMPTION(solution,env_Spike)
2:     (conjecture_sub, substitution_sub) := determiner_conjecture_subsumption(solution)
3:     ∀x_i ∈ Var(conjecture_sub), (x_i, t_i) := Image(substitution_sub)
4:     L := determiner_lemmes(env_Spike)
5:     R := determiner_systeme_reecriture(env_Spike)
6:     si conjecture_sub ∈ L alors
7:         nom := trouver_identifiant(conjecture_sub, L, R)
8:         retourner apply nom with (x_1 := t_1) ... (x_n := t_n)
9:     sinon
10:        uid := determiner_uid_conjecture(conjecture_sub, historique)
11:        terme_preuve := calculer_terme_preuve_subsumption(conjecture_sub, (∀i ∈
   [1..n], (x_i := t_i)))
12:        retourner refine(terme_preuve)
13:    fin si
14: fin procedure
```

6.4.2 Rewriting

La règle d'inférence Rewriting de Spike décrite dans la **Figure 2.2** peut réécrire une conjecture avec des axiomes, des hypothèses de récurrence ou des lemmes.

Dans Coq, la traduction de la Rewriting peut être effectuée par la tactique `rewrite`, cependant, il faut faire la distinction entre deux cas :

- Réécriture en utilisant un axiome ou un lemme : dans ce cas il suffit d'appliquer la tactique `rewrite` en précisant le nom de l'axiome ou du lemme ;
- Réécriture en utilisant une hypothèse de récurrence : elle consiste à réécrire en utilisant une conjecture de la preuve. De même que la subsomption, cette opération n'est pas triviale et elle nécessite la définition de cette hypothèse de récurrence afin qu'elle est utilisée par la suite.

Exemple 6.6 *Soit un état dans une preuve à la Spike décrit par la paire suivante :* $(E, H) := (\{c_2 := y + 0 = y\}, \{c_1 := x + 0 = x\})$. *La règle de réécriture peut être appliquée de la façon suivante*

$$(E, H) \xrightarrow[\text{réécriture avec } c_1]{} (\{c_3 := y = y\}, \{c_1 := x + 0 = x\}) \text{ si } c_1\{x \leftarrow y\} \leq c_1.$$

Cette règle peut être reproduite par le script suivant :

`assert` (H_Rewriting : ∀ x : nat , x + 0 = x) ; [admit/ `rewrite` H_Rewriting].

De même que la tactique `subsumption`, nous forcerons le raffinement en utilisant un terme de preuve non acceptable par Coq. Nous marquons également les cas de la réécriture avec des noms d'hypothèses de récurrence particuliers ($H\_Rewriting\_uid$).

Définition 6.7 (Rewriting) *Soient deux conjectures c_1 et c_2. S'il existe une règle de réécriture $(r := s \to t) \in H \cup L \cup R$ et une substitution θ telle que $c_1[\theta(s)] \to_{H \cup L \cup R} c_2[\theta(t)]$, alors la traduction de la règle **`rewrting`** :*

- $(E \cup \{c_1\}, H) \to (E \cup \{c_2\}, H), r \in R \cup L$ *est :*
  ***`rewrite`** nom_r **`with`** $(x_1 := t_1), \ldots, (x_n := t_n)$.*
  Sachant que nom_r est le nom de la règle de réécriture de r dans l'environnement courant du séquent dans Coq ;
- $(E \cup \{c_1\}, H) \to (E \cup \{c_2\}, H), r \in H$ *est :*
  ***`refine`** (**`fun`** H_Rewriting_uid : ∀ y_1 : S_1 , ... , y_n : S_n , r ⇒ _) ; **`rewrite`** H_Rewriting_uid with* $(x_1 := t_1), \ldots, (x_n := t_n)$.
  Sachant que $Var(c_1) = \{x_1, \ldots, x_n\}$, $Var(r) = \{y_1, \ldots, y_n\}$ et le uid dans le nom de l'hypothèse de récurrence, H_Rewriting_uid, est l'uid de but qui correspond à la règle de réécriture r.

Nous proposons l' **Algorithme 6.10** pour calculer le script équivalent de la réécriture.

6.5 Intégration d'autres raisonnements à la Spike

6.5.1 Generate

Dans Spike, la règle d'inférence Generate telle qu'elle est définie dans la **Figure 2.2**, effectue une analyse par cas sur la conjecture courante c. Cette analyse est basée sur la technique de surréduction. Le résultat est un ensemble de nouvelles conjectures qui sont des instances réduites de cette conjecture c. Chaque cas est calculé par une substitution et un axiome. Dans Coq, une telle analyse par cas peut être effectuée par raffinement du but courant avec un terme pattern matching. L'un des avantages de cette traduction est la vérification de la complétude forte. D'une

Algorithm 6.10 Application de la règle Rewriting

1: **procedure** REWRITING(regle,solution)
2: $(regle_{rew}, substitution_{rew}) := determiner\_conjecture\_reecriture(solution)$
3: $\forall x_i \in Var(conjecture_{rew}), (x_i, t_i) := Image(substitution_{rew})$
4: **si** $regle \in R \cup L$ **alors**
5: $nom := trouver\_identifiant(conjecture_{rew}, L, R)$
6: **retourner** $rewrite\ nom\ with (x_1, t_1) \dots (x_n, t_n)$
7: **sinon** ▷ Ce cas correspond à $regle \in H$
8: $uid := determiner\_uid\_conjecture(conjecture_{sub}, historique)$
9: $terme\_preuve, nom := calculer\_terme\_preuve\_rewrite(conjecture_{rew}, (\forall i \in [1..n], (x_i := t_i)))$
10: **retourner** $refine(terme\_preuve); rewrite\ nom\ with (x_1, t_1) \dots (x_n, t_n).$
11: **fin si**
12: **fin procedure**

part, Coq n'accepte qu'un terme pattern matching que s'il associe à chaque cas un pattern. D'autre par, notre procédure assure pendant la construction que la complétude est respectée à l'aide des procédure de vérification de typage des termes pattern matching de code source de Coq.

La tactique `generate` commence par le calcul de l'ensemble des substitutions couvrantes noté par Sub_c. Ces substitutions sont totales par rapport à l'ensemble des variables de la conjecture, Var_c. En effet, même dans le cas où la procédure de la règle d'inférence Generate fournit une substitution non totale, nous avons une procédures qui complète toute substitution par rapport à l'ensemble des variables d'une clause. L'utilisation des substitutions totales nous a permis de mieux gérer les variables. Cette opération améliore le terme de preuve fournit pendant la construction de la preuve et par conséquent facilite son utilisation pendant la validation, i.e. détails fournis dans la partie validation. À partir des ensembles Sub_c et Var_c, **generate** construit un terme pattern matching qui sera utilisé par la suite pour raffiner le but courant à l'aide de la tactique primitive `refine`. Ce raffinement est suivi par une simplification en utilisant l'ensemble d'axiomes, i.e. chaque instance est réécrite avec l'axiome qui a permis de la calculer.

Exemple 6.8 *Soit la conjecture* $c := \forall x : nat, x + 0 = x$ *et l'ensemble d'axiomes* $Ax := \{(ax_1 := \forall (x : nat), 0 + x = 0), (ax_2 := \forall (x\ y : nat), S(x) + y = S(x + y))\}$. *L'ensemble des substitutions couvrantes de* c *est* $\{\theta_1, \theta_2\}$. *La substitution* $\theta_1 = \{x \leftarrow 0\}$, *respectivement* $\theta_2 = \{x \leftarrow S(y)\}$, *est déterminée en fonction de* ax_1, *respectivement* ax_2. *Les nouvelles instances sont simplifiées par les axiomes* ax_1 *et* ax_2. *Nous obtenons le terme pattern matching suivant* t *:*

```
fun x : nat ⇒
match x with
| 0 → _
| S y → _
end
```

La règle ***generate*** *est reproduite par l'application de :* `refine` t *;* `[rewrite` ax_1`/rewrite` ax_2`]` .

Dans un cas conditionnel, nous pouvons avoir deux instances obtenues par la même substitution σ', mais elles seront simplifiées en utilisant deux axiomes de la forme suivante :

$$(ax_1) : P_1 \wedge c_1 = true \Rightarrow s_1 = t_1$$

$$(ax_2) : P_2 \wedge c_2 = \mathit{false} \Rightarrow s_2 = t_2$$

Ce cas correspond à une réécriture conditionnelle qui peut être effectuée de la même façon que **Total Case Rewriting** que nous exposons dans ce qui suit.

Définition 6.9 (Generate) *Soient c une conjecture avec $Var(c) := \{x_1, \ldots, x_m\}$ étant l'ensemble de ses variables, $Ax := \{ax_1, \ldots, ax_n\}$ l'ensemble d'axiomes et $\Theta := \{\theta_1, \ldots, \theta_n\}$ l'ensemble des substitutions couvrantes. Sachant que $Dom(\theta_1) = \ldots = Dom(\theta_n) = Var(c)$ et que $\forall i \in [1, \ldots, n], Image(\theta_i)$ est l'image de θ_i. La tactique* `generate` *est la traduction de la règle* ***generate*** *suivante : $(E \cup \{c\}, H) \vdash (E \cup \{c_1, \ldots, c_n\}, H \cup \{c\})$. La structure générale de* `generate` *correspond à la tacticielle :*

```
refine(
  fun x_1 : S_1, ..., x_m : S_m =>
    match x_1, ..., x_m with
    | t_1^1, ..., t_m^1 -> c_1
    ⋮
    | t_1^n, ..., t_m^n -> c_n
    end); simplification .
```

Sachant que $\forall i \in [1..n], Image(\theta_i) := \{t_1^i, \ldots, t_m^i\}$, et `simplification` *est une tactique qui permet de simplifier chaque instance avec l'ensemble d'axiomes convenables.*

Nous proposons de mettre en oeuvre la tactique `generate` de la façon décrite dans l' **Algorithme 6.11**.

Algorithm 6.11 Application de la règle Generate

```
1: procedure CALCUL_GENERATE(solution,env_Spike)
2:     nouvelles_conjectures := determiner_nouvelles_conjectures(solution)
3:     conjectures_axiomes := ∅
4:     pour tout  c ∈ nouvelles_conjectures faire
5:         conjecture_axiome := determiner_conjecture_axiome(c, env_Spike)
6:         conjectures_axiomes := conjectures_axiomes ∪ {conjecture_axiome}
7:     fin pour
8:     (terme_preuve, tac_simplification) := calculer_terme_preuve_generate(conjectures_axiomes)
9:     retourner refine(terme_preuve); tac_simplification
10: fin procedure
```

6.5.2 Total Case Rewriting

La règle d'inférence Total Case Rewriting de Spike décrite dans la **Figure 2.2** s'applique à une conjecture qui possède un sous-terme a réductible en utilisant un sous ensemble des règles du système de réécriture induit par les axiomes. La recherche de cet sous ensemble est effectuée par une procédure Spike à laquelle nous fournissons tous les axiomes (le système de réécriture R) ainsi que la conjecture courante. Dans nos travaux nous considérons uniquement des conditions booléennes. Donc, la procédure retourne toujours deux règles de réécritures dont l'une contient une condition de la forme : $c = \mathit{true}$ et l'autre une condition de la forme $c = \mathit{false}$ (*true* et *false* sont les deux constructeurs de la sorte *bool*).

Dans Coq, la traduction de Total Case Rewriting s'effectue en deux étapes :

- Montrer la complétude des conditions : nous ne considérons dans notre cas que des conditions booléennes. Nous commençons par monter la proposition suivante : $c = true \vee c = false$;
- Effectuer la réécriture en utilisant les deux règles de réécriture correspondant aux deux atomes : $c = true$ et $c = false$.

Exemple 6.10 *Nous continuons avec la même spécification décrite dans l'**Introduction**. Soit la conjecture suivante :* $even(S(S(add(x, x)))) = true$. *Le sous-terme* $even(S(S(add(x, x))))$ *est une instance de la partie gauche des deux axiomes suivants :*

$(even_3)$ $\forall x, odd(x) = false \Rightarrow even(S(S(x))) = true$ *et*

$(even_4)$ $\forall x, odd(x) = true \Rightarrow even(S(S(x))) = false$.

Donc, l'application de la règle Total Case Rewriting *peut être appliquée de la façon suivante :*
`assert( H := (odd (add x x) = false) ∨ (odd (add x x) = true)).`
`destruct (odd (add x x)).`
`now right.`
`now left.`
`destruct H;`
`[rewrite even3;[|apply H]|`
`rewrite even4;[|apply H]].`

Nous obtenons les deux sous-buts suivants :

$\forall x, odd(add(x, x)) = false \Rightarrow true = true$

$\forall x, odd(add(x, x)) = true \Rightarrow false = true$

Pour faciliter et généraliser l'automatisation de la réécriture conditionnelle, nous proposons de créer un théorème qui montre la complétude des conditions. La preuve de ce théorème peut être effectuée de façon séparée de la preuve courante dans l'étape de prétraitement.

Exemple 6.11 *Nous reprenons le même exemple précédent. La traduction est la suivante :*

1. *Preuve de complétude des conditions :*

   `Theorem odd_true_false : ( ∀ x, (odd x = false) ∨ (odd x = true)).`
   `intro x.`
   `destruct (odd x).`
   `now right.`
   `now left.`
   `Qed.`

2. *Traduction de* Total Case Rewriting *:*
   `assert(HTCR := odd_true_false (add x x)).`
   `destruct HTCR as [HTCR|HTCR];`
   `[rewrite even3;[|apply HTCR]|`
   `rewrite even4;[|apply HTCR]].`

La procédure générale de la tactique Total Case Rewriting est décrite dans l'**Algorithme 6.12**. À partir des résultats retournés par la procédure de Spike (*solution*), la procédure *pre_traitement* déclare et prouve le théorème de complétude des conditions et retourne son identificateur.

Algorithm 6.12 L'algorithme de la règle Total Case Rewriting

1: **procedure** CALCUL_TOTAL_CASE_REWRITING(*solution*,*env_Spike*)
2: *condition_reecriture* := *pre_traitement*(*solution*, *env_Spike*)
3: *unificateur* := *determiner_unificateur*(*solution*)
4: *tac* := assert (HTCR := *condition_reecriture*(*unificateur*)). destruct HTCR as [HTCR|HTCR] ; [rewrite r_{true} ; [| apply HTCR] | rewrite r_{false} ; [| apply HTCR]].
5: **retourner** *tac*
6: **fin procedure**

6.5.3 Positive/Negative Decomposition

A. Positive Decomposition

La règle d'inférence Positive Decomposition décrite dans le système d'inférence Spike de la **Figure 2.2** simplifie une conjecture qui a une conclusion dont les littéraux construits par le même constructeur et qui est décrite par la forme générale suivante :

$$P \Rightarrow c(x_1, \ldots, x_n) = c(y_1, \ldots, y_n)$$

En se basent sur le fait que les constructeurs (dans notre cas) sont libres, la règle élimine la conjecture courante et la remplace par un ensemble des conjectures. Cet ensemble contient des conjectures qui vérifient l'égalité entre les arguments de constructeur -deux à deux- entre les deux côtés.

Dans Coq nous pouvons traduire cette règle dans deux étapes :
- Définir la décomposition du constructeur : soit un constructeur c. La décomposition positive consiste à prouver la proposition suivante $\forall x_1, \ldots, x_n, y_1 \ldots, y_n, x_1 = y_1 \rightarrow \ldots \rightarrow x_n = y_n \rightarrow c(x_1, \ldots, x_n) = c(y_1, \ldots, y_n)$;
- L'application de théorème d'injection sur le but courant.

Exemple 6.12 *Soit la conjecture suivante* $c1 := Cons(u1, App(Flat(u4), Flat(u5))) = Cons(u1, Flat(Noeud(u4, u5)))$ *sachant que Cons est un constructeur. L'application de la règle Positive Decomposition engendre les deux conjectures suivantes :* $u1 = u1$ *et* $App(Flat(u4), Flat(u5)) = Flat(Noeud(u4, u5))$. *La traduction de l'application de la règle consiste à :*
- *Définir le théorème de décomposition et le prouver :*
  `Theorem` *positif_decomp_th* := $\forall$ $(x_1 : nat)$ $(x_2 : nat)$ $(y_1 : list\ nat)$ $(y_2 : list\ nat)$, $x_1 = x_2 \rightarrow y_1 = y_2 \rightarrow Cons\ x_1\ y_1 = Cons\ x_2\ y_2$.
  `intros` x_1 x_2 y_1 y_2 H_1 H_2.
  `rewrite` H_1.
  `rewrite` H_2.
  `reflexivity`.
  `Qed`.
- *Applique le théorème sur le but courant :*
  `eapply` *positif_decomp_th*.

Algorithm 6.13 L'algorithme de la règle positive decomposition

1: **procedure** CALCUL_POSITIVE_DECOMPOSITION(solution)
2: *constructeur* := *recuperer_racine*(*solution*)
3: *th_positive_decomposition* := *declaration_theoreme_pd*(*constructeur*)
4: **retourner** *eapply identifiant_th_pd*
5: **fin procedure**

La procédure générale de la tactique `positive decomposition` est décrite dans l' **Algorithme 6.13**.

B. Negative Decomposition

La règle d'inférence Negative Decomposition du système d'inférence de Spike décrite dans la **Figure 2.2** simplifie une conjecture par la décomposition d'une condition ayant la forme suivante :

$$P_1 \wedge c(x_1, \ldots, x_n) = c(y_1, \ldots, y_n) \wedge P_2 \Rightarrow e$$

Dans Coq, la tactique `inversion_clear` décrite dans le **Chapitre 4** s'applique à une hypothèse du contexte de la forme : $c(x_1, \ldots, x_n) = c(y_1, \ldots, y_n)$. De ce fait, cette tactique suffit pour traduire Negative Decomposition.

Exemple 6.13 *Soit la conjecture* $c1 := even(S(add(x, x))) = true \Rightarrow true = true \Rightarrow false = true$ *se simplifie à* $c2 := even(S(add(x, x))) = true \Rightarrow false = true$ *en éliminant la condition* $true = true$*. L'application de cette règle peut être traduite par le script suivant :*
`intros H0 H1.`
`inversion_clear H1.`
`revert H0.(* Cette étape est recommandée pour obtenir un but qui rassemble exactement à la conjecture résultat de Spike*)`

La procédure générale de la tactique est décrite par l' **Algorithme 6.14**. La procédure *recuperer_negdecomposition* récupère la condition c à laquelle s'applique la décomposition. La procédure *creer_nom_hypotheses* introduit toutes les conditions en tant qu'hypothèses dans le contexte courant. La procédure *determiner_negcondition* détermine l'identificateur de la condition c dans le contexte courant.

Algorithm 6.14 L'algorithme de la règle negative decomposition

1: **procedure** CALCUL_NEGATIVE_DECOMPOSITION(solution)
2: *condition* := *recuperer_negdecomposition*(*solution*)
3: *hypotheses* := *creer_nom_hypotheses*(*conjecture*)
4: *neghypothese* := *determiner_negcondition*(*condition*, *hypotheses*)
5: **retourner** intros hypotheses. inversion_clear neghypothese. revert_all.
6: **fin procedure**

Récapitulation

Certaines règles de Spike ont des tactiques équivalentes et existantes déjà dans Coq à l'instar **tautology** qui correspond à `auto`. Nous notons que les tactiques qui implémentent la récurrence implicite ne construisent pas un terme de preuve bien-typé. Par conséquent, le terme de preuve construit ne peut jamais être accepté par Coq. La validation de la pré-preuve construite à partir de ces tactiques est nécessaire. Dans le cas où nous avons de grandes preuves, nous pouvons utiliser dans la stratégie à la Spike dans Coq toutes les tactiques automatiques qui permettent de réduire un but.

Nous notons que dans cette approche la complétude forte nous intéresse surtout dans le cas de la tactique `generate`. Et dans ce cas, cette propriété est assurée par le fait que le terme pattern matching construit pour raffiner le but courant est vérifié qu'il couvre tous les cas.

6.6 Validation automatique d'une preuve par récurrence implicite à la Spike

Nous adoptons la même méthode de certification de la tactique `Spike` qui consiste à valider l'application de la récurrence noethérienne à l'ensemble des clauses de la preuve. Cependant, les étapes de validation d'une preuve sont largement améliorées pour cette nouvelle tactique `StrategieSpike`. Cette amélioration apparu surtout dans le fait que une clause est prouvée de la même façon pendant la construction de la preuve et pendant la validation. Ce qui nous a permis d'éviter certains cas d'échec rencontré auparavant. En effet, nous avons créé une arborescence *historique* qui sert comme un historique de la pré-preuve. Cette arborescence contient les numéros des anciens buts auxquels nous associons les numéros de leurs sous-buts. La procédure générale de l'automatisation de la certification des pré-preuves est décrites dans l' **Algorithme 6.15**. Nous détaillerons dans ce qui suit chacune des sous-procédures de cette procédure générale.

Algorithm 6.15 L'algorithme de la validation automatique d'une pré-preuve par récurrence implicite

```
1: procedure VALIDER_PRE_PREUVE(historique)
2:     definition_algebre_terme(env_courant)
3:     validation_theoreme_initial(historique)
4: fin procedure
1: procedure VALIDATION_THEOREME_INITIAL(historique)
2:     LF := declarer_ensemble_buts(historique)
3:     main_th := declarer_th_recurrence(LF, historique, env_courant)
4:     validation_theoreme_initial(main_th)
5: fin procedure
```

6.6.1 Définition automatique de l'algèbre de termes

La définition d'une algèbre de termes est identique à celle de la tactique `Spike`. Toutefois, les procédures permettant d'effectuer cette définition sont différentes. En effet, nous préparons dans `StrategieSpike` l'algèbre de termes en même temps que la définition de l'environnement *env_Spike*, et ce afin d'optimiser le temps d'exécution total de la tactique. Sur le plan pratique,

cette algèbre construit une bibliothèque qui possède le même nom que la preuve courante postfixé de la chaîne "_spec". Si la construction de la pré-preuve est terminée, cette bibliothèque sera importée afin que son algèbre de termes soit utilisée par la suite. Elle contient les mêmes définitions que nous avons évoqué dans la **Sous-section 5.4** (l'ensemble des symboles, la signature, le status, l'index, l'instance du module Term et l'instance du module RPO).

6.6.2 Validation automatique du théorème initial

La tactique `StrategieSpike` essaie de prouver que le théorème initial est une conséquence inductive d'un ensemble des clauses d'une pré-preuve basée sur la récurrence sur les clauses. Le théorème initial correspond à une clause de cet ensemble. Cette proposition se définit de la même façon que la tactique `StrategieSpike`. Les améliorations résident notamment dans la preuve du théorème de l'application de la récurrence noethérienne sur l'ensemble des clauses de la pré-preuve. La forme générale du théorème identique à celle générée par la tactique `Spike` est la suivante :

$\forall\ F,\ In\ F\ LF \rightarrow \forall\ u,\ (\forall\ F',\ In\ F'\ LF \rightarrow \forall\ e,\ less\ (snd\ (F'\ e))\ (snd\ (F\ u)) \rightarrow fst\ (F'\ e)) \rightarrow fst\ (F\ u)$.

Sachant que LF est l'ensemble des fonctionnelles associées à toutes les clauses de la pré-preuve, u et e sont les vecteurs de variables universelles des clauses. La preuve de ce théorème se base sur :

1. Une analyse par cas sur l'ensemble LF. Cette analyse génère n sous-buts avec n le nombre des clauses dans LF ;
2. Chaque clause est résolue de la même façon que le but correspondant dans la pré-preuve. Toutefois, pendant cette étape de validation les hypothèses de récurrence deviennent des instances des clauses de l'ensemble LF. Ceci est établi par l'application de la récurrence noethérienne sur LF.

La preuve d'une clause e, i.e. $fst\ (F\ u)$, peut être résolue par l'une de deux façons suivantes, et ce en fonction de l'ensemble Φ_e des clauses engendrées par l'application d'une tactique à la Spike sur la clause e. Ainsi, on vérifie à chaque fois les hypothèses de récurrence notées par $\Phi_{\leq_e}$, i.e. $(\forall\ F',\ In\ F'\ LF \rightarrow \forall\ e,\ less\ (snd\ (F'\ e))\ (snd\ (F\ u)) \rightarrow fst\ (F'\ e))$. Nous considérons le même ordre *less* qui traduit l'ordre noethérien défini sur les clauses égalitaires ($\leq_c$) :

$\Phi_{\leq_e} = \emptyset$: Ce cas correspond à une clause qui n'a pas d'hypothèses de récurrence. Pour cette raison, elle peut être résolue de la même façon que dans la pré-preuve. Nous récupérons le sous-terme de preuve qui a résolu e dans la pré-preuve et nous le ré-appliquons directement à e en utilisant la tactique `refine`.

$\Phi_{\leq_e} \neq \emptyset$: Ce cas correspond à une clause e qui a des clauses plus petites, i.e ces clauses jouent le rôle d'hypothèses de récurrence. De la même façon que dans le cas précédent, nous récupérons le sous-terme de preuve, t_e, ayant résolu e dans la pré-preuve. Cependant, ce cas correspond à une utilisation d'une hypothèse de récurrence, i.e t_e n'est pas bien typé à cause de la présence de cette hypothèse non prouvée. Cette hypothèse est changée par l'application de la clause correspondante e_H avec $e_H \in LF$. Ce nouveau sous-terme est bien typé si nous pouvons prouver que $e_H \leq_e e$. Cette comparaison s'effectue par *less* et est toujours valide parce que nous ne considérons que des preuves réductibles.

6.7 Résultats et limites

Cette approche est une implémentation de l'algorithme général **Algorithme 3.2** ayant Spike comme ATP et Coq comme ITP. Elle nous a permis d'intégrer le raisonnement par récurrence implicite directement dans Coq. La construction de la preuve et sa validation par le même environnement a apporté une amélioration du processus général de la certification des preuves par récurrence implicite proposé dans le chapitre précédent.

Néanmoins, cette tactique de l'**Algorithme 3.2** possède les limites déjà citées dans le **Chapitre 3** telles que :

- `L'échec de la validation de la preuve` :
  Cet échec se produit dans le cas de la non-validation d'un théorème ou la non-validation de l'application du raisonnement par récurrence implicite sur l'ensemble des clauses de la pré-preuve. Notre tactique peut identifier le cas d'échec de la validation. Par contre, nous n'avons pas un moyen qui peut affirmer que le théorème est non valide dans le cas contraire. Notons que dans cette approche, nous utilisons la même méthode de validation qui est relativement coûteuse en termes de temps d'exécution.
- `L'échec de la construction de la pré-preuve :`
  Ce cas est le plus rencontré parce que la procédure à la Spike construit la preuve en se limitant seulement aux raisonnements réductibles existants déjà dans Spike. Il est rencontré quand il n'y a pas une preuve par récurrence implicite ou le théorème initial est non correct ou la procédure diverge. Ces trois cas correspondent aux trois cas d'échec de Spike.
- `L'échec de la récupération du contexte et de la conjecture courante :`
  Ce cas d'échec est rencontré aussi avec la première approche. Nos deux tactiques sont limitées aux spécifications conditionnelles égalitaires ayant que des conditions booléennes.

Conclusion

Dans ce chapitre, nous avons présenté une approche qui permet de construire des preuves par récurrence implicite à la Spike dans Coq. Elle ne permet pas seulement de certifier des telles preuves mais aussi d'effectuer des raisonnements par récurrence mutuelle et paresseuse dans Coq. Toutefois, cette approche n'utilise que des tactiques qui implémentent des règles d'inférence réductibles.

Chapitre 7

Perspectives : construction interactive des preuves par récurrence implicite à la Spike dans Coq

Sommaire

Notre troisième approche de certification des preuves par récurrence basées sur les clauses égalitaires construites à la Spike dans Coq. Cet environnement à la Spike contient un ensemble des tactiques et des commandes. Les tactiques permettent la construction interactive d'une pré-preuve basée sur la récurrence implicite à la Spike. Et les commandes permettent de valider la pré-preuve construite. Dans ce cas, le certificat est la pré-preuve après sa validation par Coq.

Dans ce chapitre, nous monterons d'abord nos motivations pour cette approche. Ensuite, nous présenterons son schéma général. En plus, nous expliquerons via un exemple le principe général de la construction de la preuve ainsi sa validation. Enfin, nous proposerons les éventuelles améliorations qui peuvent être apportés à cette approche.

7.1 Présentation générale de l'environnement à la Spike

7.1.1 Motivations

Dans le but de généraliser la construction des preuves par récurrence basée sur les clauses égalitaires dans Coq, nous proposons d'éliminer la contrainte de réductibilité respectée par la récurrence implicite dans les deux approches précédentes. Cette relaxation peut être réalisée en construisant la pré-preuve par des tactiques à la Spike ou par d'autres tactiques déjà existantes dans Coq. Par conséquent, l'approche de validation proposée auparavant ne peut plus être convenable à la certification de la pré-preuve. Pour cette raison, il est nécessaire de trouver une nouvelle méthode pour la validation. Cette méthode peut être plus générale parce qu'elle permettra une validation des preuves par récurrence basées sur les clauses égalitaires.

Cette approche ne permet pas uniquement aux utilisateurs Coq de créer leurs preuves à la Spike interactivement, mais elle permettra aussi de construire des preuves non-réductibles par récurrence paresseuses et mutuelles basées sur les clause égalitaires dans Coq.

7.1.2 Schéma général

Notre approche générale décrite dans la **Figure 7.1** consiste à créer des tactiques qui permettent de construire une pré-preuve par récurrence implicite à la Spike, et des commandes qui permettent de valider la preuve. En partant du terme de la pré-preuve construite, la validation s'effectue par le calcul d'un schéma de récurrence général. La preuve sera recommencée en se basant sur ce schéma.

Nous avons créé pour chaque règle d'inférence une tactique se basant sur l'algorithme correspondant présenté dans le chapitre précédent. De ce fait, nous nous concentrerons dans ce chapitre que sur la validation d'hypothèses de récurrence d'une pré-preuve.

7.2 Construction interactive des preuve par récurrence basée sur les clauses égalitaires

L'idée consiste à créer pour chaque règle d'inférence de Spike une tactique correspondante. Cette tactique possède la même procédure générique proposée, respectivement la même procédure OCAML développée, dans l'approche précédente. Donc, au lieu de construire la preuve automatiquement avec une stratégie à la Spike, elle est construite interactivement par l'utilisateur avec des tactiques à la Spike. Les tactiques qui nous intéressons le plus sont les tactique qui implémentent la récurrence sur les clauses égalitaires telles que : `subsumption` et `rewriting`.

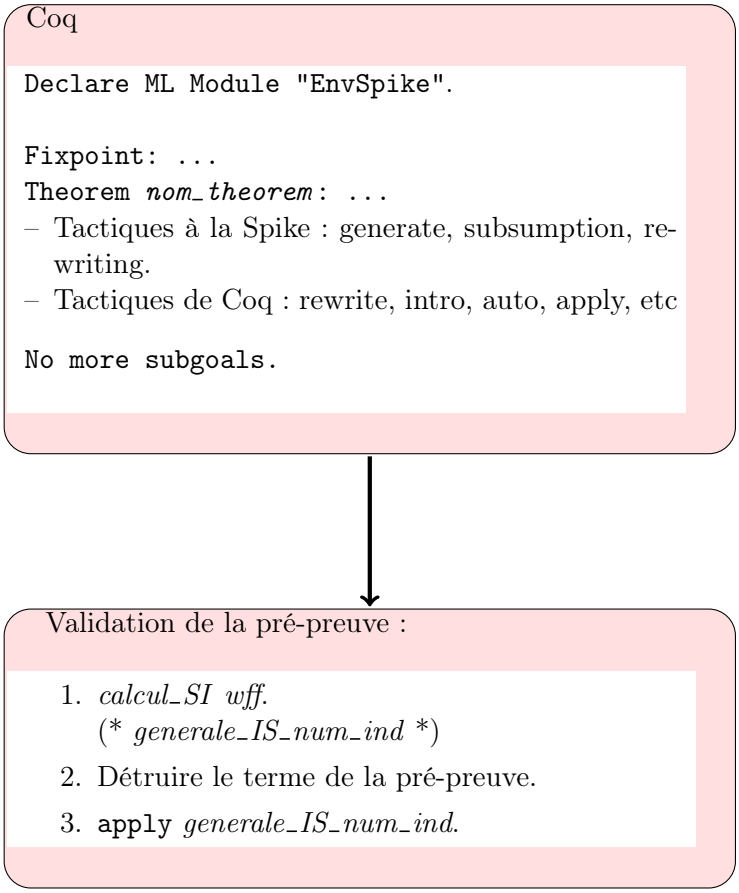

FIGURE 7.1 – Le schéma général de la construction interactive d'une preuve par récurrence basée sur les clauses égalitaires

Exemple 7.1 *Soit c la conjecture* $\forall$ *u1 u2 , u1 + u2 = u2 + u1 qui correspond à la commutativité de l'addition sur les naturels. La preuve par récurrence basée sur les clauses égalitaires à la Spike de c peut être construite interactivement de la façon suivante :*

```
Add LoadPath "../EnvSpike".

Axiom add0 : ∀ n , 0 + n = n.
Axiom addS : ∀ n n1, S(n) + n1 = S(n+n1) .

Theorem addx1 ( x y :nat) : x + S y = S ( x + y).
... (* Preuve de addx1*)
Qed.

Theorem th_add_comm : ∀ u1 u2 , u1 + u2 = u2 + u1.
intros.
generate.
auto.
intros. rewrite addS.
positive decomposition.
generate.
```

```
auto.
intros. apply eq_S.
subsumption.
intros. rewrite add0.
apply eq_S.
generate.
auto.
intros. apply eq_S.
subsumption.
intros. rewrite addS.
apply eq_S.
rewrite addx1.
rewrite addx1.
apply eq_S.
subsumption.
```

Nous pouvons remarquer ainsi que l'utilisation de la récurrence sur les clauses égalitaires apparut en trois positions de la preuve via l'utilisation de la tactique **subsumption**. *Ces trois endroits correspondent à l'utilisation des trois hypothèses de récurrence.*

Le terme de preuve obtenu par la construction de la preuve par récurrence basée sur les clauses égalitaires n'est pas accepté par Coq parce qu'il contient des hypothèses de récurrence qui ne sont pas validées.

Exemple 7.2 *Le terme de pré-preuve obtenu par la pré-preuve de l'**Exemple 7.1** :*

```
(fun u1 u2 : nat ⇒
              (fun u3 u4 : nat ⇒
               match u3 as u5 return (u5 + u4 = u4 + u5) with
               | 0 ⇒
                   match u4 as u5 return (0 + u5 = u5 + 0) with
                   | 0 ⇒ plus_n_O 0
                   | S u5 ⇒
                       (fun u6 : nat ⇒
                        eq_ind_r (fun n : nat ⇒ S u6 = n)
                          ((fun u7 : nat ⇒
                            match u7 as u8 return (S u8 = S (u8 + 0)) with
                            | 0 ⇒ eq_refl
                            | S u8 ⇒
                                (fun u9 : nat ⇒
                                 eq_S (S u9) (S (u9 + 0))
                                   (fun
                                      H_Subsumption801 :
                                       ∀ u10 : nat,
                                        S u10 = S (u10 + 0) ⇒
                                      H_Subsumption801 u9)) u8
                            end) u6) (addS u6 0)) u5
                   end
               | S u5 ⇒
```

```
match u4 as u6 return (S u5 + u6 = u6 + S u5) with
| 0 ⇒
    (fun u6 : nat ⇒
     eq_ind_r (fun n : nat ⇒ S (u6 + 0) = n)
       (eq_S (u6 + 0) u6
          ((fun u7 : nat ⇒
            match u7 as u8 return (u8 + 0 = u8) with
            | 0 ⇒ eq_refl
            | S u8 ⇒
                (fun u9 : nat ⇒
                 eq_S (u9 + 0) u9
                   (fun
                      H_Subsumption832 :
                       ∀ u10 : nat,
                       u10 + 0 = u10 ⇒
                     H_Subsumption832 u9)) u8
            end) u6)) (add0 (S u6))) u5
| S u6 ⇒
    (fun u7 u8 : nat ⇒
     eq_ind_r (fun n : nat ⇒ S (u7 + S u8) = n)
       (eq_S (u7 + S u8) (u8 + S u7)
          (eq_ind_r (fun n : nat ⇒ n = u8 + S u7)
             (eq_ind_r (fun n : nat ⇒ S (u7 + u8) = n)
                (eq_S (u7 + u8) (u8 + u7)
                   (fun
                      H_Subsumption739 :
                       ∀ u9 u10 : nat,
                       u9 + u10 = u10 + u9 ⇒
                     H_Subsumption739 u7 u8))
                (addx1 u8 u7)) (addx1 u7 u8)))
          (addS u8 (S u7))) u5 u6
end
end) u1 u2)
```

Déjà difficile à lire et à suivre le raisonnement effectué, le terme de preuve construit nécessite une validation automatique de ses hypothèses.

7.3 Validation automatique des hypothèses de récurrence

L'idée générale de la validation des hypothèses de récurrence se base sur la reconstruction de la preuve en utilisant un schéma de récurrence général contenant ces hypothèses. Ce schéma est calculé à partir du terme de la pré-preuve. Le calcul s'effectue en analysant syntaxiquement ce terme et en capturant les endroits de l'emploi des hypothèses de récurrence. Enfin, nous pouvons déterminer un schéma de récurrence général que nous appliquerons au but initial. Pour cette raison, le théorème courant aura une preuve par récurrence structurelle dont le schéma est calculé à partir d'une preuve par récurrence basée sur les clauses égalitaires, i.e. calculée d'une façon non-réductible, paresseuse et mutuelle. Dans ce type de récurrence, les hypothèses de récurrence sont toutes des instances de la conjecture du séquent initial. La validation des

hypothèses consiste à montrer que leur emploi est correct, i.e qu'elles sont toujours plus petites que leurs conclusions suivant l'ordre de récurrence. Cet ordre qui se définit localement au niveau du schéma, i.e. il sert à comparer des instances de la même clause.

7.3.1 Principe général

La technique de validation de cette approche consiste à une traduction des pré-preuves par récurrence basée sur les clauses égalitaires construites d'une façon non-réductible, paresseuses et mutuelles en des preuves par récurrence structurelles. L'idée de la traduction est inspirée de [Stratulat2012]. En effet, dans [Stratulat2012] on propose une nouvelle technique de preuve par récurrence qui consiste à la récurrence cyclique. Ce raisonnement permet de combiner les avantages de la récurrence structurelle -non-réductible- et la récurrence implicite -paresseuse et mutuelle-. Son idée consiste à construite d'abord une preuve par récurrence d'une façon non-réductible, paresseuse et mutuelle. Ensuite, on valide les hypothèses de récurrence en capturant des cycles dans la preuve. Ces cycles sont construits par un sous ensemble des clauses de la preuve. L'ordre de récurrence est défini localement au niveau de chaque cycle. Pour une sous-classe de ces preuves le cycle peut être représenté par un schéma de récurrence, i.e. dans le cas ou les hypothèses et les conclusions sont des instances d'une même clause.

En s'inspirant de cette idée générale de traduction des preuves non-réductible, paresseuse et mutuelle, nous proposons que les preuves non-réductibles construites à la Spike seront validées par leur traduction en des preuves par récurrence structurelle dans Coq. Et, nous calculons le schéma de récurrence suivant ces étapes :

1. À partir du terme de pré-preuve nous associons à chaque conjecture simplifiée la substitution cumulative -la composition de toutes les substitution- entre la conjecture initiale et cette conjecture ;
2. Nous calculons pour chaque hypothèse de récurrence la substitution cumulative entre la conjecture initiale et la conjecture de cette hypothèse de récurrence ;
3. Nous associons à chaque conjecture de 1 ses hypothèses de récurrence sous la forme d'une instance de la conjecture initiale en utilisant la substitution déterminée en 2 ;
4. Nous calculons le schéma de récurrence, et par la suite nous le déclarerons et nous prouverons sa correction ;
5. Nous reconstruisons la preuve avec le schéma de 4 et avec les mêmes étapes que la pré-preuve, à l'exception des étapes de récurrence qui deviennent des applications des hypothèses de récurrence introduites par l'application du schéma de récurrence à la conjecture initiale.

La procédure est expliquée dans l'**Algorithme 7.16**. Les détails des procédures de la recherche des hypothèses de récurrence, du calcul de schéma de récurrence à partir d'un terme de preuve, ainsi que la reconstruction de la preuve sont fournis dans la suite.

7.3.2 Recherche des hypothèses de récurrence

Les hypothèses de récurrence sont des instances directes ou simplifiées de l'une des conjectures initiales de la preuve. Pendant la construction de la pré-preuve, nous les marquons toutes par des notations syntaxiques pour faciliter leur collection.

Exemple 7.3 *Soit le terme de preuve cité dans l'**Exemple 7.2**. Les hypothèses de récurrence introduites par la tactique* `subsumption` *sont identifiées par leurs nom qui contiennent*

Algorithm 7.16 L'algorithme de la validation des hypothèses de récurrence

```
1: procedure VALIDER_HI
2:     terme_pre_preuve := calcul_pre_preuve ()   ▷ Cette instruction récupère le terme de la
   pré -preuve avec des procédures appartenant au code source de Coq
3:     (schema_general_recurrence,tac_correction) :=
                calcul_schema_general_recurrence (terme_pre_preuve)
4:     declaration_schema_recurrence(schema_general_recurrence,tac_correction)
5:     script := calcul_nouveau_script (terme_pre_preuve )
6:     reconstruction_preuve (schema_general_recurrence, script)
7: fin procedure
```

la chaîne de caractères **Subsumption** *et les numéros* $801, 832, 739$. *Ces numéros sont les uids ou les identifiants des buts qui correspondent aux conjectures utilisées comme des hypothèses de récurrence. Pour calculer la substitution cumulative, par exemple pour la première hypothèse, on considère le terme de preuve partiel qui mène du but initial jusqu'au but* 801. *Ce calcul peut être effectué par des procédures de réduction de code source de Coq. On élimine la valeur de la variable existentielle* ?801 *du contexte courant* Γ. *Soit* $\Gamma := [?821; ?801; ?743; ?740; ?739; ?727; ?726; ?725; ?725; ?832; ?739; ?724] \cup \Gamma'$ *avec* ?724 *est le but initial, les autres uids correspondent à des sous-buts de la preuve et* Γ' *est le reset de contexte. On laisse à une procédure Coq de calculer le terme partiel de la preuve qui permet de lier le but* 801 *et le but initial* 724. *Le but* 821 *est résolu par subsomption. Donc* 801 *est l'hypothèse de récurrence et* 821 *est la conclusion. Dans la suite, nous donnons les types et les varleurs des varaibles existentielles correspondant aux buts qui permettent de calculer la substituion cumulative.*

?821 : $(Su1 = S(u1 + 0))$:= *(H_Subsumption801 u1)*

?801 : $(Su1 = S(u1 + 0))$:=

```
match u1 with
    | 0 ⇒ ?802
    | S x ⇒ ?803 x
end
```

?743 : $(\forall u3 : nat, 0 + Su3 = Su3 + 0)$:= *(fun u3 : nat ⇒ ?801)*

?740 : $(0 + u2 = u2 + 0)$:=

```
match u2 with
    | 0 ⇒ ?742
    | S x ⇒ ?743 x
end
```

?739 : $(u1 + u2 = u2 + u1)$:=

```
match u1 with
    | 0 ⇒ ?740
    | S x ⇒ ?741 x
end
```

?727 : $(u1 + u2 = u2 + u1)$:= *(?739 u1 u2)*

?726 : $(u1 + u2 = u2 + u1)$:= *?727*

?725 : $(\forall u2 : nat, u1 + u2 = u2 + u1)$: = *(fun u2 : nat ⇒ ?726)*

?724 : $(\forall u1u2 : nat, u1 + u2 = u2 + u1)$:= *(fun u1 : nat ⇒ ?725)*

Notons que nous avons masquer certaines varaibles et certaines étapes pour faciliter la lecture.

Étant réduit par des procédures de réduction qui appartient à Coq, le terme de preuve partiel obtenu est :

```
?724 : (fun u1 u2 : nat ⇒
        match u1 with
                | 0 ⇒ match u2 with
                        | 0 ⇒ ?742
                        | S x ⇒ (fun u3 : nat ⇒ ?801) x
                      end
                | S x ⇒ ?741 x
        end) u1 u2
```

Nous obtenons que la substitution cumulative entre le but initial est la conjecture qui correspond au but 801 *est égale à* $\theta_{801} := \{u1 \to 0, u2 \to S\ x\}$.

La substitution cumulative qui lie la conjecture initiale du but 724 *et la conjecture* 821 *peut être calculée de la même façon et elle est égale à* $\theta_{821} := \{u1 \to 0, u2 \to S(S\ x)\}$.

La procédure générale de recherche des hypothèses de récurrence et de calcul des substitutions cumulatives correspondantes est représentée par l' **Algorithme 7.17**. *calcule_terme_preuve* permet de calculer le terme de preuve partiel entre le but initial et n'importe quel but, sachant que ce calcul correspond à une évaluation de la variable existentielle du but initial dans le contexte courant en éliminant la variable existentielle du but correspondant.

Algorithm 7.17 L'algorithme de recherche d'une hypothèse de récurrence et de sa substitution cumulative

1: **procedure** RECHERCHER_HYP_SUB(uid_initial,uid_hyp,Γ)
2: $\quad\Gamma'' := \Gamma \backslash \{uid\_hyp\}$
3: $\quad terme\_preuve\_partiel := calcule\_terme\_preuve(uid\_initial, \Gamma'')$
4: $\quad\theta_{hyp} := calcul\_substitution(calcule\_terme\_preuve, uid\_hyp)$
5: $\quad$**retourner** (uid_hyp, θ_{hyp})
6: **fin procedure**

7.3.3 Calcul du schéma de récurrence

Le schéma de récurrence sera capturé à partir de terme de la preuve. Les parties concernées sont celles qui correspondent à des instanciations des varaibles, i.e des substitutions cumulatives de la conjecture initiale. Dans le même principe, nous gardons la même structure que le terme de la pré-preuve. Les sous-termes qui peuvent contenir des instanciations dans notre cas sont les termes ayant les formes suivantes : `match ... with`, `let ... in` ou (`fun ... ⇒`) args. La conservation de ces termes remplace le calcul des substitutions cumulatives pour chaque branche de la preuve.

Chaque application d'une hypothèse de récurrence est remplacée par un appel au nom du schéma de récurrence avec les mêmes arguments que l'hypothèse. Ces arguments correspondent aux substitutions d'instanciations des conjectures qui ont donné les hypothèses de récurrence. On précéde cet appel d'un terme de la forme ((`fun ... ⇒ let ... in ...`) args) qui correspond à la substitution cumulative de l'hypothèse de récurrence.

Une opération de lifting est effectuée par les procédures de réduction de Coq. Cette réduction permet d'avoir le schéma sous sa forme finale.

Enfin, le schéma de récurrence se base sur un ordre de récurrence qui permettra de comparer deux instances de la conjecture initiale. Nous utilisons COCCINELLE pour effectuer l'abstraction des termes de Coq. Ainsi, toutes les déclarations nécessaires pour la création d'une algèbre de termes (ces déclarations sont introduites dans le chapitre précédent) sont utilisées dans cette partie. La déclaration de schéma de récurrence s'effectue par `Function` qui correspond à une généralisation d'une déclaration d'une fonction `fixpoint`. L'argument de récurrence est *wff* qui correspond à un ordre bien fondé. Cet argument servira pour prouver la terminaison des appels récursifs de la fonction et peut ne pas être structurellement décroissant.

Exemple 7.4 *Sachant que l'abstraction en une liste des termes COCCINELLE de la conjecture initiale* $u1 + u2 = u2 + u1$ *est la suivante :*
((Term id_Plus ((model_nat (fst x)) : : (model_nat (snd x)) : :nil)) : :

(Term id_Plus ((model_nat (snd x)) : : (model_nat (fst x)) : :nil)) : :nil)

L'ordre de récurrence défini pour le schéma de récurrence de l'exemple 7.1 est :
`Definition` *ord_PlusS (x y : (nat×nat)) :* `Prop` *:=*
less
((Term id_Plus ((model_nat (fst x)) : : (model_nat (snd x)) : :nil)) : :(Term id_Plus ((model_nat (snd x)) : : (model_nat (fst x)) : :nil)) : :nil)

((Term id_Plus ((model_nat (fst y)) : : (model_nat (snd y)) : :nil)) : :(Term id_Plus ((model_nat (snd y)) : : (model_nat (fst y)) : :nil)) : :nil).

`Definition` *wff :=* `fun` *(x y : nat ×nat) ⇒ ord_PlusS x y .*

La définition ord_PlusS est la définition de la comparaison entre deux instances de la clause initiale. Il sera utilisé pour comparer une hypothèse de récurrence et sa conclusion. L'ordre less est le même ordre noethérien sur les clauses égalitaires défini et utilisé dans les deux approches précédentes. L'argument wff est une généralisation de l'ordre qui nous servira dans l'automatisation de notre approche.

Exemple 7.5 *Le schéma de récurrence calculé pour le terme de preuve de*

```
Function generale_IS_724 u1 {wf wff u1} :=
match fst u1 with
| 0 ⇒
    match snd u1 with
    | 0 ⇒ True
    | 1 ⇒ True
    | S (S u8) ⇒ generale_IS_724 (0, S u8)
    end
| S u5 ⇒
    match snd u1 with
    | 0 ⇒ match u5 with
            | 0 ⇒ True
            | S u8 ⇒ generale_IS_724 (S u8, 0)
            end
    | S u6 ⇒ generale_IS_724 (u6, u5)
    end
end.
```

Notons que la commande **`Function`** *n'accepte pas qu'un ordre wf défini par rapport à un seul argument. Pour cette raison, dans cette définition* $u1$ *est une paire qui est composée de deux varaibles* $u1$ *et* $u2$ *de la conjecture initiale. Donc fst u1 (respectivement fst u2) correspond à la variable* $u1$ *(respectivement* $u2$*) de la conjecture c.*

La déclaration du schéma de récurrence engendre une preuve de sa correction. Les obligations de preuve se présentent sous la forme des sous-buts. Dans notre cas, ces obligations correspondent principalement à un ensemble de buts permettant de vérifier que les hypothèses de récurrence dans le cas de récurrence respectent l'ordre *wff* et que cet ordre *wff* est bien fondé.

Exemple 7.6 *La déclaration du schéma generale_IS_724 de l'exemple 7.1 génère les sous-buts suivants : 4* **`subgoals`**, **`subgoal`** *1 (ID 1125)*

```
  ============================
   ∀ u1 : nat × nat,
   fst u1 = 0 →
   ∀ n u8 : nat, n = S u8 → snd u1 = S (S u8) → wff (0, S u8) u1

subgoal 2 (ID 1127) is :
 ∀ (u1 : nat × nat) (u5 : nat),
 snd u1 = 0 →
 ∀ u8 : nat, u5 = S u8 → fst u1 = S (S u8) → wff (S u8, 0) u1
subgoal 3 (ID 1129) is :
 ∀ (u1 : nat × nat) (u5 : nat),
 fst u1 = S u5 → ∀ u6 : nat, snd u1 = S u6 → wff (u6, u5) u1
subgoal 4 (ID 1130) is :
 well_founded wff
```

La définition complète du schéma de récurrence -sa déclaration et la preuve de ses obligations- génère des objets dépendants de ce schéma tels que : le schéma de récurrence.

Exemple 7.7 *La déclaration de schéma de récurrence de l'**Exemple 7.5** engendre les déclarations suivantes :*

```
generale_IS_724_tcc is defined
generale_IS_724_terminate is defined
generale_IS_724_ind is defined
generale_IS_724_rec is defined
generale_IS_724_rect is defined
R_generale_IS_724_correct is defined
R_generale_IS_724_complete is defined
generale_IS_724 is defined
generale_IS_724_equation is defined
```

Donc, le schéma de récurrence final est interprété automatiquement à partir de la fonction récursive. Et il reflète la structure récursive de la fonction même.

Exemple 7.8 *Parmi les objets générés dans l'**Exemple 7.7**, nous nous intéressons au schéma de récurrence generale_IS_724_ind qui sera utilisé pour reconstruire la preuve et pour valider les hypothèses de récurrence.*

7.3.4 Reconstruction de la preuve par récurrence structurelle

Si le schéma de récurrence est déclaré et si ses obligations de preuve sont validées, alors nous effectuons la reconstruction de la preuve de la façon suivante :

1. La destruction de terme de pré-preuve déjà construit ;
2. Le regroupement des variables du but initial dans un vecteur des variables ;
3. La factorisation de but initial sous la forme d'un produit par rapport au vecteur des variables et au schéma de récurrence ;
4. L'application du schéma de récurrence ;
5. La construction de la preuve à partir des mêmes étapes de la pré-preuve par récurrence implicite, à l'exception des étapes de récurrence qui seront remplacées par l'application des hypothèses de récurrence introduites par le schéma de récurrence ;
   (a) notons que pour qu'elles soient réutilisables, les hypothèses de récurrence doivent être développées par rapport au vecteur de variables et simplifiées de la même façon que dans la pré-preuve ;
6. La validation du terme de preuve final par le noyau de Coq.

Exemple 7.9 *La reconstruction de la preuve de la conjecture de l'exemple 7.1 se fait de la façon suivante :*

```
1. Restart.
2. intros.
   change ( let u := (u1,u2) in fst(u) + snd (u) = snd(u) + fst(u)).
3. intro.
   pattern u, (generale_IS_724 u).
4. apply generale_IS_724_ind.
5. (* Cas de base correspond à u1 :=0 *)
   intros e e0.
   rewrite e.
   rewrite e0.
   auto.
   (* Cas de base correspond à u1 :=S 0 *)
   intros e e0.
   rewrite e.
   rewrite e0.
   rewrite addS.
   apply eq_S.
   rewrite add0.
   auto.
   (* Cas de récurrence correspond à u1 :=S S x *)
   intros u0 cas1 u8 cas2 u3 u4 Hind.
   rewrite cas1.
   rewrite cas2.
   rewrite addS.
   rewrite add0.
   apply eq_S.
```

(a) `unfold` *fst, snd, u3, u4* `in` *Hind.*
(`unfold` correspond à la substitution*)*
`rewrite` *add0* `in` *Hind.*

`apply` *Hind.*
*(*Cette étape correspond à l'application de subsomption en utilisant la conjecture* 801**)*
...
(Les mêmes étapes doivent être effectuées pour tous les cas*)*

6. `Qed.`

Nous remarquons que la partie de reconstruction de la preuve est fortement automatisable.

7.4 Résultats et limites

Cette approche est une implémentation de l'algorithme général **Algorithme 3.4**. Elle nous a permis d'intégrer la récurrence non-réductible, paresseuse et mutuelle dans le moteur de preuve de Coq. De plus, elle nous a permis de construire interactivement des preuves par récurrence basée sur les clauses égalitaires dans Coq.

L'inconvénient majeur de cette approche est le décalage de l'étape de la validation jusqu'à la fin de la preuve. Pour l'instant, notre approche choisit les hypothèses de récurrence et effectue la validation à la fin de la preuve telle que la tactique `fix` fait.

Néanmoins, il est plus naturel pour une approche qui implèmente la récurrence basée sur les clauses de laisser l'utilisateur choisir les hypothèses de récurrence. Dans ce cas, il faut créer des outils qui permettent de guider le choix des hypothèses de récurrence à partir de la preuve. Nous pouvons proposer une solution graphique qui se base sur ProofTree [The PROOFTREE development team] en indiquant au cours d'une preuve les conjectures qui peuvent être utilisées comme des hypothèses de récurrence.

Techniquement, différentes étapes de cette approche telles que la définition de l'ordre de récurrence, la preuve de correction de schéma de récurrence et la reconstruction de la preuve en se basant sur le nouveau schéma de récurrence sont effectuées manuellement quoiqu'elles sont fortement automatisables. Donc, il est fortement recommandé de les automatiser.

Enfin, la commande de la validation est limitée à des conjectures ayant seulement deux variables à cause de la définition de l'argument de l'ordre de récurrence lors de la déclaration du schéma de récurrence, comme nous l'avons expliqué dans l'**Exemple 7.5**.

Conclusion

Nous avons présenté dans ce chapitre un environnement à la Spike. Cet environnement contient un ensemble de tactiques et des commandes. Les tactiques sont développées pour créer interactivement des preuves par récurrence à la Spike. Elles permettent de construire des pré-preuves non-réductibles par récurrence paresseuse et mutuelle et basées sur des clauses égalitaires. Les commandes permettent de valider ces pré-preuves en les traduisant en des preuves par récurrence structurelle. Malgré le fait que cette approche est un prototype limité à une sous-classe des termes de preuve, elle peut être généralisée et améliorée pour construire un outil mûr et consistant pour traiter des termes de pré-preuves plus développés.

Conclusions

L'objectif majeur que nous nous sommes fixé avant l'élaboration de ce travail est d'offrir aux utilisateurs de Coq les moyens nécessaires pour effectuer du raisonnement par récurrence implicite. Dans ce but, nous avons développé différentes approches de construction et de certification des preuves par récurrence implicite (comme si ces preuves étaient assistées par Spike) dans l'environnement certifié Coq. La construction de la preuve peut être effectuée de manières différentes :

- Automatique : i) en utilisant Spike comme outil externe à Coq, ou ii) directement dans Coq en construisant le raisonnement par récurrence implicite à la Spike ;
- Interactive en offrant les outils nécessaires aux utilisateurs de Coq.

Pour chacune de ces méthodes, la preuve est vérifiée par le noyau de Coq afin d'obtenir un certificat de sa correction. Nos contributions sont principalement sous la forme d'un ensemble des tactiques Coq développées en OCAML.

La tactique Spike

C'est une tactique automatique qui construit la preuve par récurrence implicite d'un but courant par l'appel à Spike, et qui ensuite la traduit dans un script Coq. Ce script contient un théorème équivalent au théorème initial, ainsi que la traduction de sa preuve par récurrence implicite qui inclut à la fois l'ensemble des formules de cette preuve, la traduction des règles appliquées et l'ordre de récurrence. Une fois ce script validé par le noyau de Coq, le théorème équivalent est utilisé pour résoudre le but courant en une seule étape. Tout ce processus s'effectue de manière transparente par rapport à l'utilisateur. Première expérimentation de la vérification complètement automatique des preuves de Spike, la tactique nous a permis de rejouer dans Coq un sous-ensemble de preuves de validation de la conformité du protocole ABR. Néanmoins, cette approche peut engendrer des échecs dont les causes peuvent être soit la divergence des preuves de Spike, ou soit la mauvaise traduction de certaines étapes de ses preuves.

La tactique à la Spike

Elle implémente en deux étapes une stratégie à la Spike qui s'exécute directement sur le but courant de Coq. Pendant la première étape, elle construit la preuve par récurrence implicite. En lui accordant un but courant, elle décide quelle règle d'inférence sera appliquée en suivant une stratégie de preuve à la Spike. Les nouveaux buts sont engendrés par l'application du script Coq correspondant à la règle appliquée. Si la règle utilise des hypothèses de récurrence, la tactique les introduit en tant qu'hypothèses sans être prouvées dans Coq. Les nouveaux sous-buts seront traités comme le but courant jusqu'au moment où il n'y aura plus de buts à prouver. Pendant la deuxième étape, la pré-preuve obtenue sera validée en utilisant l'approche de la tactique précédente. Cette tactique est également automatique et permet de résoudre certaines de ses

limites. En effet, elle évite définitivement la dépendance de Spike en tant qu'outil externe et elle traduit les règles d'inférence de manière plus fiable. Par contre, la méthode de validation est limitée uniquement aux preuves réductibles. En effet, Cette approche ne peut valider que des preuves par récurrence implicite construites avec des outils à la Spike. Donc, elle ne permet pas de profiter de la variété des tactiques existantes dans Coq.

L'ensemble des tactiques interactives à la Spike

Cette approche consiste à intégrer des règles d'inférence de Spike directement dans le moteur de preuve de Coq en tant que tactiques afin de construire interactivement une pré-preuve par récurrence implicite à la Spike. Contrairement à l'approche précédente de validation d'hypothèses de récurrence, on traduit les pré-preuves en preuves par récurrence explicite. Chaque preuve obtenue se base sur un seul schéma de récurrence calculé à partir du terme de la pré-preuve par récurrence implicite, et qui s'applique à la conjecture à prouver. Cette approche est pourtant limitée à une classe de pré-preuves ayant une certaine structure.

Perspectives

Les tactiques proposées peuvent raisonner uniquement sur des équations. Pour cela, une extension de ces travaux pour le traitement des formules construites avec des prédicats autres que l'égalité paraît importante dans la mesure où elle permettra non seulement d'enrichir l'approche, mais également de profiter du langage expressif de Coq et de ses divers moyens de raisonnement.

Les tactiques automatiques proposées sont basées sur Spike ou sur ses procédures qui peuvent diverger ou échouer. A ce niveau, ces cas sont difficiles à distinguer. Il serait intéressant de créer un moyen pour fournir un contre-exemple en cas d'échec.

L'approche interactive a comme inconvénient majeur le décalage de la validation des hypothèses de récurrence jusqu'à la fin de la preuve. Dans des preuves de grande taille, il sera décevant d'annoncer le refus de l'utilisation de certaines hypothèses après la terminaison de la preuve, notamment si cette dernière est interactivement construite. Par conséquent, il sera intéressant de créer des outils permettant de donner des indications sur les formules qui peuvent jouer le rôle d'hypothèses de récurrence et de guider l'utilisateur dans le choix des hypothèses.

Notre approche générale de certification des preuves par récurrence implicite en utilisant un environnement certifié est réalisée dans cette thèse à l'aide de Coq. Nous pensons que cette approche peut être exploitée par d'autres environnements de preuve tels que celui fourni par ISABELLE. Cette proposition engendra naturellement la représentation d'ordre noethérien dans le nouveau environnement de preuve choisi.

Bibliographie

[Akers1978] S.B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27(6) :509–516, June 1978.

[Al-Karkhi et Woepcke1853] Abou Bekr Mohammed Ben Alhacan Al-Karkhi et Franz Woepcke. *Extrait du Fakhri traite d'algebre (manuscript 952, supplement arabe de la Bibliotheque imperiale) par Abou Bekr Mohammed Ben Alhacan Alkarkhi.* Á l'Imprimerie imperiale, 1853.

[Alvarado et Nguyen2000] Cuihtlauac Alvarado et Quang-Huy Nguyen. ELAN for equational reasoning in Coq. Dans *Proceedings of 2nd Workshop on Logical Frameworks and Meta-languages. Institut National de Recherche en Informatique et en Automatique, ISBN*, pages 2–7261, 2000.

[Armando *et al.*2002] Alessandro Armando, Michaël Rusinowitch et Sorin Stratulat. Incorporating Decision Procedures in Implicit Induction. *Journal of Symbolic Computation*, 34(4) :241–258, 2002.

[Asperti2012] Andrea Asperti. Proof, message and certificate. Dans *Proceedings of 11th International Conference, Intelligent Computer Mathematics - AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012*, rédacteurs Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel et Volker Sorge, volume 7362 de *Lecture Notes in Computer Science*, pages 17–31. Springer, 2012.

[Baader et Nipkow1998] Franz Baader et Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[Balaa et Bertot2000] Antonia Balaa et Yves Bertot. Fix-point equations for well-founded recursion in type theory. Dans *Theorem Proving in Higher Order Logics*, pages 1–16. Springer, 2000.

[Barrett *et al.*2009] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia et Cesare Tinelli. Satisfiability Modulo Theories. Dans *Handbook of Satisfiability*, rédacteurs Armin Biere, Marijn Heule, Hans van Maaren et Toby Walsh, volume 185 de *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[Barthe *et al.*2006] Gilles Barthe, Julien Forest, David Pichardie et Vlad Rusu. Defining and reasoning about recursive functions : A practical tool for the coq proof assistant. Dans *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*, rédacteurs Masami Hagiya et Philip Wadler, volume 3945 de *Lecture Notes in Computer Science*, pages 114–129. Springer, 2006.

[Barthe et Stratulat2003] Gilles Barthe et Sorin Stratulat. Validation of the JavaCard Platform with Implicit Induction Techniques. Dans *Proceedings of 14th International Conference,*

Rewriting Techniques and Applications, RTA 2003, Valencia, Spain, June 9-11, 2003, rédacteur Robert Nieuwenhuis, volume 2706 de *Lecture Notes in Computer Science*, pages 337–351. Springer, 2003.

[Barwise *et al.*2000] Jon Barwise, John Etchemendy, Gerard Allwein, Dave Barker-Plummer et Albert Liu. *Language, proof and logic*. CSLI publications, 2000.

[Bertot et Castéran2004] Yves Bertot et Pierre Castéran. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Springer, 2004.

[Bezem *et al.*2002] Marc Bezem, Dimitri Hendriks et Hans de Nivelle. Automated Proof Construction in Type Theory Using Resolution. *J. Autom. Reasoning*, 29(3-4) :253–275, 2002.

[Bliksem development team] The Bliksem development team. *Bliksem*. www.ii.uni.wroc.pl/~nivelle/software/bliksem/.

[Böhme et Nipkow2010] Sascha Böhme et Tobias Nipkow. Sledgehammer : Judgement Day. Dans *Proceedings of 5th International Joint Conference, Automated Reasoning, IJCAR 2010, Edinburgh, UK, July 16-19, 2010*, rédacteurs Jürgen Giesl et Reiner Hähnle, volume 6173 de *Lecture Notes in Computer Science*, pages 107–121. Springer, 2010.

[Borovanský *et al.*1998] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau et Christophe Ringeissen. An overview of ELAN. *Electr. Notes Theor. Comput. Sci.*, 15 :55–70, 1998.

[Bouhoula *et al.*1995] Adel Bouhoula, Emmanuel Kounalis et Michaël Rusinowitch. Automated Mathematical Induction. *J. Log. Comput.*, 5(5) :631–668, 1995.

[Bouhoula1994] Adel Bouhoula. *Preuves automatiques par récurrence dans les théories conditionnelles*. Thèse de doctorat, Université Henri Poincaré-Nancy I, 1994.

[Bove *et al.*2009] Ana Bove, Peter Dybjer et Ulf Norell. A Brief Overview of Agda - A Functional Language with Dependent Types. Dans *Proceedings of 22nd International Conference, Theorem Proving in Higher Order Logics, TPHOLs 2009, Munich, Germany, August 17-20, 2009*, rédacteurs Stefan Berghofer, Tobias Nipkow, Christian Urban et Makarius Wenzel, volume 5674 de *Lecture Notes in Computer Science*, pages 73–78. Springer, 2009.

[Boyer et Moore1988a] Robert Stephen Boyer et J Strother Moore. *A Computational Logic Handbook*. Academic Press Professional, Inc., San Diego, CA, USA, 1988.

[Boyer et Moore1988b] Robert Stephen Boyer et J Strother Moore. Integrating decision procedures into heuristic theorem provers : a case study of linear arithmetic. Dans *Machine intelligence 11*, pages 83–124. Oxford University Press, Inc., 1988.

[Braibant2012] Thomas Braibant. *Emancipate yourself from LTac : Your first Coq plugin*. http://gallium.inria.fr/blog/your-first-coq-plugin/, 2012.

[Brassinne1853] Èmile Brassinne. *Précis des oeuvres mathématiques de Pierre Fermat et de l'arithmétique de Diophante*. Douladoure, 1853.

[Burstall1969] Rodney Martineau Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1) :41–48, 1969.

[Buss1998a] Samuel R Buss. *Handbook of proof theory*. Elsevier, 1998.

[Buss1998b] Samuel R Buss. An introduction to proof theory. *Handbook of proof theory*, 137 :1–78, 1998.

[Chlipala2011] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011. http://adam.chlipala.net/cpdt/.

[CiME Development Team] The CiME Development Team. *CiME*. http://cime.lri.fr.

[Clarke *et al.*2001] Edmund M. Clarke, Orna Grumberg et Doron Peled. *Model checking*. MIT Press, 2001.

[Contejean *et al.*2007] Évelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons et Xavier Urbain. Certification of Automated Termination Proofs. Dans *Proceedings of 6th International Symposium on Frontiers of Combining Systems, FroCoS 2007, Liverpool, UK, September 10-12, 2007*, rédacteurs Boris Konev et Frank Wolter, volume 4720 de *Lecture Notes in Computer Science*, pages 148–162. Springer, 2007.

[Contejean2005] Évelyne Contejean. *Coccinelle*. https://www.lri.fr/~contejea/Coccinelle/coccinelle.html, 2005.

[Contejean2014] Évelyne Contejean. *Facettes de la preuve*. Habilitation à diriger des recherches, Université Paris-Sud, 2014.

[Coq development team] The Coq development team. *Coq*. http://coq.inria.fr/refman/.

[Coquand et Huet1988] Thierry Coquand et Gérard Pierre Huet. The Calculus of Constructions. *Inf. Comput.*, 76(2/3) :95–120, 1988.

[Corbineau2003] Pierre Corbineau. First-Order Reasoning in the Calculus of Inductive Constructions. Dans *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, rédacteurs Stefano Berardi, Mario Coppo et Ferruccio Damiani, volume 3085 de *Lecture Notes in Computer Science*, pages 162–177. Springer, 2003.

[Courant1996] Judicaël Courant. Proof reconstruction. Rapport technique, Research Report RR96-26, LIP, 1996. Preliminary version, 1996.

[Delahaye2000] David Delahaye. A Tactic Language for the System Coq. Dans *Proceedings of 7th International Conference, Logic for Programming and Automated Reasoning, LPAR 2000, Reunion Island, France, November 11-12, 2000*, rédacteurs Michel Parigot et Andrei Voronkov, volume 1955 de *Lecture Notes in Computer Science*, pages 85–95. Springer, 2000.

[Dershowitz1982] Nachum Dershowitz. Orderings for Term-Rewriting Systems. *Theor. Comput. Sci.*, 17 :279–301, 1982.

[Dyckhoff1992] Roy Dyckhoff. Contraction-Free Sequent Calculi for Intuitionistic Logic. *J. Symb. Log.*, 57(3) :795–807, 1992.

[Gentzen1935] Gerhard Gentzen. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift*, 39 :176–210, 1935.

[Gordon *et al.*1979] Michael Gordon, Robin Milner et Christopher Wadsworth. Edinburgh LCF : a mechanized logic of computation, volume 78 of Lecture Notes in Computer Science, 1979.

[Gordon2000] Michael J. C. Gordon. Reachability Programming in HOL98 Using BDDs. Dans *Proceedings of 13th International Conference, Theorem Proving in Higher Order Logics, TPHOLs 2000, Portland, Oregon, USA, August 14-18, 2000*, rédacteurs Mark Aagaard et John Harrison, volume 1869 de *Lecture Notes in Computer Science*, pages 179–196. Springer, 2000.

[Gramlich2005] Bernhard Gramlich. Strategic Issues, Problems and Challenges in Inductive Theorem Proving. *Electr. Notes Theor. Comput. Sci.*, 125(2) :5–43, 2005.

[Hardy1929] Godfrey Harold Hardy. Mathematical proof. *Mind*, pages 1–25, 1929.

[Henaien et Stratulat2013] Amira Henaien et Sorin Stratulat. Performing implicit induction reasoning with certifying proof environments. Dans *Proceedings of 4th International Symposium on Symbolic Computation in Software Science, SCSS 2012, Gammarth, Tunisia, 15-17 December 2012*, rédacteurs Adel Bouhoula, Tetsuo Ida et Fairouz Kamareddine, volume 122 de *EPTCS*, pages 97–108, 2013.

[Hickey et Nogin2000] Jason Hickey et Aleksey Nogin. Fast Tactic-Based Theorem Proving. Dans *Proceedings of 13th International Conference, Theorem Proving in Higher Order Logics, TPHOLs 2000, Portland, Oregon, USA, August 14-18, 2000*, rédacteurs Mark Aagaard et John Harrison, volume 1869 de *Lecture Notes in Computer Science*, pages 252–267. Springer, 2000.

[Howard1980] William Alvin Howard. The formulas-as-types notion of construction. Dans *To H. B. Curry : Essays on Combinatory Logic, Lambda Calculus, and Formalism*, rédacteurs J. P. Seldin et J. R. Hindley, pages 479–490. Academic Press, 1980.

[Hurd1999] Joe Hurd. Integrating Gandalf and HOL. Dans *Proceedings 12th International Conference, Theorem Proving in Higher Order Logics, TPHOLs'99, Nice, France, September, 1999*, rédacteurs Yves Bertot, Gilles Dowek, André Hirschowitz, C. Paulin et Laurent Théry, volume 1690 de *Lecture Notes in Computer Science*, pages 311–322. Springer, 1999.

[Hurd2003] Joe Hurd. First-order proof tactics in higher-order logic theorem provers. *Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports*, pages 56–68, 2003.

[Hutter et Sengler1996] Dieter Hutter et Claus Sengler. INKA : The Next Generation. Dans *Automated Deduction - CADE-13,Proceedings of 13th International Conference on Automated Deduction, New Brunswick, NJ, USA, July 30 - August 3, 1996*, rédacteurs Michael A. McRobbie et John K. Slaney, volume 1104 de *Lecture Notes in Computer Science*, pages 288–292. Springer, 1996.

[Kaliszyk2005] Cezary Kaliszyk. Validation des preuves par récurrence implicite avec des outils basés sur le calcul des constructions inductives. *Master's thesis, Université Paul Verlaine-Metz*, 2005.

[Kanso et Setzer2010] Karim Kanso et Anton Setzer. Integrating automated and interactive theorem proving in type theory. Dans *AVOCS*, volume 2, 2010.

[Kanso2012] Karim Kanso. *Agda as a platform for the development of verified railway interlocking systems.* Thèse de doctorat, Swansea University, 2012.

[Kerber2010] Manfred Kerber. Proofs, Proofs, Proofs, and Proofs. Dans *Proceedings of 10th International Conference, Intelligent Computer Mathematics, AISC 2010, 17th Symposium, Calculemus 2010, and 9th International Conference, MKM 2010, Paris, France, July 5-10, 2010*, rédacteurs Serge Autexier, Jacques Calmet, David Delahaye, Patrick D. F. Ion, Laurence Rideau, Renaud Rioboo et Alan P. Sexton, volume 6167 de *Lecture Notes in Computer Science*, pages 345–354. Springer, 2010.

[Klein *et al.*2010] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch et Simon Winwood. seL4 : Formal Verification of an Operating-System Kernel. *Commun. ACM*, 53(6) :107–115, 2010.

[Knuth et Bendix1983] Donald E Knuth et Peter B Bendix. Simple word problems in universal algebras. Dans *Automation of Reasoning*, pages 342–376. Springer, 1983.

[Kounalis et Rusinowitch1990] Emmanuel Kounalis et Michaël Rusinowitch. Mechanizing inductive reasoning. *Bulletin of the EATCS*, 41 :216–226, 1990.

[Kreitz2002] Christoph Kreitz. The Nuprl Proof Development System, Version 5 : Reference Manual and User's Guide. *Department of Computer Science, Cornell University*, 2002.

[Leroy *et al.*2014] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy et Jérôme Vouillon. *The OCaml system release 4.02*, 2014.

[McCarthy et Painter1967] John McCarthy et James Painter. Correctness of a compiler for arithmetic expressions. *Mathematical aspects of computer science*, 1, 1967.

[McCarthy1959] John McCarthy. A basis for a mathematical theory of computation. *Studies in Logic and the Foundations of Mathematics*, 26 :33–70, 1959.

[Meng *et al.*2006] Jia Meng, Claire Quigley et Lawrence C. Paulson. Automation for interactive proof : First prototype. *Inf. Comput.*, 204(10) :1575–1596, 2006.

[Musser1980] David R. Musser. On Proving Inductive Properties of Abstract Data Types. Dans *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980*, rédacteurs Paul W. Abrahams, Richard J. Lipton et Stephen R. Bourne, pages 154–162. ACM Press, 1980.

[Owre *et al.*1993] Sam Owre, Natarajan Shankar et JM Rushby. User Guide for the PVS Specification and Verification System (Beta Release). *Computer Science Laboratory, SRI International, Menlo Park, CA*, 1993.

[Pascal et Lafuma1963] Blaise Pascal et Louis Lafuma. *Pascal : oeuvres complètes*. Macmillan, 1963.

[Paulin-Mohring1993] Christine Paulin-Mohring. Inductive Definitions in the system Coq - Rules and Properties. Dans *Proceedings of International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993*, rédacteurs Marc Bezem et Jan Friso Groote, volume 664 de *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993.

[Peano1908] Giuseppe Peano. *Formulaire de mathématiques*, volume 5. Bocca frères, Ch. Clausen, 1908.

[Poincaré1898] Henri Poincaré. La science et l'hypothèse. *Science*, 6 :1–13, 1898.

[Protzen1994] Martin Protzen. Lazy Generation of Induction Hypotheses. Dans *Proceedings of 12th International Conference on Automated Deduction, Automated Deduction - CADE-12, Nancy, France, June 26 - July 1, 1994*, rédacteur Alan Bundy, volume 814 de *Lecture Notes in Computer Science*, pages 42–56. Springer, 1994.

[Rusinowitch *et al.*2000] Michaël Rusinowitch, Sorin Stratulat et Francis Klay. Mechanical Verification of an Ideal Incremental *ABR* Conformance. Dans *Proceedings of 12th International Conference, Computer Aided Verification, CAV 2000, Chicago, IL, USA, July 15-19, 2000*, rédacteurs E. Allen Emerson et A. Prasad Sistla, volume 1855 de *Lecture Notes in Computer Science*, pages 344–357. Springer, 2000.

[Rusinowitch *et al.*2003] Michaël Rusinowitch, Sorin Stratulat et Francis Klay. Mechanical verification of an ideal incremental ABR conformance algorithm. *J. Autom. Reasoning*, 30(2) :53–177, 2003.

[Saint-Hilaire1843] Jules Barthélemy Saint-Hilaire. *Logique d'Aristote*, volume 4. Ladrange, 1843.

[Schmitt *et al.*2001] Stephan Schmitt, Lori Lorigo, Christoph Kreitz et Aleksey Nogin. JProver : Integrating Connection-Based Theorem Proving into Interactive Proof Assistants. Dans *Proceedings of 1st International Joint Conference, Automated Reasoning, IJCAR 2001, Siena,*

Italy, June 18-23, 2001, rédacteurs Rajeev Goré, Alexander Leitsch et Tobias Nipkow, volume 2083 de *Lecture Notes in Computer Science*, pages 421–426. Springer, 2001.

[Spike Development Team] The Spike Development Team. *Spike theorem prover*. http://code.google.com/p/spike-prover/.

[Stratulat et Demange2011] Sorin Stratulat et Vincent Demange. Automated Certification of Implicit Induction Proofs. Dans *Proceedings of 1st International Conference, Certified Programs and Proofs, CPP 2011, Kenting, Taiwan, December 7-9, 2011*, rédacteurs Jean-Pierre Jouannaud et Zhong Shao, volume 7086 de *Lecture Notes in Computer Science*, pages 37–53. Springer, 2011.

[Stratulat2000] Sorin Stratulat. *Preuves par récurrence avec ensembles couvrants contextuels. Applications á la vérification de logiciels de télécommunications*. Thèse de doctorat, Université Henri Poincaré Nancy 1, 2000.

[Stratulat2001] Sorin Stratulat. A General Framework to Build Contextual Cover Set Induction Provers. *J. Symb. Comput.*, 32(4) :403–445, 2001.

[Stratulat2008] Sorin Stratulat. Combining Rewriting with Noetherian Induction to Reason on Non-orientable Equalities. Dans *Proceedings of 19th International Conference, Rewriting Techniques and Applications, RTA 2008, Hagenberg, Austria, July 15-17, 2008*, rédacteur Andrei Voronkov, volume 5117 de *Lecture Notes in Computer Science*, pages 351–365. Springer, 2008.

[Stratulat2010] Sorin Stratulat. Integrating Implicit Induction Proofs into Certified Proof Environments. Dans *Proceedings of 8th International Conference, Integrated Formal Methods, IFM 2010, Nancy, France, October 11-14, 2010*, rédacteurs Dominique Méry et Stephan Merz, volume 6396 de *Lecture Notes in Computer Science*, pages 320–335. Springer, 2010.

[Stratulat2012] Sorin Stratulat. A Unified View of Induction Reasoning for First-Order Logic. Dans *Turing-100 - The Alan Turing Centenary, Manchester, UK, June 22-25, 2012*, rédacteur Andrei Voronkov, volume 10 de *EPiC Series*, pages 326–352. EasyChair, 2012.

[Sutcliffe2000] Geoff Sutcliffe. The CADE-16 ATP System Competition. *J. Autom. Reasoning*, 24(3) :371–396, 2000.

[Szabo1969] Manfred Egon Szabo. *The collected papers of Gerhard Gentzen*, volume 160. North-Holland Amsterdam, 1969.

[Tammet1997] Tanel Tammet. Gandalf. *J. Autom. Reasoning*, 18(2) :199–204, 1997.

[The HOL development team] The HOL development team. *The HOL System TUTORIAL*. http://hol.sourceforge.net/.

[The ISABELLE development team] The ISABELLE development team. *ISABELLE*. http://isabelle.in.tum.de/.

[The PROOFTREE development team] The PROOFTREE development team. *PROOFTREE*. http://askra.de/software/prooftree/.

[The PVS development team] The PVS development team. *PVS*. http://pvs.csl.sri.com/.

[Voicu et Li2009] Razvan Voicu et Mengran Li. Descente infinie proofs in Coq. Dans *Proceedings of the 1st Coq Workshop, Technische Universität München*, 2009.

[Wirth2004] Claus-Peter Wirth. Descente Infinie + Deduction. *Logic Journal of the IGPL*, 12(1) :1–96, 2004.

Index