UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Méthodes pour le raisonnement d'ordre supérieur dans SMT

THÈSE

présentée et soutenue publiquement le 11 février 2021

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Daniel El Ouraoui

Composition du jury

Rapporteurs :	Micaela MAYERO
	Yakoub SALHI
Examinateurs :	David DÉHARBE
	Catherine DUBOIS
	Chantal KELLER
Encadrants :	Jasmin BLANCHETTE
	Pascal FONTAINE
	Stephan MERZ

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thesul.

Remerciements

Et voilà, l'aventure touche à sa fin ! Ce document signe l'accomplissement d'un long travail durant lequel de nombreuses personnes ont été impliquées, et je dois avouer que la liste est longue. L'investissement, pour cette thèse ne se limite pas aux trois années passées en tant que doctorant au LORIA, mais a débuté bien plus tôt. C'est pourquoi je considère que toutes ces personnes doivent figurer dans ce document pour premièrement adresser toute ma gratitude à tous ceux qui ont contribué directement ou indirectement à l'aboutissement de ce travail, mais aussi parce que ce document persistera dans le futur, et que ma reconnaissance sera inscrite ici pour ces personnes aussi longtemps que ce document existera.

Cette aventure a débuté en 2012, lorsque j'ai décidé d'arrêter mon activité professionnelle à l'hôpital Henri Mondor, pour me consacrer à ma passion : l'informatique. Poussé par une curiosité, et l'envie de comprendre de manière profonde les fondements de l'informatique, j'ai dès le début manifesté une forte envie pour de longues études et dès le début je le savais, je voulais faire une thèse !

Ainsi je dois remercier mon premier soutien, ceux qui m'ont accueilli, soutenus financièrement et moralement dans cette aventure : mes Parents qui m'ont assuré un soutien sans faille, malgré l'envergure de mon projet. Annoncer que l'on souhaite tout arrêter après 7 ans de travail pour revenir à la maison, et reprendre ses études pour 8 années, ça peut faire peur. Je remercie aussi mon petit frère pour sa gentillesse, sa disponibilité et son soutien, pendant ces années d'études partagées à la maison.

Sans le soutien et la compréhension de mon ancien employeur, mais aussi de ma hiérarchie (les ressources humaines et la direction des services techniques de l'hôpital Henri Mondor) il ne m'aurait pas été possible de suivre ma première année de licence informatique à l'université Paris Diderot. Je remercie donc Alain Cailliaux et Pascal Debizet, pour m'avoir soutenu et permis d'organiser mon temps de travail afin que je puisse suivre mes cours à la fac.

Je remercie tous les professeurs de la licence informatique et du master MPRI de l'université Paris Diderot, qui m'ont enseigné l'informatique, et qui m'ont permis d'avoir le niveau de connaissance que je détiens maintenant. En particulier je remercie Arnaud Sangnier, pour sa confiance, et son soutien. Un soutien, qui m'a permis d'intégrer le parcours très compétitif qu'est le MPRI. Je remercie aussi Sophie Laplante pour avoir soutenu ma candidature au MPRI. Je dois aussi adresser toute ma reconnaissance à Micaela Mayero, et Damiano Mazza, pour m'avoir donné ma chance en stage et l'opportunité de travailler sur mon premier vrai sujet de recherche, et m'avoir donné le goût des méthodes formelles. C'est quelque part grâce à ce stage au LIPN, que j'ai pu réaliser tous les enjeux et l'importance que pourrait avoir une automatisation efficace de la logique d'ordre supérieur dans les assistants de preuve. Je remercie Rose Goulancourt-Bouaziz pour son soutien et son aide, durant mes études. Je remercie également Bernard Druesne, ses ateliers de communication m'ont été d'une grande utilité pendant ma thèse et m'ont fait évoluer positivement.

J'adresse maintenant toute ma gratitude, à mes encadrants Jasmin Blanchette, Pascal Fontaine et Stephan Merz, pour m'avoir accueilli en stage de Master 2, puis en thèse dans l'équipe VeriDis. Je leur adresse un grand merci, car sans leur aide, leur soutien, leur disponibilité et leur confiance ce document n'aurait jamais vu le jour. Je remercie Jasmin Blanchette, pour m'avoir intégré au sein du projet Matryoshka, et m'avoir accordé ma bourse de thèse. Merci aussi à Jasmin pour nos nombreuses discussions scientifiques, la disponibilité, les conseils et l'esprit critique. Merci aussi à Pascal pour sa franchise et pour m'avoir toujours dit ce qu'il pensait, même si ce devait être négatif : c'est grâce à cela que j'ai pu réaliser que j'avais de réelles difficultés de communication. Sans cette remise en question ce manuscrit ne serait pas ce qu'il est aujourd'hui. Merci aussi à Stephan pour sa gentillesse, son accueil et sa disponibilité, c'est vraiment un plaisir d'avoir un chef d'équipe, et encadrant si agréable au quotidien. Je remercie Haniel Barbosa, pour nos nombreuses discussions, pour m'avoir intégré dans la collaboration avec l'équipe de cvc4, et m'avoir répondu à mes nombreuses questions sur le code de veriT. Je remercie toutes les personnes avec qui j'ai pu collaborer dans ce projet d'extension des solveurs SMT à l'ordre supérieur Clark Barrett, Andrew Reynolds, et Cesare Tinelli. Merci à Cezary Kaliszyk, pour cette collaboration, merci pour sa gentillesse, son temps, et pour m'avoir fait découvrir le domaine de l'intelligence artificielle. Merci aussi à Sophie Tourret, pour notre collaboration pour l'extension de CCFV à l'ordre supérieur, son temps et son aide. Je remercie l'entreprise CLEARSY, et toutes les personnes que j'ai pu rencontrer pendant mon stage. Merci de m'avoir offert cette opportunité de découvrir le monde de l'industrie et m'avoir accueilli dans ses locaux. En particulier je remercie David Déharbe pour m'avoir offert cette opportunité, et pour m'avoir aidé tout au long de ce stage. Je remercie aussi Fernando Mejia, pour nos discussions et pour sa disponibilité. Je remercie aussi Étienne Prun pour sa gentillesse et nos conversations.

Merci à toute l'équipe VeriDis, tous ceux avec qui j'ai pu discuter, m'aérer l'esprit, ou prendre un verre. Merci à Ahmed Bhayat, Alexander Bentkamp, Simon Cruanes, Mathias Fleury, Martin Riener, Hans-Jörg Schurr, Petar Vukmirović pour nos discussions scientifiques, et aussi pour nos moments de détentes. Merci à tous mes collègues de bureau, toutes les personnes qui m'ont suivi scientifiquement au cours de cette thèse, et tous les membres de l'équipe pour ces moments de détentes partagés : Étienne André, Guillaume Bonfante, Martin Bromberger, Horatiu Cirstea, Sylvain Contassot, Antoine Defourné, Marie Duflot-Kremer, Yann Duplouy, Margaux Duroeulx, Sophie Drouot, Nazim Fates, Alexis Grall, Igor Konnov, Sergueï Lenglet, Pierre Lermusiaux, Dominique Mery, Hamid Rahkooy, Nicolas Schnepf, Sorin Stratulat, Thomas Sturm. Merci aussi à toutes les personnes qui ont pris le temps de me faire des remarques, constructives et qui m'ont permis d'améliorer ce document, et ma présentation.

Je tiens à remercier les membres de mon jury de thèse d'avoir accepté d'examiner cette thèse. Je remercie tout particulièrement Micaela Mayero et Yakoub Salhi d'avoir accepté de rapporter mon travail. Je les remercie également pour leurs rapports détaillés et constructifs.

Je remercie enfin Nathalie, ma partenaire de vie, pour sa présence, son soutien au quotidien, sa franchise et pour être là tant dans les bons moments que dans les mauvais. Merci à Merdy pour être toujours présent, et à l'écoute, et enfin merci à tous mes amis pour les moments de

détente et de rigolade.

Table des matières

Table des figures

1

Introduction

L'informatique fait aujourd'hui partie intégrante de notre vie quotidienne. Smartphones, ordinateurs, voitures, que ce soit dans notre vie personnelle ou professionnelle nous sommes tous les jours au contact de logiciels. Nos transactions, informations personnelles, déplacements sont en partie ou intégralement gérés par des logiciels. Que ce soit les trains, avions ou voitures, tous disposent d'une assistance logicielle permettant d'automatiser en partie ou complètement des tâches. Néanmoins, nous ne sommes pas à l'abri de bugs, d'erreurs logicielles introduites lors du développement. Ces erreurs peuvent se loger partout, et être plus ou moins critiques. Par exemple, une erreur qui se produit dans le système d'exploitation d'un ordinateur de bureau n'a généralement pas de conséquence dramatique. Par contre, si une erreur se produit lors de l'exécution d'un programme de guidage d'une fusée, des vies humaines peuvent être en jeu. En effet, de lourds investissements peuvent peser sur ces programmes, et parfois des vies sont en jeu, à la merci de la moindre erreur. Les conséquences de telles erreurs peuvent donc être dramatiques, crash d'avion, accident de la route causé par une voiture autonome, ordres d'achats involontaires sur les marchés boursiers, erreur de dosage sur des patients, etc. La vérification d'absence d'erreur sur des programmes comportant des risques financiers ou humains est donc une nécessité.

Pour anticiper des comportements défectueux, des batteries de tests peuvent être définies lors de la spécification et de la conception du logiciel. Ces tests permettent d'évaluer la robustesse de chaque partie du logiciel tout au long de son développement. Relativement simple à mettre en œuvre, cette approche n'est malheureusement pas exhaustive. Pour couvrir l'ensemble des comportements possibles d'un programme, il est nécessaire de faire appel à des méthodes exhaustives. Plus coûteuses, en temps, et souvent difficiles à mettre en œuvre, ces méthodes sont communément regroupées sous la dénomination de méthodes formelles.

Les méthodes formelles s'appuient sur des outils mathématiques permettant de formaliser totalement le comportement de programmes. Ces approches font appel à des structures mathématiques bien connues, et autour desquelles de nombreux outils ont été développés. Certains de

ces outils permettent notamment de résoudre toute une catégorie de problèmes définis dans ces structures. Une classe d'outils de plus en plus utilisés pour la vérification de programmes sont les assistants de preuve. Ces outils permettent de vérifier certaines propriétés de façon interactive, c'est-à-dire étape par étape. Les preuves sont automatiquement validées par l'outil, ce dernier assurant à l'utilisateur la validité de sa preuve. Malheureusement les formalismes employés par ces outils demandent un très grand niveau de détail par l'utilisateur, et donc impliquent un temps de travail plus long que sur le papier. Une preuve formelle peut prendre plusieurs années, et mobiliser plusieurs personnes. Des exemples bien connus de preuves formelles de grande ampleur sont : le compilateur C certifié CompCert [75], le système d'exploitation entièrement vérifié seL4 [71], ou encore la preuve de la conjecture de Kepler prouvée dans dans HOL Light et Isabelle [60]. Pour pallier la difficulté de créer des preuves avec un grand niveau de détail, des outils de déduction automatique ont été développés.

Les solveurs SMT (satisfaisabilité modulo théorie) font partie d'une catégorie d'outils de déduction automatique permettant de traiter de manière efficace des théories dans lesquelles sont exprimées des symboles de fonctions (ou de prédicats). Des exemples de théories sont : la théorie de l'égalité, de l'arithmétique linéaire, ou de certaines structures de données telles que les datatypes, les tableaux ou les vecteurs de bits ; ces théories peuvent être combinées sous certaines conditions. À la différence d'autres outils traitant les théories de façon axiomatique, les solveurs SMT s'appuient sur une collection de procédures de décision dédiées pour chaque théorie, et d'un SAT solveur pour gérer la structure logique des formules (voir figure 1.1). Couplés à un assistant de preuve les solveurs SMT peuvent se montrer particulièrement utiles. L'assistant de preuve Isabelle [89] emploie un hammer, appelé Sledgehammer [27], pour prouver un maximum de sous-buts automatiquement. Ces hammers font appel à une collection d'outils de déduction automatique, dont notamment de nombreux solveurs SMT tels que cvc4 [44], z3 [42], Yices [49], veriT [31], ou encore des hybrides tels que Vampire [73].

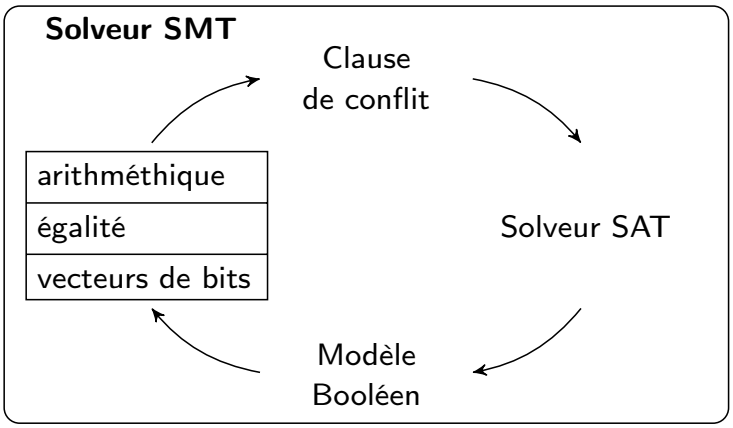

FIGURE 1.1 – Architecture d'un solveur SMT.

L'utilité des solveurs SMT dans les assistants de preuve est en partie due aussi au support pour les quantificateurs. Initialement les solveurs SMT sont conçus pour raisonner sur des formules sans quantificateur. Simplify, tel que présenté en 2003 dans l'article [45] est certainement l'un des plus proches ancêtres des solveurs SMT modernes, qui permet la gestion des quantifica-

teurs sur la base d'une architecture très similaire à ce que l'on connaît maintenant. Aujourd'hui il existe deux grandes familles d'outils de déduction automatique pour raisonner sur la logique de premier ordre : d'une part les solveurs SMT et d'autre part les solveurs basés sur des système de dérivations (superposition [4], tableaux et résolution [103], connexion [22]). Les deux familles se complètent. L'une est plus efficace pour le raisonnement avec théories, par exemple les solveurs SMT sont meilleurs pour les problèmes avec de l'arithmétique ou d'autres théories, alors que la seconde est plus appropriée pour des problèmes avec beaucoup de quantificateurs et du raisonnement équationnel.

1.1 Problématique

Dans le but de contribuer à la démocratisation massive des outils de preuve interactifs, il est indispensable que les outils automatiques soient plus performants, et en particulier les solveurs SMT. Généralement les assistants de preuve reposent sur un formalisme plus expressif que celui accepté par les solveurs SMT : des variantes de la logique d'ordre supérieur. Pour déléguer la résolution de certaines contraintes, les assistants de preuve reposent sur des traductions, permettant d'exprimer les problèmes de la logique d'ordre supérieur dans un formalisme plus adapté aux démonstrateurs automatiques, qui est la logique du premier ordre. Malheureusement, cette approche n'est pas optimale. En effet ces traductions augmentent considérablement la taille du problème original et introduisent une forme de bruit qui nuit à la résolution du problème, et par conséquent limite l'efficacité des solveurs.

Ce travail de thèse s'inscrit dans le cadre du projet ERC Matryoshka de J. Blanchette, un projet qui vise à concevoir des prouveurs automatiques utiles pour la vérification interactive, et à réduire l'écart entre les prouveurs interactifs et solveurs automatiques. L'un des objectifs concrets du projet est d'étendre les capacités de raisonnement des solveurs SMT vers l'ordre supérieur.

1.2 Défis liés à la logique d'ordre supérieur

L'objectif premier de cette thèse vise à proposer une nouvelle architecture pour les solveurs SMT, pour la logique d'ordre supérieur. La conception de cette architecture s'articule en deux étapes. La première consiste à repenser la partie dite ground du solveur (ce qui correspond aux formules sans quantificateur), et en particulier, la procédure de décision pour la théorie de l'égalité (QF_UF). La deuxième étape consiste à introduire une approche adaptée aux problèmes provenant des assistants de preuves pour le raisonnement d'ordre supérieur avec quantificateurs.

Pour raisonner avec des formules de la logique d'ordre supérieur, les solveurs SMT doivent être capables de manipuler des expressions contenant des applications partielles de fonctions, et des variables dites fonctionnelles (c'est-à-dire qui peuvent être instanciées par des fonctions). À

l'ordre supérieur il est commun de représenter les applications de fonctions dans un style dit curryfié c'est-à-dire que toute fonction est unaire ; par exemple $f(a, g(b, c))$ est compris comme $((f\,a)\,((g\,b)\,c))$. En second lieu, les logiques d'ordre supérieur permettent de raisonner avec des formules où les variables quantifiées peuvent potentiellement représenter des fonctions ou des prédicats. Un exemple canonique est le principe d'induction

$$\forall P\,(P(0) \Rightarrow (\forall k\ .\ P(k) \Rightarrow P(k+1))) \Rightarrow \forall x\,P(x)$$

où P est une variable de prédicat quelconque.

Par conséquent pour qu'un solveur SMT puisse manipuler de telles expressions, deux principaux modules doivent être étendus : la théorie de l'égalité et le module d'instanciation. Le premier objectif de cette thèse vise donc à implémenter un algorithme pour la théorie de l'égalité dans le solveur SMT veriT qui puisse gérer des symboles de types d'ordre supérieur. Originairement, la procédure de décision pour l'égalité, appelée fermeture de congruence, a été développée par Nelson et Oppen [84] en 1980 et se basait essentiellement sur la structure de données Union-Find. Peu de temps après Downey et al [47] proposent une version améliorée de cet algorithme avec une complexité $\mathcal{O}(n \log n)$ contre $\mathcal{O}(n^2)$ pour la première version. Bien que cette procédure de décision soit une excellente base pour la théorie de l'égalité, il lui manque néanmoins plusieurs fonctionnalités pour convenir telle quelle dans un SMT solveur. En effet, le fonctionnement des solveurs SMT est fortement dynamique, leur architecture s'inspire directement de celle des solveurs SAT. Le solveur veriT implémente donc une variante de l'algorithme de fermeture de congruence, proposé en 2007 par Nieuwenhuis et al. [85] permettant une adaptation parfaite au calcul CDCL($\mathcal{T}$) tout en préservant la complexité $\mathcal{O}(n \log n)$ de l'algorithme de Downey et al.

Bien qu'étant très performante, cette implémentation est assez difficile à modifier car l'ensemble des composants sont fortement interdépendants. Ainsi l'idée proposée pour étendre cette procédure de décision à l'ordre supérieur, est d'implémenter une version "flexible" de l'algorithme de fermeture de congruence non plus basée sur la structure Union-Find mais sur une structure de graphe qui suit plus naturellement le problème d'entrée. Cette approche permet notamment d'implémenter de façon plus évidente plusieurs extensions basées directement ou indirectement sur la fermeture de congruence, comme par exemple la procédure de décision pour les datatypes [14,96].

D'une autre part la gestion des quantificateurs nécessite aussi d'être étendue, et pour cela nous proposons dans cette thèse une extension des approches employées dans les solveurs SMT qui permettent en particulier de gérer les variables fonctionnelles.

La première partie de cette thèse propose donc une architecture permettant aux solveurs SMT d'être utilisés sans gros effort de traduction par les assistants de preuve et ce grâce aux extensions décrites juste avant.

1.3 Défis liés au raisonnement avec quantificateurs

Aujourd'hui quelques solveurs SMT (z3, cvc4, veriT) acceptent des formules de la logique du premier ordre avec quantificateurs. Pour être résolu, le problème donné en entrée du solveur est découpé de façon à séparer les formules contenant des quantificateurs des formules dites "ground" (c'est-à-dire sans quantificateur). Les formules "ground" peuvent être efficacement traitées par le solveur SMT grâce à des procédures de décision efficaces. Les formules avec quantificateurs doivent être instanciées, c'est-à-dire que les variables liées aux quantificateurs doivent être remplacées par des termes appropriés. La formule obtenue par ce procédé est appelée une instance. Cependant cette approche n'est pas optimale. Et l'une des causes principales d'inefficacité des solveurs SMT est intrinsèquement liée au nombre important d'instances générées par ce procédé. Les techniques utilisées pour instancier les formules dans les solveurs SMT peuvent donc être améliorées en réduisant le nombre d'instances utilisées à celles réellement utiles pour résoudre le problème. Généralement quand un solveur SMT déclare un problème non satisfaisable, il est capable de produire une preuve. Cette preuve peut ensuite être réduite en éliminant toutes les branches inutiles. Les instances apparaissant dans cette version réduite de la preuve sont les instances dites utiles. L'objectif de ce travail est d'améliorer les performances du solveur SMT veriT via un algorithme d'apprentissage basé sur des arbres de décision. L'algorithme est préalablement entraîné sur plusieurs exemples, puis ensuite utilisé comme assistant au sein du solveur, lui permettant ainsi de choisir les instances de manière éduquée sur de nouveaux problèmes.

1.4 Objectif et contributions de cette thèse

L'objectif de cette thèse dans sa globalité est d'offrir des solutions pour améliorer les interactions entre solveur automatique et assistant de preuve. En particulier nous répondons à deux problématiques importantes permettant d'améliorer les usages des solveurs SMT au sein des assistants de preuve. Notre première contribution permet de réduire l'écart entre solveur et assistant de preuve en proposant une architecture adaptée pour la logique d'ordre supérieur. La seconde contribution permet d'améliorer les capacités de raisonnement des solveurs SMT pour les quantificateurs. Pour les deux approches développées nous apportons un ensemble d'évaluations sur des problèmes extraits pour la grande majorité de tâches de formalisation. Les résultats obtenus lors de ces évaluations sont encourageants et montrent que les techniques développées dans cette thèse peuvent apporter de bonnes améliorations pour les solveurs SMT.

1.5 Méthode et plan

Notre travail dans le cadre de cette thèse consiste à élaborer des techniques de raisonnement permettant de manipuler des expressions de la logique d'ordre supérieur (chapitre 3) mais aussi d'améliorer les méthodes de raisonnement pour les formules avec quantificateurs (section 5) dans

le solveur SMT veriT. Nous débutons cette thèse par décrire le fonctionnement des solveurs SMT dans le chapitre 2. Notre première contribution, vise donc à étendre le solveur SMT veriT à la logique d'ordre supérieur. Cette extension se fait en deux étapes : premièrement, définir et implémenter une technique de décision pour les formules sans quantificateur (section 3.6) ; deuxièmement étendre les techniques actuelles d'instanciation à l'ordre supérieur (section 3.7). Le chapitre se conclut par une série d'évaluations de notre prototype en section 3.8. Dans le chapitre 4 nous décrivons l'implémentation du ground SMT solveur implémenté dans veriT pour traiter des formules de la logique d'ordre supérieur sans quantificateur. Dans le chapitre 5, nous proposons une approche hybride pour améliorer la gestion des quantificateurs dans le solveur SMT veriT. En particulier nous proposons une approche par filtrage s'appuyant sur des méthodes d'apprentissage automatique. Nous définissons un encodage pour le problème de l'instanciation vers un algorithme d'apprentissage (section 5.3.3), puis une méthode de sélection (section 5.3.6) permettant d'utiliser les prédictions d'un modèle pour éliminer les instances jugées inutiles du flot d'instances dérivees à partir de stratégies existantes dans le solveur veriT. Nous concluons cette étude par une série d'évaluations sur des problèmes concrets (section 5.4). Enfin nous concluons cette thèse et apportons les différentes directions futures dans le chapitre 7.

2

Le problème SMT

Sommaire

Dans ce chapitre nous présentons les briques fondamentales qui constituent un solveur SMT. Dans la première section nous abordons le principe de fonctionnement d'un solveur SAT, et les techniques développées pour résoudre des problèmes de la logique propositionnelle. Dans la deuxième section nous étudions l'architecture des solveurs SMT. Dans la troisième section nous allons plus loin et étudions comment résoudre des problèmes avec quantificateurs, et enfin nous terminons ce chapitre avec les challenges d'aujourd'hui et un aperçu des contributions de cette thèse.

2.1 Satisfaisabilité booléenne

La satisfaisabilité booléenne est un problème fondamental, au cœur de l'informatique théorique. Appelé SAT, ce problème consiste à déterminer si une formule de la logique propositionnelle admet une solution, c'est-à-dire qu'il existe une interprétation, pour chacune des variables de la formule, telle que la formule est vraie. Dans cette section nous déterminons formellement la syn-

taxe de la logique propositionnelle, et sa sémantique, puis nous parlons des aspects théoriques, et notamment de l'intérêt de ce problème d'un point de vue algorithmique. Nous rapellerons comment il est possible de résoudre ce problème, et quels sont les algorithmes qui permettent d'y arriver.

2.1.1 Syntaxe et sémantique

La logique propositionnelle se compose d'une collection d'opérateurs logiques : la négation $\neg$, la conjonction $\wedge$, et la disjonction $\vee$. Notons qu'il est possible de réduire sans perdre d'expressivité ; l'ensemble des opérateurs logiques à $\{\neg, \vee\}$, $\{\neg, \wedge\}$, etc. La logique propositionnelle se compose de deux constantes $\top$ et $\bot$, et d'un ensemble infini de variables propositionnelles $\mathcal{V}$. Ainsi $\top$ est une formule (resp $\bot$). Si x est une variable, x est une formule (appelé aussi *atome*). Si φ est une formule $\neg\varphi$ est une formule. Les variables et leurs négations sont appelés *littéral*. Si φ_1 et φ_2 sont des formules $\varphi_1 \vee \varphi_2$ (resp $\varphi_1 \wedge \varphi_2$). L'implication logique se note $\varphi_1 \Rightarrow \varphi_2$, et est équivalente à $\neg\varphi_1 \vee \varphi_2$. L'équivalence $\varphi_1 \Leftrightarrow \varphi_2$ est équivalente à $\varphi_1 \Rightarrow \varphi_2 \wedge \varphi_2 \Rightarrow \varphi_1$. On note qu'un modèle satisfait une formule $\mathcal{M} \models \varphi$, si l'interprétation de φ dans $\mathcal{M}$ est vraie. Cela est noté $[\![\varphi]\!]^{\mathcal{M}} = \top$ (voir [51] pour plus de détails). Une formule est dite *satisfaisable* si et seulement si elle admet une interprétation $\mathcal{M}$ telle que $\mathcal{M} \models \varphi$. Quand $\mathcal{M} \models \varphi$ on dit par abus de langage que $\mathcal{M}$ est un modèle. À contrario si φ n'admet aucun modèle, alors φ est dit non satisfaisable. On notera $\mathcal{M} \models \neg\varphi$, et $\mathcal{M} \not\models \varphi$ lorsque φ ne satisfait pas $\mathcal{M}$.

Exemple 2.1.1. Soient x, y, z des variables propositionnelles, la formule $((x \Rightarrow y) \vee (y \Rightarrow z)) \Rightarrow z$ est satisfaisable. En effet il suffit de prendre le modèle où z est interprété à $\top$, les valeurs de x et y n'ont pas d'importance, car $\varphi \Rightarrow z$ est vraie pour toute formule φ. •

2.1.2 Complexité algorithmique

La théorie de la complexité est une discipline de l'informatique théorique qui permet d'évaluer l'efficacité en temps, ou en espace des algorithmes. L'efficacité en temps est mesurée par un nombre d'étapes atomiques, nécessaires à l'exécution d'un algorithme sur l'ensemble de ses entrées. L'efficacité d'un algorithme est mesurée dans le meilleur des cas (l'entrée pour laquelle l'algorithme se comporte le mieux), dans le pire des cas (l'entrée pour laquelle l'algorithme se comporte le moins bien), et en moyenne.

Tout problème, décidable, peut être résolu par un algorithme en nombre fini d'étapes, et cet algorithme admet une complexité algorithmique. Cette complexité est formellement évaluée par une fonction de temps, prenant en argument la taille de l'entrée, et renvoyant en sortie le nombre d'étapes nécessaires à l'algorithme pour s'exécuter sur cette entrée. Lorsque cette fonction s'exprime comme un polynôme, on dit que la classe de complexité pour cet algorithme est polynomiale, et est notée P. Lorsqu'un algorithme est décidable, et que l'on peut vérifier en un temps polynomial si une solution convient, on dit que la classe de complexité pour cet

algorithme est NP. On remarque donc que tout problème dans P est aussi dans NP, cependant l'inverse reste encore indéterminé (P = NP ?).

Lorsque qu'il est possible de réduire tout problème A (par exemple SAT) de NP à un autre problème B, via un algorithme (réduction) polynomial, alors le problème B est appelé NP-difficile, car A est au moins plus facile que B. Si un problème est à la fois dans NP, et est NP-difficile alors ce problème est appelé NP-complet [93].

SAT est un problème NP-complet bien connu [35]. Par ailleurs SAT est le problème NP-difficile canonique. Il est donc suffisant de construire une réduction en temps polynomial à partir de SAT pour démontrer qu'un autre problème est NP-difficile. Par conséquent résoudre SAT efficacement revient à résoudre n'importe quel autre problème NP-difficiles efficacement. C'est une des raisons pour lesquelles le problème SAT est un problème important.

Bien qu'en théorie le problème SAT se trouve être un problème difficile, en pratique, et grâce aux évolutions technologiques de ces dernières décennies, de nombreux algorithmes, sophistiqués sont capables de traiter de grosses instances de SAT dans des temps raisonnables. C'est d'ailleurs ce résultat, en apparente contradiction avec la théorie, qui place aujourd'hui le problème SAT au cœur des enjeux de l'informatique moderne.

2.1.3 Traitement des formules

Le format de formules habituellement supporté par les solveurs SAT est la forme normale conjonctive CNF (Conjunctive Normal Form). Ce format permet de représenter un problème comme une conjonction de disjonctions, appelées *clauses*. Nous verrons dans la section suivante que les algorithmes développés pour les solveurs SAT s'appuient fortement sur cette présentation. En effet pour déterminer la valeur de vérité d'une clause il suffit qu'au moins un littéral booléen soit vrai. Par conséquent, pour qu'une formule en CNF soit non satisfaisable il suffit qu'au moins une clause soit fausse (c.-à-d. qu'aucun littéral dans cette clause n'est vrai). L'algorithme de mise CNF naïf est exponentiel en le nombre de clauses. L'opérateur de disjonction et de conjonction sont distributif, par conséquent il suffit d'appliquer le principe de distributivité des opérateurs de disjonction jusqu'à obtenir une CNF. Par exemple pour transformer $a \vee (b \wedge c)$, où a, b et c sont des atomes en CNF on distribue $\vee$ dans la formule $(b \wedge c)$ ce qui nous permet d'obtenir la CNF suivante $(a \vee b) \wedge (a \vee c)$. De plus cette transformation préserve l'équivalence des formules et par conséquent ne modifie pas la satisfaisabilité de la formule d'origine. L'algorithme de Tseitin [117], est un algorithme efficace pour transformer les formules en CNF. Néanmoins l'algorithme de Tseitin préserve uniquement l'équisatifiabilité des formules. L'algorithme applique sur toutes les sous-formules de la formule d'origine un certain nombre de transformations. Chaque opérateur logique, est appelé porte ou combinateur, et est associé à une formule équivalente. Chacune de ces formules est une définition qui introduit une nouvelle variable, de sorte que la sous-expression puisse être substituée à cette formule dans toutes les autres sous-expressions où elle apparaît. L'algorithme procède inductivement par traiter la plus petite sous-expression de la formules, et s'arrête lorsque

la sous-expression est la formule elle-même. Par exemple pour traduire la formule $a \vee (b \wedge c)$ en CNF, on considère l'ensemble des sous-expressions $\{(b \wedge c), a \vee (b \wedge c)\}$, on remarque que les sous-expressions b et c sont ignorées étant atomiques. On prend la première sous-expression $b \wedge c$ à laquelle on applique la porte $\wedge$, ce qui nous permet d'obtenir l'équivalence $x_1 \leftrightarrow (b \wedge c)$. La forme développée de l'expression $x_1 \leftrightarrow (b \wedge c)$ est l'expression en CNF : $(\neg b \vee \neg c \vee x_1) \wedge (b \vee \neg x_1) \wedge (c \vee \neg x_1)$. Dans la seconde sous-expression $a \vee (b \wedge c)$ on peut remplacer $(b \wedge c)$ par x_1, ce qui donne $a \vee x_1$. On applique maintenant la porte $\vee$ à l'expression $a \vee x_1$ ce qui nous permet d'obtenir l'expression $x_2 \leftrightarrow (a \vee x_1)$. Une fois développée l'expression $x_2 \leftrightarrow (a \vee x_1)$ correspond à l'expression en CNF : $(a \vee x_1 \vee \neg x_2) \wedge (\neg b \vee x_2) \wedge (\neg x_1 \vee x_2)$. Finalement en exprimant la conjonction de toutes les sous-expressions crées on obtient la formule $x_2 \wedge (x_1 \leftrightarrow (b \wedge c)) \wedge (x_2 \leftrightarrow (a \vee (b \wedge c)))$. Si l'on remplace les expressions $x_1 \leftrightarrow (b \wedge c)$ et $x_2 \leftrightarrow (a \vee x_1)$ par leurs formes développées dans la formule $x_2 \wedge (x_1 \leftrightarrow (b \wedge c)) \wedge (x_2 \leftrightarrow (a \vee (b \wedge c)))$ alors on obtient une formule en CNF équisatisfiable qui est la formule : $x_2 \wedge (\neg b \vee \neg c \vee x_1) \wedge (b \vee \neg x_1) \wedge (c \vee \neg x_1) \wedge (a \vee x_1 \vee \neg x_2) \wedge (\neg b \vee x_2) \wedge (\neg x_1 \vee x_2)$. Cet algorithme est très largement utilisé par les solveurs SAT et SMT.

2.1.4 Résoudre SAT

La difficulté du problème SAT réside en ce que chaque variable, d'une formule propositionnelle, dispose d'une valeur de vérité telle que la formule soit vraie. L'ensemble des paires (variable – valeur de vérité) est communément appelé interprétation. Sachant que pour chaque variable propositionnelle deux valeurs de vérité sont possibles, le nombre de modèles possibles est de 2^n où n est le nombre de variables de la formule. Par exemple avec une formule contenant plus de 200 variables, le nombre d'interprétations approche le nombre de particules de l'univers. Aucune machine existante ne serait capable de traiter exhaustivement de tels problème. Néanmoins nous allons voir qu'avec les algorithmes proposés dans cette section il est possible de traiter de telle formule efficacement. Étant donnée une interprétation (booléenne), vérifier la valeur de vérité d'une formule est possible en temps linéaire. Comme vu dans la section précédente, SAT est bien un problème dans NP.

L'une des première méthode pour résoudre SAT adaptée à l'automatisme des machines fut la méthode par résolution [100], introduite par Robinson. Cette méthode s'appuie sur une unique règle de dérivation, qui peut être vue comme l'application directe de la règle de coupure. Si $\varphi_1 \Rightarrow x$ et $x \Rightarrow \varphi_2$ alors $\varphi_1 \Rightarrow \varphi_2$, où x est une variable booléenne (un littéral), φ_1 une conjonction et φ_2 une clause. Cette règle s'exprime généralement sous forme clausale (voir la figure 2.1). Si l'on considère les formules représentées comme des ensembles conjonctifs d'ensembles disjonctifs, cette règle est complète pour la réfutation. Autrement dit si φ_1 n'est pas satisfaisable, alors il existe une suite finie de résolutions aboutissant à la clause vide, permettant donc de déduire qu'aucun modèle ne peut satisfaire φ_1.

L'objectif est donc d'obtenir un algorithme capable de déterminer la satisfaisabilité (ou non satisfaisabilité), pour toutes les formules de la logique propositionnelle, d'en exhiber un modèle quand la formule est satisfaisable, et de fournir une preuve dans le cas contraire. Une approche

$$\frac{(\varphi_1 \vee x) \wedge (\neg x \vee \varphi_2)}{\varphi_1 \wedge \varphi_2}\ \textsc{Res}$$

FIGURE 2.1 – Règle de résolution

tout à fait naïve, mais complète, pour résoudre SAT, consisterait à essayer exhaustivement l'ensemble de toutes les possibilités, pour chacune des variables. Ce qui en pratique n'est pas raisonnable. Une première approche, ingénieuse, et complète, au sens que pour toute formule, une réponse peut être proposée, est suggérée par Davis et Putnam, via l'algorithme DP60 [40]. Cette procédure est améliorée par Davis, Logemann et Loveland, par l'algorithme DPLL [39] présenté dans la figure 2.2. L'algorithme DPLL énumère l'ensemble des modèles, pour une formule propositionnelle de façon incrémentale. Les formules considérées par l'algorithme doivent être en forme normale conjonctive, on considère généralement un ensemble de clauses. Ceci permet à l'algorithme de procéder de façon chronologique. A chaque étape, l'algorithme va soit, essayer d'augmenter le modèle d'une nouvelle assignation, pour une variable — cette opération est appelée décision (ligne 7) — ou ajouter une contrainte de propagation (ligne 2), permettant de déduire de manière déterministe à partir du modèle partiel la valeur de vérité pour une nouvelle variable dans la formule. Si les deux valeurs de vérité vraie et fausse sont propagées pour une variable (ligne 2), alors l'algorithme retourne UNSAT. Lorsque l'algorithme a assigné toutes les variables (ligne 4), la formule est vérifiée, et l'algorithme termine en répondant SAT, sinon l'algorithme revient à sa dernière décision (ligne 8) et explore les autres possibilités. Si aucune branche n'est satisfaisable alors l'algorithme termine, et retourne la valeur UNSAT (la formule n'est pas satisfaisable). C'est un exemple canonique d'algorithme exploratoire, qui en pratique se comporte bien.

```
1  Function DPLL(Γ) :
2      if ¬ Propagate(Γ) then
3          return UNSAT
4      else if all variables are assigned then
5          return SAT
6      else
7          L ← decide_literal(Γ)
8          if DPLL(Γ ∪ {L}) == SAT then
9              return SAT
10         return DPLL(Γ ∪ {¬L})
```

FIGURE 2.2 – Algorithme DPLL

L'algorithme DPLL effectue une recherche de modèle exhaustive, chaque branche est explorée de manière chronologique. C'est-à-dire que lorsque l'interprétation obtenue ne satisfait pas la formule, l'algorithme va défaire les opérations, décisions, propagations, dans l'ordre. Cependant il est possible de procéder de façon non chronologique, et d'éliminer plusieurs possibilités

trivialement fausses. Pour cela il suffit de remonter à la décision qui est à l'origine de l'échec, de l'algorithme, puis d'apprendre les raisons de cet échec. Cette idée est exploitée par l'algorithme CDCL (Conflict Driven Clause Learning) [125] présenté dans la figure 2.3. Comme dans l'algorithme DPLL, la recherche est exploratoire. On commence par propager (ligne 3), si cela est possible. Une fois toutes les propagations effectuées, on déduit soit un modèle complet soit on décide un nouveau littéral. Plutôt que de revenir à la dernière décision, lorsque la fonction de propagation déduit la clause vide, le CDCL cherche l'origine du conflit (ligne 6), et remonte à la décision qui est à la source du conflit (ligne 7). Une variable globale est maintenue pour mémoriser la profondeur de la recherche. Dans le cas ou la profondeur courante est nulle (ligne 4) cela signifie que toutes les possibilités ont été explorées par l'algorithme, et dans ce cas il retourne UNSAT. La terminaison de cet algorithme est un peu plus subtile que celle du DPLL [87].

```
1  Function CDCL(Γ) :
2      while true do
3          if ¬ Propagate(Γ) then
4              if level == 0 then
5                  return UNSAT
6              Analyse(Γ)
7              Backjump(Γ)
8          else if all variables are assigned then
9              return SAT
10         else
11             L ← decide_literal(Γ)
12             Γ ← (Γ ∪ {L})
```

FIGURE 2.3 – Algorithme CDCL

Dans cette section nous avons décrit le fonctionnement interne d'un solveur SAT. Cependant, de nombreuses optimisations n'ont pas été décrites, comme les watched literal [82], les redémarrages rapides [62, 77] ou encore le pré-traitements [70]. De plus les opérations de propagation et d'analyse de conflits n'ont pas été étudiées en détail. Pour plus d'information le lecteur peut consulter [24, 87]. Nous allons maintenant étudier comment un solveur SMT fonctionne sur la base d'un SAT solveur en coopération avec des procédures de décisions.

2.2 SMT

Les solveurs SMT peuvent, en quelque sorte, être vus comme une évolution des solveurs SAT, pour la logique du premier ordre avec théories. En effet une grande famille de solveurs SMT, tels que cvc4 [44], z3 [42], Yices [49] ou veriT [31], reprennent l'architecture des solveurs SAT, pour traiter des problèmes de la logique du premier ordre. L'algorithme développé par Nieuwenhuis,

Oliveras, et Tinelli [87], reprend l'algorithme CDCL, et l'étend pour la logique du premier ordre avec théories, cette algorithme est communément appelé CDCL($\mathcal{T}$).

2.2.1 Syntaxe et sémantique

La logique du première ordre est un langage plus expressif que la logique propositionnelle. Il permet notamment de raisonner avec des prédicats, et des fonctions. Pour exprimer les termes et les formules de la logique du premier ordre on définit généralement une signature, qui n'est rien d'autre qu'un triplet d'ensembles de variables, prédicats et fonctions. Les fonctions (resp. prédicats) sont rangées par arités et par sortes, l'arité nulle correspond aux constantes, l'arité 1 aux fonctions (resp prédicats) à un argument, etc. En logique du premier ordre avec sortes toute fonction est sortée (et de même pour tout prédicat), c'est-à-dire que les arguments et la valeur de retour doivent appartenir à des domaines spécifiés par la sorte de la fonction. Dans cette configuration on doit donc spécifier pour chaque fonction, ou prédicat, l'arité, la sorte de chacun de ses arguments, et la sorte de retour de la fonction. Si f est une fonction d'arité 2, que la sorte de ses deux arguments est A, et que sa sorte de retour est B, on notera sa signature $f(A, A) : B$ et sa sorte $A \times A \to B$. Il faut remarquer que les prédicats sont un cas particulier, ou la sorte de retour est toujours la sorte Booléenne (notée Bool) c'est-à-dire la sorte qui sera toujours associée au domaine à deux éléments $\bot$ et $\top$.

Exemple 2.2.1. Considérons le problème suivant, en logique du première ordre avec sortes :

$$P(g(a)) \Rightarrow Q(f(a, b)) \Leftrightarrow P(f(b, b)) \vee Q(g(b)).$$

Supposons que la sorte des constantes a, et b soit A. On a donc, si la formule est correctement sortée, que f est de sorte $A \times A \to B$, g est de sorte $A \to B$, et P, Q sont tous des prédicats d'arité 1. La signature associée à ce problème est donc : $\Sigma = \{\emptyset, \{P(B) : \text{Bool},\ Q(B) : \text{Bool}\}, \{g(A) : B,\ f(A, A) : B\}\}$ •

Dans le contexte SMT, on utilise une interprétation particulière pour les fonctions, et les prédicats que l'on appèle interprétation de Herbrand. Plus précisément, dans cette interprétation chaque fonction, ou prédicat est interprété par un terme syntactique qui apparaît dans le domaine de la sorte qui lui est associé. Étant donné que les notions de sémantique ne sont que très peu abordées dans cette thèse on invitera le lecteur à consulter des ouvrages plus détaillés [51].

La notion de satisfaisabilité est elle aussi étendue. On parlera dans cette thèse de satisfaisabilité modulo théories, ce qui signifie qu'une formule peut être interprétée dans une théorie particulière ou plusieurs. Une théorie $\mathcal{T}$ consiste en un ensemble d'axiomes satisfaisables, ou autrement dit non contradictoires. Plus spécifiquement, un SMT solveur cherche à déterminer si un ensemble de littéraux, noté E est satisfaisable modulo plusieurs théories. Au première ordre, un littéral est une formule atomique qui peut être falsifié, et un atome est soit une variable, soit un prédicat (sans négation à la racine). On note que E satisfait une formule φ, dans une théorie $\mathcal{T}$, $E \models_{\mathcal{T}} \varphi$, si $E \cup \mathcal{T}$ satisfait φ.

Par abus de langage on dit qu'un ensemble de littéraux est cohérent, s'il est satisfaisable, et incohérent s'il est non satisfaisable.

Exemple 2.2.2. Supposons le symbole $\simeq$, le symbole interprété de la théorie de l'égalité. C'est-à-dire que $\simeq$ est la plus petite relation, telle que $\simeq$ est réflexive, symétrique, transitive et congruente. Prenons A et B deux sortes, $f(A) : B$, $g(B, A) : A$, $a : A$, et $b : B$.

$$\{g(f(a), a) \simeq g(b, a),\ a \simeq g(b, a)\} \not\models_{eq} \neg f(g(b, a)) \simeq f(a)$$

Le littéral $\neg f(g(b, a)) \simeq f(a)$ n'est pas une conséquence logique de l'ensemble de littéraux $\{g(f(a), a) \simeq g(b, a),\ a \simeq g(b, a)\}$ puisque on peut déduire $f(g(b, a)) \simeq f(a)$ à partir de $a \simeq g(b, a)$. Cependant en changeant par exemple la polarité du littéral $a \not\simeq g(b, a)$ dans le modèle, la formule $\neg f(g(b, a)) \simeq f(a)$ devient satisfaisable.

$$\{g(f(a), a) \simeq g(b, a),\ a \not\simeq g(b, a)\} \models_{eq} \neg f(g(b, a)) \simeq f(a)$$

•

2.2.2 CDCL($\mathcal{T}$)

Un solveur SMT utilise des procédures de décision pour traiter chaque théorie de manière indépendante. Pour gérer la structure booléenne des formules, les solveurs SMT font appel à un solveur SAT. En réalité, lorsque l'on parle de structure booléenne, on fait référence à la structure CNF de la formule, car comme pour les solveurs SAT, les solveurs SMT transforment systématiquement les formules en formes CNF. Une procédure de décision raisonne généralement sur des conjonctions de littéraux. Ce cloisonnement volontaire des tâches permet à chaque module de travailler de manière optimale, la philosophie étant que chaque module se restreint à ce qu'il sait faire de mieux. Pour assurer la coopération entre chaque théorie, deux écoles existent. Il y a la méthode de combinaison de théories de Shostak [108], et celle de Nelson-Oppen [83]. L'outil veriT utilise la seconde. Les figures 2.4 et 1.1, illustrent le fonctionnement de l'algorithme CDCL($\mathcal{T}$).

Dans la figure 2.4, on suppose qu'à l'appel de la fonction `CheckSat` les prétraitements (mise en forme normale CNF, élimination de symétries, etc.) de la formule ont été effectués en amont. La fonction effectue une recherche de modèle. Elle détermine premièrement un ensemble de littéraux pour lequel la formule est vérifiée au niveau propositionnel (ligne 3), et on suppose que la fonction `CheckBoolean`, implémente l'algorithme CDCL donné dans la figure 1.1. Pour cela l'algorithme s'appuie sur une abstraction de la formule commandée par la fonction `FO2Bool`, qui pour chaque atome de la formule, associe une variable propositionnelle. Le modèle renvoyé par le SAT solveur est ensuite vérifié pour chacune des théories dans $\mathcal{T}$, spécifiées en argument de la fonction (ligne 6). Si l'une des théories détecte une incohérence, une clause de conflit est produite par la procédure de décision (ligne 6), et cette clause est ajoutée au problème (ligne 7), c'est la phase d'apprentissage (clause learning). Si aucune incohérence n'est détectée par les théories, alors le problème est satisfaisable, et l'algorithme termine (ligne 8). Dans le cas où aucun modèle

```
1 Function CheckSat(Γ, 𝒯) :
2     while true do
3         E ← CheckBoolean(FO2Bool(Γ))
4         if E == ∅ then
5             return UNSAT
6         C = CheckTheories(E, 𝒯)
7         Γ = Γ ∪ C
8         if C == ∅ then
9             return SAT
```

FIGURE 2.4 – Algorithme CDCL($\mathcal{T}$)

propositionnel ne s'accorde avec les théories, alors le problème est non satisfaisable (ligne 4). En effet, si le solveur SAT retourne la clause vide cela signifie que l'ensemble des modèles ont été énumérés.

Maintenant attardons nous quelque peu sur ce qui fait le cœur du solveur, la fonction `CheckTheories`. Cette fonction décrite sommairement plus haut, est en réalité très complexe au sens que dans la présentation on suppose que cette fonction choisit, et exécute les procédures de décision associées aux théories passées en argument. De plus cette procédure s'occupe de faire coopérer les procédures entre elles, en utilisant la méthode de combinaison Nelson-Oppen. Pour communiquer entres elles les théories s'échangent des égalités. L'exemple 2.2.3 décrit ce processus. Le nombre de théories existantes est grand, et il ne serait par raisonnable de toutes les implémenter. De plus toutes les théories ne sont pas compatibles entres elles, et la combinaison (union) de deux théories décidables peut être indécidable. Ainsi en pratique, un bon SMT solveur, utilisable depuis un assistant de preuve, par exemple, doit au moins disposer d'une procédure de décision pour l'arithmétique, le simplex [50], pour les nombres réels, et d'une procédure de décision pour l'égalité, la fermeture de congruence [85]. À noter qu'il est parfois possible d'utiliser la procédure de décision pour les réels, pour résoudre des problèmes d'arithmétique entier cette solution ne fonctionne pas systématiquement. Pour illustrer le comportement de chaque procédure de décision, regardons au travers de l'exemple 2.2.3, les différents rôles joués par ces procédures.

Exemple 2.2.3. Supposons la formule ci-dessous φ, où $\leq, +$ and $\simeq$ sont des symboles interprétés, par leur théories respectives :

$$\varphi = a \leq b \wedge b \leq a + x \wedge x \simeq 0 \wedge [f(a) \not\simeq f(b) \vee q(a)] \wedge [f(a) \not\simeq f(b) \vee \neg q(b + x)].$$

La formule φ, ci-dessus fait intervenir deux théories, l'égalité et l'arithmétique linéaire. Pour déterminer la satisfaisabilité de cette formule il faudra donc que ces deux théories s'accordent sur un modèle proposé par le SAT solveur. Pour proposer un modèle le SAT solveur a besoin de l'abstraction propositionnelle `FO2Bool`(φ), qui est représentée ci-dessous :

$$\texttt{FO2Bool}(\varphi) = p_{a\leq b} \wedge p_{b\leq a+x} \wedge p_{x\simeq 0} \wedge (\neg p_{f(a)\simeq f(b)} \vee p_{q(a)}) \wedge (\neg p_{f(a)\simeq f(b)} \vee \neg p_{q(b+x)}).$$

Le SAT solveur pourrait ainsi proposer le modèle suivant :

$$\{p_{a\leq b},\, p_{b\leq a+x},\, p_{x\simeq 0},\, \neg p_{f(a)\simeq f(b)}\}.$$

Les procédures de décisions peuvent donc chacune à leur tour vérifier la satisfaisabilité de ce modèle $E = \{a \leq b, b \leq a + x, x \simeq 0, f(a) \not\simeq f(b)\}$. La coopération de la fermeture de congruence, pour l'égalité, et du simplex, pour l'arithmétique permet de conclure que ce modèle n'est pas satisfaisable. En effet le module arithmétique va déduire, à partir de ce modèle que a et b sont égaux. Cette égalité va être transmise à la procédure d'égalité qui à son tour déduira $f(a) \simeq f(b)$ par congruence. Malheureusement cette égalité rend le modèle incohérent. La clause de conflit devra mettre en évidence que a et b doivent être différents, de même que $f(a) \simeq f(b)$, le conflit est donc le modèle complet :

$$\neg(a \leq b) \vee \neg(b \leq a + x) \vee \neg(x \simeq 0) \vee f(a) \simeq f(b).$$

Cette clause est ajoutée à φ. Par la suite le SAT solveur est de nouveau interrogé sur le problème enrichi de la clause de conflit, et peut produire le modèle suivant :

$$E = \{a \leq b, b \leq a + x, x \simeq 0, q(a), \neg q(b + x)\}.$$

Ce modèle est de nouveau incohérent, car $a \simeq b$ peut toujours être déduit par la procédure d'arithmétique ce qui a pour conséquence de rendre $q(a)$ et $\neg q(b + x)$ égaux, et de produire un nouveau conflit.

$$\neg(a \leq b) \vee \neg(b \leq a + x) \vee \neg(x \simeq 0) \vee \neg q(a) \vee q(b + x).$$

La nouvelle formule est incohérente au niveau propositionnel, ce qui a pour conséquence que la formule complète est incohérente donc φ est non satisfaisable modulo l'union des théories de l'égalité et de l'arithmétique. •

En règle générale les procédures de décision doivent disposer d'un certain nombre de fonctionnalités pour s'accorder de façon harmonieuse au reste du solveur. En particulier, chaque procédure de décision doit être *backtrackable*, c'est-à-dire qu'elles doivent maintenir une trace des décisions, et opérations effectuées pour chaque modèle produit par le SAT solveur. En particulier, la procédure doit être capable de revenir en arrière, sans tout recalculer. L'ordre des opérations est généralement orchestré par le SAT solveur. Cette fonctionnalité est indispensable pour que le solveur puisse être utilisable sur des problèmes réalistes. La procédure de décision doit aussi produire un *modèle*, lorsque le problème est satisfaisable, et être capable de le communiquer aux autres procédures, par échange d'égalités de variables. La procédure doit être capable de produire des *explications*, et plus spécifiquement de construire une *clause de conflit*, qui met en évidence l'ensemble des littéraux incriminés dans l'incohérence du problème. Une fonctionnalité importante est la propagation d'information. Ce peut être par exemple, lorsque le solveur de théorie déduit de nouveaux littéraux qui n'apparaissent pas dans le problème original. Dans ce cas précis la procédure de décision peut suggérer au SAT solveur de prendre en compte ce nouveau littéral dans la formule. Cette optimisation permet généralement de réduire le nombre de candidats modèles dans l'espace de recherche et donc d'améliorer les performances du solveur.

2.3 SMT et quantificateurs

Les quantificateurs universels et existentiels, représentés respectivement par les symboles $\forall$, $\exists$ sont des opérateurs de la logique du premier ordre (ou similaire). Il sont utilisés pour raisonner sur des ensembles, finis ou infinis, d'éléments. Le quantificateur universel permet d'exprimer qu'une proposition est vraie pour tous les éléments d'un ensemble (ou domaine), par exemple pour tout entier naturel, la proposition $0 \simeq 0 \times x$ est vraie quel que soit le naturel x, ce qui se traduit par $\forall x : \mathbb{N}, 0 \simeq 0 \times x$. De même l'opérateur existentiel permet d'affirmer qu'une proposition est vraie pour au moins un élément d'un ensemble, par exemple il existe un entier naturel non nul se traduit par $\exists x : \mathbb{N}, x \not\simeq 0$. Ces opérateurs sont reliés à des variables, qui peuvent prendre n'importe quelle valeur de l'ensemble. On appelle la formule obtenue après remplacement des variables par des valeurs une *instance*. Pour interpréter ces opérateurs on énumère toutes les instances possibles de la proposition originale. Pour vérifier qu'une formule, avec comme opérateur de tête l'opérateur universel, est vraie, il faut vérifier que la propriété est vraie pour tous les éléments de l'ensemble, cela revient donc à évaluer la conjonction de toutes les instances. Donc pour evaluer la formule $\forall x : \mathbb{N}, 0 \simeq 0 \times x$ cela revient à évaluer la conjonction $0 \simeq 0 \times 0 \wedge 0 \simeq 0 \times 1 \wedge 0 \simeq 0 \times 2 \wedge ...$, la conjonction est infinie. Pour le quantificateur existentiel il suffit que la proposition soit vraie pour une seule instance, cela revient à exprimer une clause de toutes les instances de la formule. Néanmoins, cet opérateur est généralement éliminé via la Skolémisation [51].

La majorité des outils automatiques ne sont que partiellement complets pour ce type de raisonnement. La première limitation à cela est purement théorique, puisque il a été démontré par Gödel que la logique du premier ordre, avec l'égalité, est semi-décidable. Il n'existe donc pas de procédure de décision permettant de répondre SAT ou UNSAT sur toutes les formules de la logique du premier ordre. Cependant il existe des théories décidables avec quantificateurs, par exemple les polynômes sur les réels. Dans le domaine du raisonnement automatique on s'intéresse, donc à une forme de complétude plus restreinte, la complétude pour la réfutation de formules. Cela signifie que l'algorithme peut toujours répondre UNSAT, si le problème d'entrée n'est pas satisfaisable. Par conséquent, les outils automatiques sont développés dans le but de réfuter des problèmes de la logique du premier ordre avec quantificateurs. Plusieurs systèmes de preuve automatique ont été développés pour la logique du premier ordre : hyper-résolution [102], para-modulation [99], superposition [4, 105], tableaux et hyper-tableaux [15, 103], ou encore les méthode de calcul basés sur des matrices, appelés calculs de connexion [22, 23],...

Le problème de la satisfaisabilité modulo théories avec quantificateurs est un problème difficile (souvent indécidable) en général. La combinaison de théories avec quantificateurs est très difficile, et des théories décidables sans quantificateur, peuvent devenir indécidables avec quantificateurs. Par exemple l'arithmétique sans quantificateur est décidable mais pas l'arithmétique avec quantificateurs (arithmétique de Peano). Cependant l'arithmétique réelle reste décidable avec ou sans quantificateur. Par conséquent les solveurs SMT utilisent une approche basée sur l'instanciation, pour gérer les problèmes avec quantificateurs. Dans le cadre des solveurs SMT, le calcul CDCL($\mathcal{T}$) a été étendu pour manipuler des formules quantifiées. Le tout premier SMT solveur ayant proposé une approche par instanciation pour manipuler les quantificateurs est le

solveur Simplify [45]. Simplify peut être considéré comme un SMT solveur au sens qu'il propose une architecture basée sur la méthode de combinaison Nelson-Oppen. Une approche par instanciation de quantificateur par-dessus le cœur du solveur permet de manipuler les quantificateurs. Dans la section 1.7.2 du Handbook of Satisfiability, Clark Barrett propose deux règles qui étendent le calcul (système de transition) développé dans [87]. Cette approche par instanciation est largement inspirée des travaux de Davis et Putnam [40], proposant un calcul générant des instances de Herbrand.

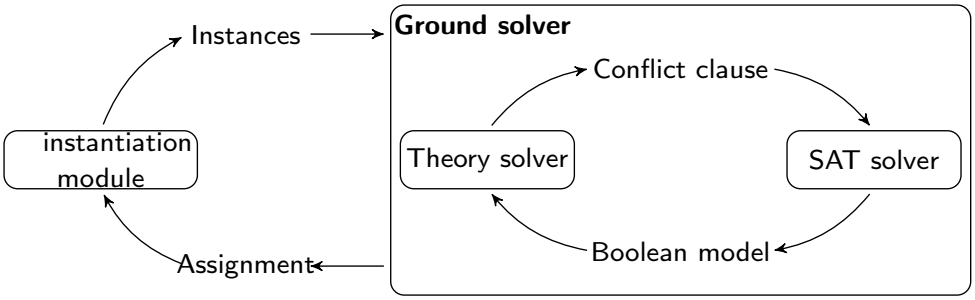

FIGURE 2.5 – Architecture SMT pour quantificateurs.

L'approche consiste à séparer les propositions qui ne contiennent pas de quantificateurs de celles qui en contiennent. L'ensemble des propositions sans quantificateur, appelé *ground problem* en anglais, est traité comme précédemment. On note E l'ensemble des littéraux qui satisfont le problème ground tant au niveau propositionnel, et sur lesquels toutes les théories s'accordent. L'ensemble des propositions quantifiées sont placées par le solveur dans un ensemble nommé Q. Comme décrit dans la figure 2.5, une fois l'ensemble E dérivée par le ground solveur, l'ensemble des formules quantifiées Q, et E sont passés à un module externe implémentant plusieurs heuristiques et stratégies permettant l'instanciation des quantificateurs. Les instances dérivées par ce module sont ajoutées au problème original. La tautologie ci-dessous est ajoutée à la formule originale, pour chaque instance dérivée à partir d'une substitution σ. Cette proposition est appelée lemme d'instanciation :

$$\forall \bar{x}.\ \varphi[\bar{x}] \Rightarrow \varphi\sigma$$

L'approche utilisée par le solveur SMT veriT pour l'instanciation des quantificateurs est une approche dite paresseuse (lazy approach). La figure 2.5, ci-dessus, schématise son fonctionnement, et l'algorithme 2.6 donné plus bas, implémente cette approche. On remarque d'ailleurs que la figure 2.6 décrit une extension du CDCL($\mathcal{T}$) (voir figure 2.4), permettant de manipuler des formules avec quantificateurs. Dans l'algorithme 2.5, la fonction `Inst` (ligne 9) joue le rôle de module d'instanciation. Elle prend en argument E, l'ensemble des formules quantifiées universellement, ainsi que les théories. Elle retourne l'ensemble des lemmes d'instances I. Si cet ensemble est vide alors la procédure termine et répond `Unknown` (ligne 10). Autrement l'ensemble des instances est ajouté au problème (ligne 12)

Plusieurs raisons peuvent expliquer un échec de la part du module d'instanciation. Généralement les approches implémentées par ce module sont soit incomplètes, les domaines ne sont pas complètement énumérés, ou la stratégie, en raison de certaines limitations, abandonne à un certain point, ou le nombre d'instances générées est volontairement limité.

```
1  Function CheckSat(Γ, T, Q) :
2      while true do
3          E ← CheckBoolean(FO2Bool(Γ))
4          if E == ∅ then
5              return UNSAT
6          C = CheckTheories(E, T)
7          Γ = Γ ∪ C
8          if C == ∅ then
9              I ← Inst(E, Q, T)
10             if I == ∅ then
11                 return Unknown
12             I = Γ ∪ I
```

FIGURE 2.6 – Algorithme CDCL($\mathcal{T}$) + Q

Plusieurs approches ont été développées pour attaquer le problème de l'instanciation. Parmi les stratégies développées on recense : l'approche par énumération [95], l'approche par triggers [41, 45] ou encore l'approche par conflits [97]. Il existe aussi d'autres approches, notamment visant à déterminer la satisfaisabilité des problèmes contenant des quantificateurs, telle que l'approche MBQI [55]. Les approches de ce type ne seront pas discutées dans cette thèse, car elles sortent du cadre de notre étude. Nous nous concentrons uniquement sur les problèmes avec quantificateurs qui sont réfutables.

Exemple 2.3.1. Considérons l'ensemble de littéraux $E = \{\neg \mathsf{P}(\mathsf{a}), \neg \mathsf{R}(\mathsf{b}), \mathsf{R}(\mathsf{a})\}$ et la formule quantifiée $\varphi = \forall x. \mathsf{P}(x) \vee \neg \mathsf{R}(x)$. Le problème de l'instanciation revient ici à trouver un ensemble I d'instances de φ tel que I est incohérent avec E.

Étant donné que les uniques termes sur l'univers de Herbrand sont a et b, il est suffisant, d'après le théorème de Herbrand, de considérer l'ensemble d'instances I contenant les instances suivantes $\mathsf{P}(\mathsf{a}) \vee \neg \mathsf{R}(\mathsf{a})$ et $\mathsf{P}(\mathsf{b}) \vee \neg \mathsf{R}(\mathsf{b})$.

De plus l'instance $\mathsf{P}(\mathsf{a}) \vee \neg \mathsf{R}(\mathsf{a})$, est contradictoire avec E. Ce type d'instance est appelé instance de *conflit*. Nous verrons plus loin qu'il est essentiel de pouvoir dériver ce type d'instance, et notamment dans le chapitre 5 nous montrerons que ces instances sont plus utiles que les autres.
•

Instanciation par énumération. D'après le théorème de Herbrand [51], toute formule quantifiée $\forall x \,.\, \psi[x]$ peut être vue comme conjonction infinie $\bigwedge_t \psi[t]$ sur l'ensemble des termes de Herbrand t. Le théorème de compacité assure qu'il existe toujours un sous-ensemble fini de formules contradictoires, pour tout ensemble de formules contradictoires pour la logique du premier ordre avec l'égalité. Par conséquent il est suffisant d'énumérer les instances de Herbrand pour

obtenir un algorithme complet pour le problème de l'instanciation.

Instanciation par trigger. Plutôt que d'instantier à l'aveugle les formules quantifiées en utilisant des termes de Herbrand arbitraires, l'instanciation par trigger identifie un ensemble de motifs, qui sont des termes contenant des variables libres, au sein de chaque formule quantifiée. Ces motifs appelés *triggers*, sont mis en correspondance avec des termes qui apparaissent dans E, afin de dériver des instances directement en lien avec le problème traité. Par exemple si l'on considère la formule $\forall x.\, \mathsf{f}(\mathsf{g}(x)) \simeq x$. Un trigger adapté pour cette formule pourrait être $\mathsf{f}(\mathsf{g}(x))$. Un tel trigger permettrait de déduire la substitution $x \mapsto \mathsf{c}$, à partir de $E = \{\mathsf{a} \simeq \mathsf{f}(\mathsf{b}), \mathsf{b} \simeq \mathsf{g}(\mathsf{c})\}$, ce qui permettrait de dériver, par l'application de la substitution $x \mapsto \mathsf{c}$ à la formule $\forall x.\, \mathsf{f}(\mathsf{g}(x)) \simeq x$, l'instance $\mathsf{f}(\mathsf{g}(c)) \simeq c$. Plusieurs stratégies ont été développées dans le but de rendre cette approche plus efficace [5, 41, 45, 48].

Instanciation par conflict. L'approche par conflit est une technique d'instanciation originalement introduite par Reynolds et al. [97]. Cette technique a pour objectif d'améliorer les performances des solveurs SMT sur des problèmes avec quantificateurs, et généralement non satisfaisables. L'approche par conflit considère successivement chaque formule quantifiée $\forall \bar{x}.\, \varphi$ dans Q, et essaye de dériver une substitution σ, pour l'ensemble des variables universellement quantifiées $\bar{x}$, telle que la formule est contradictoire avec E, c'est à dire, $E \models \neg\varphi\sigma$. Ces substitutions peuvent être générées en utilisant l'algorithme CCFV (Congruence Closure with Free Variables), développé par Barbosa et al [8]. Considérons l'exemple 2.3.1, la formule $\mathsf{P}(\mathsf{a}) \vee \neg\mathsf{R}(\mathsf{a})$ est une instance de conflit, et peut-être obtenue à partir de E et φ en utilisant l'algorithme CCFV.

Le solveur SMT veriT implémente les trois stratégies d'instanciation décrites plus haut. veriT tente généralement en premier lieu d'appliquer l'approche par conflit ; si cette approche échoue, veriT essaye la méthode des triggers. En dernier ressort, si aucune de ces méthodes ne permet d'obtenir une solution, alors veriT lance la procédure par énumération qui est évidement plus coûteuse.

On remarque que lorsque la stratégie d'instanciation par conflit produit une solution, c'est-à-dire un ensemble d'instances, ces instances peuvent être considérées comme utiles pour la progression générale du solveur vers une contradiction, car elles contredisent le modèle produit par le SAT solveur en accord avec les théories. Idéalement chaque instance produite par le module d'instanciation devrait être de cette nature. La notion d'utilité sera développé dans le chapitre 6. Malheureusement, l'approche par conflit est limitée, et ne peut déduire d'instance de conflit qu'à partir d'une seule clause à la fois dans la formule. Ainsi afin d'obtenir une méthode complète pour la réfutation, les approches par triggers et par énumération sont indispensables. Ces deux méthodes sont cependant beaucoup trop exhaustives, et produisent un trop grand nombre d'instances, empêchant parfois le solveur de progresser dans sa recherche. Nous verrons donc dans cette thèse comment il est possible de filtrer les instances produites par ces deux approches dans le but de n'introduire que des instances utiles au sens évoqué plus haut. Et ainsi conduire la recherche de modèles plus rapidement à une contradiction, s'il en existe une.

2.4 Les challenges d'aujourd'hui et contribution de la thèse

Aujourd'hui les solveurs SMT bénéficient des évolutions technologiques tant au niveau des solveurs SAT que des procédures de décisions. Cependant les solveurs SMT restent en proie à de multiples défauts. À la fin de la section précédente de ce chapitre nous avons mis en évidence le manque d'efficacité des solveurs SMT lorsque les problèmes rencontrés contiennent des quantificateurs. Cette problématique freine l'usage des solveurs SMT dans le domaine de méthodes formelles. En effet la montée en puissance des outils de preuves de programmes du type Why3, Atelier B, ou bien des assistants de preuves comptent sur l'efficacité des outils automatiques de preuve, pour décharger le maximum de contraintes "redondantes" ou supposées facile, pour permettre aux utilisateurs de se concentrer sur le cœur du problème. Malheureusement les outils s'appuient sur des bibliothèques contenant de grande collection de théorèmes et de lemmes permettant de démontrer des propriétés. La majorité de ces théorèmes utilisent la quantification, rendant la tâche des solveurs SMT très difficile. Les systèmes de preuves automatiques basés sur la superposition, ont la réputation de manipuler les formules contenant des quantificateurs de façon efficace, cependant ces outils peinent face à des problèmes contenant beaucoup de raisonnement avec théories, par exemple arithmétiques, ou encore des data-types. Des systèmes hybrides ont vu le jour récemment permettant de combiner la force des systèmes basés sur la résolution avec des solveurs SMT, comme AVATAR [120], implémenté dans le prouveur Vampire. D'autres tentatives du coté SMT ont vu le jours sans implémentation largement diffusée, je fais notamment référence au système CDCL($\Gamma + \mathcal{T}$) [29,30], qui est en réalité une extension du calcul CDCL($\mathcal{T}$) aux règles de calcul de superposition. En tout état de cause, pour que les solveurs soient plus performants, et plus polyvalents il est nécessaire d'améliorer les méthodes de raisonnement. Dans cette thèse nous proposons donc d'améliorer l'architecture des solveurs SMT sous deux aspects. L'aspect principal de cette thèse, discuté dans l'introduction, met en avant l'écart entre les outils de méthodes formelles, et les solveur SMT. Dans la première partie de cette thèse nous proposons donc une extension du solveur SMT veriT pour l'ordre supérieur. Sur un second plan nous proposons une approche par apprentissage permettant d'améliorer l'instanciation des quantificateurs, et une seconde approche plus pragmatique visant à réduire le nombre d'instances introduites dans le modèle via un certain nombre d'heuristiques.

3

Automatiser la logique d'ordre supérieur pour SMT

Sommaire

3.1 Introduction

La vérification formelle de systèmes complexes est un processus lourd et coûteux qui nécessite très souvent la traduction de principes mathématiques simples, et souvent considérés comme acquis pour un raisonnement humain. A mi-chemin entre le raisonnement humain et le langage machine, la logique d'ordre supérieur est un langage mathématique formel qui offre un haut

niveau d'expressivité. En comparaison avec la logique du premier ordre, la logique d'ordre supérieur offre la possibilité de quantifier sur des fonctions, et de façon plus générale, offre un cadre plus souple pour le raisonnement mathématique. Elle est notamment largement utilisée dans les assistants de preuve pour fournir des preuves mathématiques vérifiables par une machine. L'un des principaux défis de ces outils est d'automatiser autant que possible la production de ces preuves formelles, dans le but de réduire la charge de travail pour les utilisateurs. Une approche efficace pour parvenir à automatiser la résolution d'obligations de preuves consiste à s'appuyer sur des démonstrateurs automatiques moins expressifs mais totalement automatisés. Des systèmes tels que HOLyHammer, MizAR, Sledgehammer, et Why3, permettent de déléguer en un clic des obligations de preuves à des démonstrateurs automatiques basés sur la logique du premier ordre. L'utilisation de tels outils a conduit, ces dernières années, à une amélioration considérable de l'automatisation dans le domaine des méthodes formelles et plus particulièrement pour les assistants de preuve [27]. À l'heure actuelle, il n'existe qu'une poignée de démonstrateurs automatiques capables de manipuler des formules de la logique d'ordre supérieur. Des démonstrateurs automatiques tels que Leo-III [111] et Satallax [33], reposent sur une approche stratifiée : les problèmes d'ordre supérieur sont décomposés, de façon à déléguer le raisonnement de premier ordre à des démonstrateurs automatiques de premier ordre. Toutefois, comme indiqué dans les travaux de [21, 79, 122], dans les deux cas, la réduction vers la logique du premier ordre présente des inconvénients : les encodages complets, comme ceux effectués par les *hammers*, peuvent entraîner des problèmes de performance, de correction ou de complétude. D'autre part les démonstrateur tels que Leo-III et Satallax sont avant tout conçus et optimisés pour la logique d'ordre supérieur. Par conséquent ces solveurs peinent à effectuer des calculs lorsque la quantité de raisonnement du premier ordre est trop importante. Et en pratique les problèmes provenant des assistants de preuve ne contiennent que très peu de raisonnement purement d'ordre supérieur. L'approche présentée dans ce chapitre vise à surmonter ces lacunes en étendant les solveurs SMT pour la prise en charge native de la logique d'ordre supérieur.

Les deux principaux défis à relever pour étendre les solveurs SMT à l'ordre supérieur résident dans le traitement des *applications de fonctions partielles*, et des *variables fonctionnelles*, c'est-à-dire des variables liées à des quantificateurs et qui peuvent être remplacées par des fonctions. L'application partielle des arguments de fonctions affectent principalement la représentation des termes et des algorithmes de bases dans le solveur SMT, qui, au premier ordre, repose sur le fait que toutes les fonctions sont totalement appliquées. La gestion des variables fonctionnelles à quant à elle un impact sur les techniques d'instanciation des quantificateurs, qui doivent tenir compte de la position de ces variables quantifiées dans les formules, et produire des solutions plus riches. De plus, souvent, les problèmes d'ordre supérieur ne peuvent être prouvés que si les variables fonctionnelles sont instanciées avec des termes λ synthétisés (en d'autres termes cela consiste à trouver une fonction), généralement via l'algorithme d'unification d'ordre supérieur [46], qui en règle générale est indécidable.

Contributions. Nous présentons dans ce chapitre, deux approches permettant d'étendre les solveurs SMT à la logique d'ordre supérieur. Une première approche appelée *pragmatique*, et qui

a été implémentée et développée par Haniel Barbosa, Andrew Reynolds, Cesare Tinelli et Clark Barrett dans le solveur SMT cvc4 est décrite en section 3.5. Nous présentons ce travail dans sa quasi intégralité ici, bien que je ne sois pas contributeur principal de ce travail, il reste indissociable de mon propre travail, qui est présenté en section 3.6 pour la partie ground solveur. Cette approche permet d'étendre de façon « simple » (envisageable) des solveurs SMT avec une architecture complexe. Elle s'appuie en particulier sur une variante de l'encodage applicatif. Dans une seconde approche appelée *redesign* (section 3.6), nous repensons les structures de données du solveur SMT veriT, et développons de nouveaux algorithmes visant spécifiquement le raisonnement en logique d'ordre supérieur. Cette approche permet notamment de conduire à de bon résultats, néanmoins cette approche ne peut être implémenté dans des solveurs tels que cvc4 ou z3 car elle nécessite une restructuration profonde de l'architecture du solveur. C'est pourquoi cette approche à spécifiquement été implémentée dans le solveur veriT, qui est un solveur plus *léger*, c'est-à-dire que la base de code est plus petite. En outre, cette approche offre une plus grande souplesse pour développer ultérieurement de nouvelles techniques particulièrement adaptées au raisonnement d'ordre supérieur. En particulier cette approche simplifiée permet l'implémentation de nouvelles théories dans veriT telles que la théorie des data-types. Les algorithmes d'instanciation utilisés pour ces approches ne sont pas formellement étendus à l'unification d'ordre supérieur, mais une heuristique est proposée. En effet, le passage à l'instanciation d'ordre supérieur est un défi important que nous laissons pour de futurs travaux. Nous présentons une évaluation expérimentale approfondie (section 3.8) des deux approches respectivement mises en œuvre dans les solveurs veriT, et cvc4. Outre les comparaisons avec des démonstrateurs d'ordre supérieurs, nous évaluons également les approches implémentées dans veriT et cvc4, avec l'approche standard reposant sur un encodage applicatif au premier ordre (sans utiliser les extensions). Les résultats montrent des améliorations significatives sur les différentes batteries de problèmes.

Travaux similaires. En 1969, Robinson [101] est l'un des premiers à introduire une traduction pour réduire le raisonnement d'ordre supérieur à la logique du premier ordre, une traduction qui a largement inspiré les outils tels que Sledgehammer [92] et CoqHammer [37] qui s'appuient sur cette idée pour automatiser le raisonnement d'ordre supérieur via des démonstrateurs automatiques du premier ordre. Les premiers travaux portant sur l'automatisation de la logique d'ordre supérieur remontent au système de résolution développé par Andrews [3], en 1971, et plus tard, en 1995, le calcul de tableaux d'ordre supérieur de Kohlhase [72]. Ces systèmes inspirent largement les démonstrateurs automatiques modernes tels que LEO-II [20] et Leo-III [111], basés respectivement sur une variante de la résolution et la para-modulation d'ordre supérieur. Satallax [33], est quant à lui fondé sur un calcul de tableau d'ordre supérieur guidé par un solveur SAT. Notre approche est conceptuellement plus proche des travaux récents de Blanchette et al. [17, 122] sur la généralisation du calcul de superposition [4, 88] pour le raisonnement d'ordre supérieur. Une première partie de ces travaux ciblent le fragment sans λ de la logique d'ordre supérieur. Ce travail aboutit à une extension du calcul de superposition complète pour la réfutation [17], et à une implémentation dans le démonstrateur Zipperposition [36]. L'approche a par la suite été intégrée dans le démonstrateur E [122]. Plus récemment, le calcul de superposition a été étendu pour prendre en compte les λ-expressions [16]. Couplé à un nouvel algorithme d'unification d'ordre

supérieur dit pragmatique [121], le démonstrateur Zipperposition à obtenu des résultats impressionnants lors de la dernière compétition CASC (2020), et a largement dominé, pour la première fois, la catégorie ordre supérieur.

D'abord nous présentons la logique d'ordre supérieur et les différentes propriétés associées à cette logique en section 3.2. Nous présentons en section 3.3 une extension du langage SMT-LIB, permettant de prendre en compte les formules de la logique d'ordre supérieur dans les solveurs SMT ainsi que les différentes règles d'inférence pour le typage des termes. Dans la section 3.4, nous discutons les différentes contraintes qu'impliquent l'extension des solveurs SMT à l'ordre supérieur. En section 3.5, nous présentons l'approche pragmatique implémenté par cvc4, puis dans la section 3.6 l'approche *redesign* implémentée dans le solveur veriT. Dans la section 3.7, nous présentons l'extension développée dans le solveur veriT du module d'instanciation. À la fin de cette section, dans la sous section 3.7.1, nous parlons de l'approche développée, communément avec nos collègues de l'Université d'Iwoa et de Stanford, pour synthétiser des fonctions, et présentons l'approche développée par nos collègues pour simuler l'unification d'ordre supérieur par le biais d'un axiome. Dans le chapitre suivant 4, nous détaillons l'implémentation de la procédure de décision implémentée dans veriT. Nous présentons dans la section 3.8 plusieurs résultats expérimentaux. Nous terminons ce chapitre par une conclusion en section 3.9.

3.2 Notations : langage et types

Dans cette section nous nous intéressons à la syntaxe, et aux règles de typage de la logique d'ordre supérieur. Dans ce travail nous utilisons le langage d'ordre supérieur dit monomorphique.

Types. Le langage $\mathscr{L}$ est défini par des opérateurs binaires associatifs à droites, appelés *constructeurs de types* $\to$, $\times$, et des ensembles infinis dénombrables $\mathcal{S}$, $\mathcal{X}$ et $\mathcal{F}$, respectivement de *types atomiques*, de *variables*, et de *symboles de fonction*. Nous utilisons les notations $\bar{a}_n$ et $\bar{a}$ pour spécifier le tuple $(a_1, \ldots, a_n)$ ou le type produit $a_1 \times \cdots \times a_n$, en fonction du contexte, avec $n \geq 0$. Nous étendons cette notation aux opérateurs binaires de manière naturelle. Par exemple pour l'opérateur binaire $\bowtie$, la notation $\bar{a}_n \bowtie \bar{b}_n$ dénote l'expression $a_1 \bowtie b_1, \ldots, a_n \bowtie b_n$. Un *type* τ est soit un élément de $\mathcal{S}$ soit un *type fonctionnel* $\bar{\tau}_n \to \tau$ du type produit $\bar{\tau}_n = \tau_1 \times \cdots \times \tau_n$ vers le type τ. Les éléments de $\mathcal{X}$ et $\mathcal{F}$ sont annotés par des types, tel que $x : \tau$ est une variable de type τ, et $f : \bar{\tau}_n \to \tau$ est un symbole de fonction *d'arité* n, et de type $\bar{\tau}_n \to \tau$. Nous identifions les symboles de fonction de type $\bar{\tau}_0 \to \tau$ avec les symboles de fonction de type τ, que nous appelons *constantes* lorsque le type τ n'est pas un type fonctionnel. Pour simplifier la présentation, nous omettrons parfois les annotations de type.

Termes. L'ensemble des termes est défini inductivement : chaque variable $x : \tau$ est un terme de type τ. Pour les variables $\bar{x}_n : \bar{\tau}_n$ et le terme $t : \tau$ de type τ, l'expression $\lambda \bar{x}_n . t$ est un terme de type $\bar{\tau}_n \to \tau$, appelé λ-*abstraction*, dont les variables $\bar{x}_n$ sont *liées* dans le *corps* t. Une

variable est considérée comme *libre* dans un terme, si elle n'est pas liée par une construction de type λ-abstraction. Pour un symbole $f : \bar{\tau}_n \to \tau$, et les termes $t_1 : \tau_1, \ldots, t_m : \tau_m$, avec $m \leq n$, l'expression $f(\bar{t}_m)$, est un terme, appelé une *application* de la *tête* de l'application aux *arguments* $\bar{t}_m$. L'application est dite *totale*, et a le type τ si $m = n$; c'est une application *partiel*, de type $\tau_{m+1} \times \cdots \times \tau_n \to \tau$, si $m < n$. Une λ-*application* est une application où la tête est une λ-abstraction. La relation de sous-terme est définie récursivement : un terme est un sous-terme de lui-même ; si un terme est une application, tous les sous-termes de ses arguments sont aussi des sous-termes. Notons que c'est une définition plus faible que la définition standard de sous-termes pour la logique d'ordre supérieur, qui inclut aussi l'ensemble des applications de têtes, et toutes les applications partielles. *L'ensemble de tous les sous termes* d'un terme t est noté $\mathbf{T}(t)$ (remarque t appartient à $\mathbf{T}(t)$).

Fonctions et prédicats. Contrairement à la logique du premier ordre, où prédicats, et fonctions sont distincts, et que les connecteurs logiques sont des constructions de langage, en logique d'ordre supérieur le type Booléen, noté o est un type de base appartenant à $\mathcal{S}$, et l'ensemble des connecteurs logiques sont des symboles de fonction. L'ensemble $\mathcal{F}$ contient les constantes Booléennes $\top$, $\bot$, la fonction unaire $\neg$, les fonctions binaires $\wedge$, $\vee$, pour tous les types τ, une famille de symbole d'égalité $\simeq : \tau \times \tau \to o$ et une famille de symboles $\mathsf{ite} : o \times \tau \times \tau \to \tau$. Ces symboles sont interprétés de la même façon que pour les constantes logiques, les connecteurs, l'identité, et les constructions *if-then-else* (ITE). Nous faisons référence aux termes de types o comme des *formules*, et des termes de types $\bar{\tau} \to o$ comme des *prédicats*. Un *atome* est une application totale d'un prédicat. Un *littéral* ou une *contrainte* est un atome ou sa négation. Nous supposons, par simplicité, que le langage contient les lieurs $\forall$ et $\exists$ de types $(\tau \to o) \to o$ sur les formules, définis comme précédemment, en plus du lieur λ. Une formule ou un terme est dit(e) *ground*, si il (elle) ne contient aucune variable libre. Nous utilisons le symbole $=$ pour l'égalité syntaxique entre les termes. Nous utilisons les noms de symboles a, b, c, f, g, h, p pour les symboles de fonction ; w, x, y, z pour désigner des variables en général ; F, G pour des variables de type fonctionnelles ; r, s, t, u pour des termes ; et φ, ψ pour des formules. La notation $t[\bar{x}_n]$ décrit un terme dont les variables libres sont incluses dans le tuple de variables distinctes $\bar{x}_n$; $t[\bar{s}_n]$ est le terme obtenu après application simultanée des substitutions des variables $\bar{x}_n$ par les termes $\bar{s}_n$ dans le terme t.

Encodage applicatif. Nous supposons que $\mathcal{F}$ contient une famille de *symboles applicatifs* $@ : (\bar{\tau}_n \to \tau) \times \tau_1 \to (\tau_2 \times \ldots \times \tau_n \to \tau)$, pour tout $n > 1$. Nous utilisons cette famille de symboles pour modéliser le concept de curryfication des termes de type fonctionnel $\bar{\tau}_n \to \tau$. Par exemple, étant donné le symbole de fonction $f : \tau_1 \times \tau_2 \to \tau_3$ et le symbole applicatif correspondant $@ : (\tau_1 \times \tau_2 \to \tau_3) \times \tau_1 \to (\tau_2 \to \tau_3)$, le termes $@(f,\ t_1)$ est équivalents au terme $\lambda x_2 : \tau_2.f(t_1, x_2)$. L'encodage applicatif est une approche bien connue permettant de s'appuyer sur des démonstrateurs automatiques de premier ordre, pour réaliser des raisonnements d'ordre supérieur. Cet encodage convertit chaque type fonctionnel en un type atomique, chaque symbole d'arité n en symbole de constante (c.-à-d. d'arité nulle), et utilise le symbole $@$ pour encoder chaque application. Ainsi toute application, partielle ou non, devient totale, et toute quantifi-

cation sur des variables de type fonctionnel se transforme en quantification sur des variables de type atomique, comme à l'accoutumée pour la logique du premier ordre.

Skolemisation et axiome du choix. L'élimination des quantificateurs existentielle est une étape importante dans le processus de preuve. Pour éliminer ce type de quantificateur, il existe plusieurs options. L'approche naïve consiste à trouver la bonne instance, mais c'est en règle général très difficile, et équivalent à un problème de synthèse de fonction [59]. Une deuxième approche consiste à supposer l'existence d'un sous ensemble, non vide, de termes, au travers d'une fonction fraîchement introduite. C'est généralement la solution privilégiée par les systèmes de preuves automatiques, et interactifs. Le ϵ-calcul [81], repose sur le schéma d'axiome suivant :

$$\exists(x:\tau)A[x] \Leftrightarrow A(\epsilon x A[x]) \qquad \forall(x:\tau)A[x] \Leftrightarrow A(\epsilon x \neg A[x])$$

Ce principe, permet au moyen d'un combinateur ϵ, d'éliminer toute forme de quantification existentiel et universel. Cette approche est notamment utilisée par l'assistant de preuve Isabelle. Le principe de skolémisation, permet d'éliminer uniquement les quantificateurs existentiels. Supposons la formule ci-dessous en forme normale prénexe, où p est un symbole de prédicat d'arité $n+1$:

$$\forall(x_1:\tau_1)\ldots\forall(x_n:\tau_n)\ \exists(y:\sigma)\ p(x_1,\ldots,x_n,y)$$

Le principe de skolémisation consiste à remplacer la variable y existentiellement quantifiée par une nouvelle fonction d'arité n, qui prend en arguments les variables universellement quantifiées. Cette transformation permet ainsi d'éliminer le lieur existentiel comme suit :

$$\forall(x_1:\tau_1)\ldots\forall(x_n:\tau_n)\ p(x_1,\ldots,x_n,f(x_1,\ldots,x_n))$$

Lorsque le nombre de quantificateurs universels qui précèdent un quantificateur existentiel, est nul, alors le symbole de Skolem introduit est une constante. Il est important de noter que le principe de Skolemisation peut être beaucoup plus difficile à appliquer à l'ordre supérieur, en particulier, si l'on doit se passer de l'axiome du choix [80]. De plus d'autres complications peuvent arriver lors de l'unification de λ-termes (voir par exemple [18, 19]). Par conséquent, nous supposons l'existence de l'axiome d'extensionnalité, ci-dessous 3.2, et de l'axiome du choix dans ce travail, similairement aux démonstrateurs automatiques d'ordre supérieur cités plus haut.

$$\forall(F:\sigma\to\tau)\forall(G:\sigma\to\tau)\ (\forall(x:\sigma)\ F\,x \simeq G\,x) \Rightarrow F \simeq G$$

D'un point de vue sémantique nous supposons que les formules sont interprétées dans la sémantique de Henkin [19, 61]. Concrètement, le domaine d'interprétation des fonctions est restreint aux termes représentables sur le langage, de sorte que la recherche de contradiction s'appuie sur une simple extension des théorèmes de Herbrand.

3.3 Une extension syntaxique pour le langage SMT-LIB

Actuellement, la version courante du langage SMT-LIB est la version 2.6 [12]. La version 3 est en préparation. Bien que certaines discussions visant à étendre ce langage à la logique d'ordre

supérieur aient eu lieu dans le passé, notamment pour inclure les λ-abstractions, le format est actuellement basé sur une logique du premier ordre. Nous proposons ici une extension de ce langage permettant d'exprimer des constructions d'ordre supérieur : fonctions d'ordre supérieur avec applications partielles, λ-abstractions, et quantificateurs autorisant l'usage de variables d'ordre supérieur. Notre extension s'inspire des travaux sur le TIP (Tools for Inductive Provers) [104], qui est une autre extension pragmatique de la SMT-LIB. La SMT-LIB 3.0 acceptera un langage très proche.

La SMT-LIB contient des commandes permettant de définir des types atomiques, et des fonctions, mais pas de type fonctionnel. Nous étendons ci-dessous le langage afin que les types fonctionnels puisse être construits :

$$\begin{array}{rcl} \langle sort \rangle & ::= & \langle identifier \rangle \mid \texttt{(} \langle identifier \rangle \ \langle sort \rangle^{+} \texttt{)} \\ & \mid & \texttt{(} \texttt{->} \ \langle sort \rangle^{+} \ \langle sort \rangle \texttt{)} \end{array}$$

La deuxième ligne correspond à l'extension de la grammaire des types aux types fonctionnels.

L'extension de la grammaire des termes est assez simple. Pour cela nous ajoutons une règle pour les λ-abstractions, et nous généralisons les constructions applicatives :

$$\begin{array}{rcl} \langle term \rangle & ::= & \langle spec\_constant \rangle \\ & \mid & \langle qual\_identifier \rangle \\ & \mid & \texttt{(} \langle term \rangle \ \langle term \rangle^{+} \texttt{)} \\ & \mid & \texttt{(} \texttt{lambda} \ \texttt{(} \langle sorted\_var \rangle^{+} \texttt{)} \ \langle term \rangle^{+} \texttt{)} \\ & \mid & \texttt{(} \texttt{let} \ \texttt{(} \langle var\_binding \rangle^{+} \texttt{)} \ \langle term \rangle \texttt{)} \\ & \mid & \texttt{(} \texttt{forall} \ \texttt{(} \langle sorted\_var \rangle^{+} \texttt{)} \ \langle term \rangle \texttt{)} \\ & \mid & \texttt{(} \texttt{exists} \ \texttt{(} \langle sorted\_var \rangle^{+} \texttt{)} \ \langle term \rangle \texttt{)} \\ & \mid & \texttt{(} \texttt{match} \ \langle term \rangle \ \texttt{(} \langle match\_case \rangle^{+} \texttt{)} \texttt{)} \\ & \mid & \texttt{(} \texttt{!} \ \langle term \rangle \ \langle attribute \rangle^{+} \texttt{)} \\ \langle sorted\_var \rangle & ::= & \texttt{(} \langle symbol \rangle \ \langle sort \rangle \texttt{)} \end{array}$$

Dans la grammaire ci-dessus la règle $(\langle term \rangle \langle term \rangle^{+})$ remplace la règle $(\langle qual\_identifier \rangle \langle term \rangle^{+})$, permettant d'exprimer des termes de la logique d'ordre supérieur. Notons, que la règle de grammaire pour les quantificateurs ne nécessite pas de modification particulière, puisque les types ont été étendus pour tenir compte des types fonctionnels.

Exemple 3.3.1. La déclaration `(declare-fun f (Int) (-> Int Int))`. permet de définir une fonction prenant un entier comme argument et renvoyant une fonction des entiers vers les entiers. Le code ci-dessous illustre un exemple d'usage de fonctions d'ordre supérieur, et d'applications partielles :

```
1 (set-logic UFLIA)
```

```
2 (declare-fun g (Int) (-> Int Int))
3 (declare-fun h (Int Int) Int)
4 (declare-fun f ((-> Int Int)) Int)
5 (assert (= (f (h 1)) ((g 1) 2)))
6 (exit)
```

Dans le code ci-dessus, le terme `(g 1)` est une fonction de type (`Int -> Int`). Il est ensuite appliqué à `2` dans l'expression `((g 1) 2)` du type `Int`. L'expression `(h 1)` est une application partielle de la fonction binaire `h`, et est donc une fonction unaire. Le terme `(f (h 1))` est donc bien typé et est de type `Int`. •

Exemple 3.3.2. L'exemple suivant présente une λ-abstraction :

```
1 (set-logic UFLIA)
2 (declare-fun g (Int) (Int))
3 (assert
4   (= ((lambda ((f (-> Int Int)) (x Int)) f x) g 1) (g 1)))
5 (exit)
```

Ci-dessus, le terme `(lambda ((f (-> Int Int)) (x Int)) f x)` est une λ-abstraction qui prend une fonction `f`, et un entier `x` comme argument. Elle est appliquée à `g` et `1`, et le terme pleinement appliqué est déclaré égal à `(g 1)`. L'assertion est une tautologie, vraie par réflexivité de l'égalité, après application de la β-réduction. •

Règles de typage Dans ce qui suit, nous donnons les détails de l'extension du jugement de typage pour le langage de la SMT-LIB. Ces règles étendent l'ensemble des règles actuelles de la SMT-LIB. Cette extension est simple. Seules les règles de typage pour l'abstraction λ et la généralisation de son application sont nouvelles. Un jugement est composé de deux éléments. À gauche, la **signature** Σ qui est une paire composé de symboles de fonction et de symboles de constantes. Ces symboles sont annotés par leurs types. À droite, le terme est annoté par son type. La notation $\Sigma[x : \tau]$ indique la signature qui associe x au type τ dans la signature Σ. Les règles ci-dessous décrivent l'extension du jugement de typage.

$$\frac{}{\Sigma[x:\tau] \vdash x : \tau}\ \texttt{var} \qquad \frac{\Sigma[x:\sigma] \vdash t : \tau}{\Sigma \vdash \lambda x.t : \sigma \to \tau}\ \texttt{lambda}$$

$$\frac{\Sigma \vdash u : \sigma \quad \Sigma[x:\sigma] \vdash t : \tau}{\Sigma \vdash \textbf{let}\ x = u\ \textbf{in}\ t : \tau}\ \texttt{let} \qquad \frac{\Sigma \vdash u : \sigma \to \tau \quad \Sigma \vdash v : \sigma}{\Sigma \vdash u\ v : \tau}\ \texttt{app}$$

Quelques commentaires sur l'implémentation. Dans la grammaire du λ-calcul, tous les termes sont sous forme curryfiée. Plus précisément, l'application est une construction syntaxique, ce qui signifie qu'un sous-terme peut aussi être une fonction. veriT est un solveur SMT pour la logique du premier ordre qui utilise une représentation des termes appelée DAG, avec partage maximal des termes, et cette structure de termes est réutilisée pour la syntaxe présentée ci-dessus de la façon suivante. Les types sont traités, en amont, de façon à obtenir une représentation uniforme de chaque type pour tous les termes. Pour pallier les problématiques liées au parenthèsage nous utilisons une transformation qui aplatit toutes les expressions de type fonctionnel. Par exemple le type de la fonction f : (`Int` $\rightarrow$ (`Int` $\rightarrow$ `Int`)) admet plusieurs parenthèsages, et sa version aplatie est la suivante : f : (`Int` $\rightarrow$ `Int` $\rightarrow$ `Int`). Intuitivement cette transformation ramène tous les types fonctionnels vers une représentation unique, au parenthèsage près. L'algorithme 3.1 ci-dessus exprimé en OCaml, donne une idée de la transformation mise en œuvre dans veriT. Le constructeur `Arrow of sort list` représente la flèche de type $\rightarrow$. La fonction `last_of l` renvoie le dernier élément de `l`, et `unhook_last l` renvoie la liste `l` sans le dernier élément. Cette fonction est appelée lors du processus d'analyse syntaxique. Intuitivement cette fonction parcours le type d'un terme, si le type est fonctionnel, `Arrow` (ligne 3), alors on vérifie qu'il n'existe pas de type fonctionnel en tête dans le type de retour de la fonction, c'est-à-dire le dernier argument dans la liste des arguments de `Arrow lty` (ligne 5). Si un des arguments de `Arrow lty` est de type fonctionnel alors cela signifie que le type n'est pas plat et est donc de la forme $(\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3))$ (ligne 6). On aplatie en profondeur `Arrow l` (ligne 7), puis on concatène la liste `lty`, à laquelle on retire le type fonctionnel (ligne 8), au type aplatie `append_type` (ligne 9). Ligne 10, si le constructeur n'est pas `Arrow` (donc fonctionnel), alors le type est plat.

```
1   (** lty = [t_1 ; ... ; t_n ; return_type]  **)
2   let rec flatting_type : sort -> sort = function
3       | Arrow lty ->
4       let return_type = last_of lty in
5       match return_type with
6        | Arrow l as arrow ->
7          let append_type = flatting_type arrow in
8          let args_ty = unhook_last lty in
9          Arrow(args_ty @ append_type)
10       | _ -> Arrow lty
11      | ty -> ty
```

FIGURE 3.1 – Algorithme d'aplatissement des types

3.4 Problématiques liées à la logique d'ordre supérieur

Pour étendre un solveur SMT à la logique d'ordre supérieur, il est nécessaire d'étendre tous ses composants, à savoir : le solveur ground et le module d'instanciation. En particulier, comme expliqué en section 2.2, le solveur ground est constitué d'un solveur SAT pour gérer la structure

logique des formules, et de plusieurs modules de théorie coopérant via le principe de combinaison de théories de Nelson et Oppen. Dans notre approche nous présentons une extension à la logique d'ordre supérieur qui ne nécessite pas la modification de chaque procédure de décision. En effet, la grammaire d'ordre supérieur affecte principalement la structure des termes, et il est possible par l'usage de certaines transformations d'isoler des constructions d'ordre supérieur, notamment au niveau des symboles de théories. Nous montrons dans ce travail qu'il est suffisant d'étendre la théorie de l'égalité pour obtenir un solveur SMT capable de raisonner à l'ordre supérieur. Néanmoins, la procédure dite *redesign* devient incomplète avec l'extensionnalité. Dans notre prototype nous contenons l'ensemble du raisonnement d'ordre supérieur au sein de la fermeture de congruence, permettant par la même occasion de garder la méthode de combinaison de théories Nelson et Oppen inchangée. De plus l'ensemble des procédures (autres que la fermeture de congruence) de décision développées pour la logique du premier ordre sont réutilisées pour la logique d'ordre supérieur modulo théories. Ce travail s'articule donc autour de l'extension des trois points suivants :

1. l'élimination des λ-abstractions par pré-traitement ;
2. l'extension du solveur ground pour manipuler des expressions contenant des applications partielles de fonctions. Nous appelons cette nouvelle procédure de décision QF_HOSMT ;
3. l'extension du module d'instanciation pour les variables de type fonctionnel et la prise en compte des applications partielles et équations fonctionnelles.

Nous présenterons d'abord une approche dite pragmatique (section 3.5) utilisée en particulier pour le solveur cvc4. Cette approche permet au solveur de manipuler des formules de la logique d'ordre supérieur sans avoir recours à de modifications lourdes du solveur. Cette approche est expliquée dans la section 3.5. Elle a fait l'objet de la publication suivante [10]. La description de cette approche nous servira par la suite de point d'appui pour décrire l'approche empruntée dans le solveur veriT, décrite en détail dans la section 3.6.

Traitement des formules d'ordre supérieur

Pour s'assurer que les formules qui atteignent le cœur du solveur SMT ne contiennent aucun lieur λ, un prétraitement en deux étapes est appliqué en amont des formules. La première étape consiste à réduire toutes les applications de λ-abstraction, via le processus de β-réduction. Concrètement chaque application de la forme $(\lambda \bar{x}.\, t[\bar{x}])\ \bar{u}$ est remplacée par le terme $t[\bar{u}]$. Cette substitution renomme les variables liées dans t de sorte à ne pas capturer de variable dans les étapes ultérieures. Une fois toutes les applications de ce type réduites, une seconde passe d'élimination des lieurs λ est appliquée.

Pour éliminer les lieurs λ non appliqués, il existe plusieurs approches, dont la transformation par combinateurs SK [90], et le λ-lifting [65]. La transformation par combinateurs SK permet de synthétiser les termes contenant des applications de λ-abstractions. Cette transformation a

l'avantage d'être simple, et permet notamment aux démonstrateurs de réaliser des raisonnements d'ordre supérieur sans avoir recours à l'unification d'ordre supérieur [64], qui est un problème indécidable. Cependant, cette traduction introduit un grand nombre de quantificateurs, et entraîne souvent une perte de performance [25], notamment en raison de la taille des formules obtenues après application de la transformation. C'est pour cette raison que nous choisissons dans ce travail d'utiliser la procédure de λ-lifting pour l'élimination des λ-abstractions. Intuitivement, les λ-abstractions sont des fonctions anonymes ; le principe du λ-lifting est donc simple, puisqu'il consiste à renverser ce processus en nommant chaque λ-abstraction avec un nouveau symbole de fonction. En conséquence, chaque occurrence d'une λ-abstraction est remplacée par un nouveau symbole de fonction dans le problème, défini préalablement par le corps de la λ-abstraction. La nouvelle fonction prend comme arguments les variables liées à la λ-abstraction, et les variables libres apparaissant dans son corps. Plus précisément, chaque λ-abstraction de la forme $\lambda\bar{x}_n.\, t[\bar{x}_n, \bar{y}_m]$, de type $\bar{\tau}_n \to \tau$, où $\bar{y}_m : \bar{v}_m$ apparaît dans le terme t, est associée à l'application (possiblement partielle) $f(\bar{y}_m)$, où f est un nouveau symbole de fonction de type $\bar{v}_m \times \bar{\tau}_n \to \tau$, et la définition $\forall\bar{y}_m\bar{x}_n.\, f(\bar{y}_m, \bar{x}_n) \simeq t[\bar{x}_n, \bar{y}_m]$ est ajoutée à la formule φ. À l'ordre supérieur il est assez courant d'utiliser des λ-abstractions pour définir des fonctions simples telles que l'identité, où encore des prédicats. Ces fonctions sont souvent passées en arguments de fonctions de map sur des structures de données, et apparaissent parfois à plusieurs reprises dans un programme ou un problème logique. Ainsi pour minimiser le nombre de nouveaux symboles de fonction, et de formules quantifiées introduites par la transformation de λ-lifting, les expressions éliminées sont sauvegardées dans un cache de sorte qu'une définition puisse être réutilisée. Pour aller plus loin des optimisations permettant de déterminer l'équivalence de définitions modulo α-renommage peuvent être utilisées pour minimiser le nombre de symboles, en utilisant des indices de De Bruijn par exemple.

En présence d'une ou plusieurs théories T, il est important de normaliser les formules de façon à ce qu'aucun symbole de théorie ne contienne d'application partielle. Pour ce faire nous utilisons un prétraitement supplémentaire, transformant chaque application partielle de symboles de théories ou de symboles *interprétés*, en application totale : chaque terme de la forme $h(\bar{t}_m)$, où $h : \bar{\tau}_n \to \tau$ est un symbole de T et $m < n$, est traduit en le terme suivant $\lambda\bar{x}_{n-m}.\, h(\bar{t}_m, \bar{x}_{n-m})$, qui est par la suite λ-lifté par la procédure décrite ci-dessus en un symbole non interprété f, défini par la formule quantifiée $\forall\bar{y}\forall\bar{x}_{n-m}.\, f(\bar{x}_{n-m}) \simeq h(\bar{t}_m, \bar{x}_{n-m})$, où $\bar{y}$ réunit les variables libres de $\bar{t}_m$. Par conséquent, l'application partielle de h est traitée par l'introduction d'un symbole de fonction non interprété f, préservant les solveurs de théories de toute construction d'ordre supérieur. Pour des raisons évidentes d'efficacité ce traitement est appliqué uniquement aux symboles interprétés.

Nous insistons sur le fait qu'une ingénierie minutieuse est nécessaire pour effectuer correctement le λ-lifting dans un solveur SMT qui n'a pas été conçu à l'origine pour cela. Par exemple, la réutilisation du code utilisé pour l'élimination des constructions if-then-else (ite dans le langage SMT-LIB) n'est pas toujours possible, car elle n'est pas capable notamment de prendre en compte des constructions sous des lieurs, ou avec des termes comme application de tête. De plus l'ensemble des mécanismes de gestion de portée des variables doivent être étendus pour tous

les termes contenant des constructions de type λ-abstractions. Dans le solveur veriT un travail important a été dédié à la mise en place de structures de données dites d'ordre supérieur, basées sur des représentations binaires des termes. Ces représentations, respectent les propriétés de partage, et ont dû être interfacées avec les anciennes représentations de terme du solveur. Dans cette interface il est important de noter que l'implémentation a été menée de sorte que la partie raisonnement à l'ordre supérieur ne contamine pas la partie premier ordre du solveur avec des termes inutiles (des applications partielles par exemples).

3.5 Une extension pragmatique pour QF_HOSMT

Dans cette section nous présentons l'approche développée plus particulièrement pour le solveur cvc4, cette approche est aussi implémentée dans le solveur veriT. L'approche développée dans cette section est une version optimisée de l'encodage applicatif. Ce travail étend la procédure de décision pour l'égalité, et par conséquent le solveur ground, pour le fragment logique QF_HOSMT. L'ensemble des formules manipulées dans ce fragment ne contiennent aucune λ-abstraction (dans la section précédente), et aucun quantificateur. Dans cette section nous étudions l'extension de l'algorithme de fermeture de congruence permettant de raisonner avec des fonctions partiellement appliquées. Une extension du module d'instanciation est étudiée dans la section 3.7.

Compte tenu du fait que toutes les contraintes envoyées au solveur ground sont pré-traitées en amont, comme décrit dans la section précédente, elles ne contiennent donc ni lieur de type λ, ni symboles de théorie partiellement appliqués. Par conséquent, étendre le solveur ground revient à étendre la procédure de décision pour la théorie de l'égalité (EUF) pour traiter les applications partielles ainsi que l'extensionnalité.

La procédure de décision utilisée pour déterminer la satisfaisabilité d'un ensemble de littéraux dans la théorie de l'égalité est basée sur l'algorithme de fermeture de congruence [47, 84]. Bien que la procédure de décision soit facilement extensible à la logique d'ordre supérieur (avec des applications partielles mais sans λ-abstractions) via un codage applicatif uniforme [86], de nombreux solveurs SMT exigent que les symboles de fonction apparaissant dans les termes du premier ordre soient entièrement appliqués. Au lieu de réimplémenter le solveur pour qu'il puisse prendre en compte l'application partielle, ce que nous verrons dans la section suivante, nous utilisons ici un *codage applicatif paresseux* dans lequel seules les applications partielles nécessaires au raisonnement sont traduites par l'encodage. En comparaison, un encodage applicatif classique traduit toutes les applications partielles de chaque terme d'une formule.

Concrètement, lors de la construction des termes, toutes les applications partielles sont converties en applications totales au moyen du symbole binaire @, tandis que les termes pleinement appliqués sont conservés dans leur représentation originale. Déterminer la satisfaisabilité, dans la théorie de l'égalité EUF, d'un ensemble de contraintes E contenant des termes dans les deux représentations se fait en deux phases : si E est déterminé comme satisfaisable par la procédure

de décision classique pour le premier ordre, nous introduisons des égalités entre les termes "réguliers" (c'est-à-dire les termes pleinement appliqués sans le symbole @), et leur représentation dite applicative. Une fois les nouvelles contraintes introduites, nous revérifions la satisfaisabilité de l'ensemble des contraintes résultantes. Toutefois, il est important de garder à l'esprit que nous n'introduisons ces égalités que pour les termes réguliers qui interagissent avec des termes partiellement appliqués. En pratique, il est possible de vérifier cette interaction en s'appuyant sur le *E-graph*, de la fermeture de congruence de E construite par la procédure de décision pour la logique EUF. En particulier, un symbole de fonction apparaît dans une classe d'équivalence s'il s'agit de l'argument d'un symbole @ ou s'il apparaît dans une égalité entre des symboles de fonction, et donc dans le cadre d'une application partielle. Par conséquent, l'égalité entre les termes réguliers et leurs codages applicatifs est maintenu à l'intérieur du graphe d'égalités E, et n'affecte donc pas les autres parties de la procédure de décision.

Exemple 3.5.1. Soient $f : \tau \times \tau \to \tau$, $g, h : \tau \to \tau$ et $a : \tau$, considérons l'ensemble de contraintes $E = \{@(f, a) \simeq g,\ f(a, a) \not\simeq g(a),\ g(a) \simeq h(a)\}$. E est initialement satisfaisable. Toutefois, comme f et g sont des occurrences d'applications partielles, nous ajoutons à l'ensemble de contraintes une égalité permettant de faire correspondre ces termes d'ordre supérieur avec une application du premier ordre permise par l'utilisation du symbole @, de f, g :

$$E' = E \cup \{@(@(f, a), a) \simeq f(a, a),\ @(g, a) \simeq g(a)\}$$

Ainsi lorsque l'on cherche à déterminer à nouveau la satisfaisabilité de E', l'égalité $@(@(f, a), a) \simeq @(g, a)$ va être produite, ce qui permettra à la procédure de dériver $f(a, a) \simeq g(a)$ par transitivité, conduisant à une incohérence. Remarquons que nous n'exigeons pas d'égalité entre les termes pleinement appliqués dont les fonctions *n'apparaissent* pas dans le E-graph, et par conséquent leur traduction dans l'encodage. En particulier, l'égalité $h(a) \simeq @(h, a)$, n'est pas introduite par exemple. •

$$\frac{t \in \mathbf{T}(E)}{t \simeq t}\ \textsc{Refl} \qquad \frac{t \simeq u}{u \simeq t}\ \textsc{Sym} \qquad \frac{s \simeq t,\ t \simeq u}{s \simeq u}\ \textsc{Trans}$$

$$\frac{\bar{t}_n \simeq \bar{u}_n \quad f(\bar{t}_n),\ f(\bar{u}_n) \in \mathbf{T}(E)}{f(\bar{t}_n) \simeq f(\bar{u}_n)}\ \textsc{Cong} \qquad \frac{t \simeq u,\ t \not\simeq u}{\bot}\ \textsc{Conflict}$$

$$\frac{f(\bar{t}_n),\ f \in \mathbf{T}(E)}{f(\bar{t}_n) \simeq @(\ldots\ (@(f,\ t_1), \ldots),\ t_n)}\ \textsc{App-encode}$$

$$\frac{f \not\simeq g \qquad f,\ g : \bar{\tau}_n \to \tau \qquad n > 0}{f(sk_1,\ \ldots,\ sk_n) \not\simeq g(sk_1,\ \ldots,\ sk_n)}\ \textsc{Extensionality}$$

où $sk_1, \ldots, sk_n$ sont de nouveaux symboles de type respectif $\tau_1, \ldots, \tau_n$.

FIGURE 3.2 – Règles de dérivation pour la procédure de décision "pragmatique" EUF, pour QF_HOSMT.

La procédure décrite informellement plus haut est formalisée par les règles de dérivation données dans la figure 3.2. Les règles de dérivation agissent sur un ensemble de contraintes E. Une dérivation peut être appliquée à E si ses prémisses sont respectées. La conclusion de chaque règle ajoute soit une égalité à l'ensemble de contraintes E, soit remplace E par $\bot$ pour indiquer que l'ensemble de contraintes est non satisfaisable. L'application d'une règle est *redondante* si sa conclusion laisse l'ensemble E inchangé. Un ensemble de contraintes est *saturé* si les applications possibles des règles sont toutes redondantes. Les règles sont appliquées sur un ensemble de contraintes initial E_0 jusqu'à ce que $\bot$ soit dérivé, ou que la saturation soit atteinte.

Les règles Refl, Sym, Trans, Cong et Conflict, sont en réalité les règles standard pour la procédure de décision pour EUF, basée sur l'algorithme de fermeture de congruence. À savoir le plus petit ensemble d'équations clos sous conséquence logique dans la théorie de l'égalité d'un ensemble de contraintes initial. Néanmoins, la règle App-encode est la seule règle "ajoutée". Elle permet de mettre en lien les termes totalement appliqués avec leur représentation dans l'encodage applicatif. Il est important que cette règle ne soit appliquée qu'aux fonctions qui apparaissent en tant que sous-termes dans E. Comme mentionné ci-dessus, cela ne peut être le cas que si la fonction elle-même apparaît comme un argument d'une application, ce qui se produit lorsqu'elle est partiellement appliquée (comme argument de @ ou $\simeq$).

Pour traiter l'extensionnalité, la procédure s'appuie sur la règle Extensionality, qui est la contraposée de l'axiome d'extensionnalité. Cette approche est largement inspirée des procédures de décision utilisées pour traiter la théorie extensionnelle des tableaux [43, 112]. Intuitivement cette règle permet, si deux fonctions d'arité non nulle sont différentes dans E, de propager une contrainte supplémentaire, permettant de déduire que ces deux fonctions sont différentes sur leur domaine respectif. En particulier, la propriété d'extensionnalité est caractérisée par l'axiome $\forall \bar{x}_n.\, f(\bar{x}_n) \simeq g(\bar{x}_n) \Leftrightarrow f \simeq g$, pour toutes les fonctions f et g de type similaires. On peut donc observer que la règle Extensionality garantit bien le sens d'application de cet axiome de gauche à droite, par le biais de la Skolemisation. Le sens opposé est quant à lui garanti par la combinaison de la règle App-encode ainsi que les règles de la fermeture de congruence. Pour simplifier la présentation nous supposons que pour tous les termes de la forme $@(\dots(@(f, t_1), \dots), t_m) : \bar{\tau}_n \to \tau \in \mathbf{T}(E)$, il existe un symbole frais (c'est-à-dire nouveau) $f' : \bar{\tau}_n \to \tau$ tel que $@(\dots(@(f, t_1), \dots), t_m) \simeq f' \in E$. Par conséquent il n'est pas nécessaire de définir une autre règle d'extensionnalité pour les termes de la forme $@(\dots(@(f, t_1), \dots), t_{n_1}) \not\simeq @(\dots(@(g, u_1), \dots), u_{n_2})$.

Exemple 3.5.2. Soient les symboles de fonction $f, g : \tau \to \tau$, $a : \tau$, et l'ensemble de contraintes $E = \{f \simeq g,\ f(a) \not\simeq g(a)\}$. Cet ensemble de contraintes est initialement satisfaisable dans la théorie de l'égalité. Toutefois, puisque $f, g \in \mathbf{T}(E)$, la règle App-encode peut être appliquée deux fois. Après application de la règle les contraintes $f(a) \simeq @(f, a)$ et $g(a) \simeq @(g, a)$ sont dérivées. ce qui permet d'utiliser la règle Cong sur l'égalité $f \simeq g$ pour déduire $@(f, a) \simeq @(g, a)$, et cela conduit à une incohérence. •

Procédure de décision

Le calcul décrit ci-dessus est une procédure de décision si pour toute entrée, et pour toute stratégie de dérivation possible le calcul s'arrête soit par saturation (l'ensemble de contraintes est satisfaisable), soit en dérivant $\bot$ (l'ensemble de contraintes est non satisfaisable). Afin de montrer que ce calcul est bien une procédure de décision pour EUF, il faut s'assurer que (1) Le calcul termine (*terminaison*) sur toutes les entrées possibles (2) que pour toute entrée satisfaisable (resp. non satisfaisable) toutes les stratégies de calcul mènent à une saturation (resp. dérivent $\bot$). Ce dernier point est prouvé en deux parties : premièrement on cherche à montrer que toute entrée non satisfaisable aboutit à la dérivation de $\bot$ (*correction pour la réfutation*), deuxièmement il faut prouver que toute entrée satisfaisable implique la saturation du calcul (*correction solution*).

Proposition 3.5.1 (Terminaison)**.** Toute séquence non redondante d'application des règles est finie.

Démonstration. La fermeture de congruence n'introduit aucun nouveau terme, par conséquent toute application non redondante des règles Refl, Sym, Trans et Cong est borné par le nombre maximal de termes dans E. Le nombre d'applications de la règle App-encode est borné par le nombre de symboles de fonction dans $\mathbf{T}(E)$. Elles sont limitées par le nombre de fonctions partiellement appliquées initialement dans E, et par conséquent, par le nombre total d'application dans $\mathbf{T}(E)$, qui est également fini. Le nombre d'applications de la règle Extensionality est borné par le nombre de d'inégalités entre les symboles de fonction dans E. L'application de la règle Extensionality n'introduit jamais ce type d'inégalité, puisque toutes les inégalités introduites sont des applications totales de fonctions. Puisque seul un nombre fini de symboles de fonction peuvent être ajoutées à E, par App-encode, il existe un nombre fini d'application de la règle Extensionality, pour tout E. Donc, étant donné que les règles App-encode et Extensionality sont les uniques règles qui introduisent de nouveaux termes, et que chacune des ces règles ne peuvent être appliquées qu'un nombre fini de fois, alors l'ensemble des applications de toutes les règles du calcul est fini, et donc le calcul termine. □

Proposition 3.5.2 (Correction pour la réfutation)**.** Un ensemble de contraintes est non satisfaisable si $\bot$ peut être dérivé.

Démonstration. Étant donné que $\bot$ n'est clairement pas satisfaisable, il suffit de montrer que toutes les règles présentes dans la figure 3.2 préservent la satisfaisabilité de l'ensemble de contraintes.

Le sous-ensemble de règles Refl, Sym, Trans et Cong préservent la satisfaisabilité. Si l'on suppose que notre ensemble de contraintes E est satisfaisable et l'une de ces règles est applicable alors clairement l'ensemble E auquel on ajoute la conclusion de la règles est toujours satisfaisable. Aussi, E ne peut contenir $t \simeq u$ et $t \not\simeq u$ pour deux termes t et u, et la règle Conflict n'est donc pas applicable.

La règle App-encode, simule l'encodage applicatif, en effet si l'on sature l'application de

cette règle sur notre ensemble de contraintes initialement alors le problème obtenu est équivalent au problème initial dans fragment applicatif. De plus le symbole @ n'apparaît pas dans les formules initiale écrites par l'utilisateur, et seule la procédure identifie les termes $f(a)$ et $@(f, a)$ (pour tout terme présent dans E). Or l'encodage applicatif préserve la satisfaisabilité et par conséquent l'application de la règle ne change donc rien à l'interprétation de l'ensemble des contraintes.

La règle EXTENSIONALITY exprime la contraposée de l'axiome d'extensionnalité. Elle combine l'introduction d'une formule existentielle avec la Skolemisation de cette dernière et préserve par conséquent la satisfaisabilité de l'ensemble des contraintes. □

Proposition 3.5.3 (Correction solution)**.** Tout ensemble de contraintes saturé est satisfaisable.

Esquisse de preuve. La preuve repose sur la construction d'une interprétation spécifique $\mathcal{I}$ générée par les termes à partir de l'ensemble saturé de littéraux d'égalités contenue dans E_0. □

En pratique, même si nous pouvons appliquer les règles dans n'importe quel ordre, pour de meilleures performances, nous n'appliquons APP-ENCODE et EXTENSIONALITY uniquement lorsqu'aucune autre règle ne peut-être appliquée. En particulier, APP-ENCODE a la priorité sur EXTENSIONALITY.

Nous avons vu dans cette section essentiellement le travail qui a été mené dans le solveur cvc4, pour étendre ses capacités de raisonnement à la logique d'ordre supérieur. Cette approche est appelée approche pragmatique car elle offre un compromis intéressant entre l'effort de travail nécessaire à son implémentation dans un solver SMT avec une architecture complexe tel que cvc4 (quelques centaines de milliers de lignes de codes), et une méthode naïve optimisée basée sur un encodage applicatif des termes de la logique d'ordre supérieur. Dans la section suivante nous étudions ce qui a été implémenté dans le solveur SMT veriT. Cette approche est appelée *redesign* car elle implique le développement d'une nouvelle catégorie de solveur SMT capable de traiter nativement, c'est-à-dire sans faire appel à une traduction applicative, les termes de la logique d'ordre supérieur. À noter que pour effectuer des comparaisons, l'approche présentée dans la section qui se termine ici à aussi été implémentée dans veriT, avec un niveau d'optimisation et d'heuristique plus faible cependant que dans le solveur cvc4.

3.6 Un solveur SMT pour la logique d'ordre supérieur

3.6.1 Repenser l'architecture SMT pour QF_HOSMT

Pour l'approche *redesign*, nous reprenons les mêmes hypothèses sur les formules d'entrée, à savoir les formules sont sans λ-abstractions et sans applications partielles de symboles théoriques, notamment grâce aux méthodes présentées dans la section 3.4. La procédure de décision pour

QF_UF implémentée dans le solveur veriT est essentiellement basée sur l'approche de fermeture de congruence développée par Nieuwenhuis et Oliveras [85], qui permet d'une part de traiter de manière efficace, en temps ($\mathcal{O}(n \log n)$), le raisonnement équationnel, et d'autre part s'intègre de façon optimale dans les solveurs SMT. En effet, l'approche proposée par Nieuwenhuis et Oliveras est incrémentale, autorise la construction de modèles, d'explications de conflits et des preuves. De plus l'approche proposée ne pénalise pas le solveur puisque sa complexité algorithmique reste $\mathcal{O}(n \log n)$. Comme expliqué dans la section 2.4, pour être intégré efficacement dans un solveur SMT une procédure de décision doit impérativement être incrémentale c'est-à-dire qu'elle doit être capable de fonctionner à flux tendu. Il faut à tout moment avoir une vue sur la satisfaisabilité de l'ensemble de contraintes, et en cas de conflit être capable de fournir des explications. Ces deux exigences ajoutent un défi supplémentaire à notre quête, celui d'étendre la procédure pour lui ajouter ces fonctionnalités tout en préservant une complexité acceptable. De plus, l'implémentation de ces algorithmes est complexe et peu flexible. Enfin, les structures de données sont complètement imbriquées entre elles et sont difficiles à modifier.

Tenter de généraliser une telle architecture pour le raisonnement d'ordre supérieur, en essayant à tout prix de maintenir la complexité optimale en temps et en espace, demanderait un travail important, et difficile. Par conséquent notre choix d'implémentation s'est porté vers une nouvelle structure de données avec des coûts de complexité plus grande, au bénéfice d'une plus grande souplesse et modularité de la procédure. En effet la nouvelle architecture permet l'ajout de nouvelles fonctionnalités telles que le traitement des fonctions injectives, l'ajout de termes "à la volée", la réécriture, le traitement natif des transformations β et η spécifiques au λ-calcul, ou encore la possibilité d'étendre la procédure à d'autres théories tels que les data-types. L'algorithme proposé est d'une complexité quadratique dans le pire des cas, ce qui est moins bon que la complexité $\mathcal{O}(n \log n)$ précédente, mais toujours polynomial, et nous verrons, en pratique totalement acceptable. De plus cette procédure permet une meilleure intégration avec l'algorithme CCFV [9]. Une version de l'algorithme CCFV utilisant cette version de la fermeture de congruence a notamment été implémentée, et est disponible à l'adresse suivante [1]. L'implémentation de la procédure est décrite dans le chapitre 4.

L'idée générale de notre algorithme de fermeture de la congruence est d'utiliser une généralisation du concept de graphe d'égalité comme dans [53] à la place du traditionnel union-find, pour calculer les classes d'équivalences. Dans notre représentation, les nœuds du graphe sont les termes, et les arêtes sont les relations entre ces termes. On distingue chaque relation par différents types d'arêtes. Dans le cas simple (sans fonctionnalité supplémentaire), seuls trois types d'arêtes sont nécessaires : les égalités, les congruences et les inégalités. Une classe d'équivalence est alors représentée par un graphe fortement connexe, c'est à dire que tous les nœuds sont liés entre eux par des arêtes qui ne sont pas des arêtes de diségalité. En d'autres termes, deux termes sont égaux si et seulement s'il existe un chemin sans arête de diségalité entre eux. En outre, pour déterminer toute forme d'incohérence entre des prédicats, chaque classe d'équivalence est initialisée avec une polarité neutre au début. Pour savoir si un ensemble conjonctif de littéraux est satisfaisant, il suffit de construire progressivement le graphe comme suit : étant donné les

1. `https://github.com/delouraoui/these`

termes t, t', w, w', p, si le littéral l est un littéral positif de la forme $t \simeq t'$ ajouter une arête entre t et t', puis déduire tous les congruences entre les termes $w \in Pred(t)$ et $w' \in Pred(t')$ ($Pred$ retourne tous les parents des termes égaux à t). Quand deux termes w et w' sont congruents on ajoute une arête (de congruence) entre w et w'. Lorsque l est un littéral négatif de la forme $t \not\simeq t'$ on ajoute une arête d'inégalité entre t et t'. Enfin, si l est un prédicat p, alors nous donnons simplement une polarité à la classe d'équivalence de p (remarque, les arêtes de congruence ne propagent la polarité que lorsque la polarité de l'une des deux classes n'est pas définie). Pour détecter toute forme d'incohérence, après chaque nouvelle assertion d'un littéral par le solveur SAT, il suffit que les deux propriétés ci-dessous soient verifiées :

1. si deux termes sont égaux et en même temps reliés par une arête de diségalité, alors l'ensemble des littéraux qui implique le graphe est incohérent.
2. si une contrainte implique l'union de deux classes d'équivalence avec deux polarités définies et différentes, alors l'ensemble des littéraux qui implique le graph est incohérent.

Par conséquent, ces propriétés permettent d'utiliser l'algorithme de façon incrémentale (de telle sorte que nous n'avons pas besoin de voir tous les littéraux pour déterminer la satisfaisabilité). Cette structure de données est naturellement traçable, il suffit d'ajouter et de supprimer des arêtes. Les explications sont également faciles à obtenir puisque d'après les propriétés précédentes il suffit de donner un chemin entre les deux termes en conflit au solveur SAT pour avoir une séquence de littéraux qui impliquent le conflit.

Enfin, nous développons dans ce chapitre une représentation spécifique des termes pour gérer les termes d'ordre supérieur. Pour une meilleure compréhension, nous utilisons une représentation d'ordre supérieur des termes, qui consiste à transformer les termes déjà présents dans le solveur veriT en une représentation binaire qui respecte la propriété de partage des DAG de veriT. Par exemple, le terme de premier ordre $f(a, g(b, c))$ sera représenté dans la nouvelle fermeture de congruence par le terme suivant $((f\,a)\,((g\,b)\,c))$. Il faut noter que les sous-termes $(f\,a)$ et $(g\,b)$ seront générés de manière interne à la fermeture de congruence de sorte à ne pas polluer l'environnement du solveur, et f et g serons considérés au sein du nouvel algorithme comme des symboles constants. Dans le but de préserver les propriétés de partage, pour garder une représentation mémoire optimale, la structure des termes est équipée d'un index. L'ensemble de ses fonctionnalités est décrit en détail dans le chapitre implémentation 4. Notablement, cette représentation a deux avantages, le premier est que la relation de congruence entre deux termes est simplifiée puisqu'il suffit de tester l'équivalence des têtes et des uniques arguments. Deuxièmement c'est une représentation totalement adaptée à la logique d'ordre supérieur qui est appliqué localement, et qui ne pollue pas le solveur avec des sous-termes inutiles pour les autres procédures de décision. En théorie cet algorithme dispose d'une complexité quadratique, car on parcourt systématiquement les deux classes équivalentes pour construire la nouvelle classe qui contient l'union des deux classes. Cependant en pratique ce dernier se comporte bien, c'est-à-dire que les performances du solveur ne sont pas dramatiquement dégradées. Dans la section 3.8 nous apportons des arguments de comparaison avec le solveur SMT cvc4 sur le fragment de logique de la théorie de l'égalité.

3.6.2 Étendre le solveur ground pour la logique d'ordre supérieur

Dans l'approche présentée ici, le solveur conserve deux représentations des termes : une représentation dite curryfiée (les fonctions sont monadiques), et une représentation non curryfiée (les fonctions sont d'arité quelconque). Dans la représentation non curryfiée, les applications partielles et totales peuvent être distinguées grâce au type. La représentation curryfiéee n'est utilisée que par l'algorithme de fermeture de la congruence. Les deux représentations cohabitent au sein du solveur grâce à une interface qui se traduit par les fonctions `curry` qui permet de passer d'une représentation non curryfiée à une représentation curryfiée, et `uncurry` qui permet d'effectuer la transformation inverse. En particulier, dans cette section nous distinguons, pour être précis, la notation applicative de la notation curryfiée. La différence est subtile entre les deux notations : la notation $@(\dots (@(f,\, t_1), \dots),\, t_n)$ décrit l'application successive du symbole de fonction @, qui applique une fonction à son argument ; alors que la notation $(\dots ((f,\, t_1), \dots),\, t_n)$ décrit une construction de langage, cette expression exprime l'application successive de f à t_1 puis de $(f\ t_1)$ à t_2, etc. L'application est donc dans ce dernier cas de figure un constructeur, et non un symbole de fonction. Nous choisissons cette présentation puisque dans cette approche dite *redesign* il n'est plus nécessaire d'introduire de symbole supplémentaire, l'application est traitée en interne par la représentation curryfiée via une structure de donnée repensée. Formellement, si l'on applique la fonction à un terme en représentation non curryfiée on obtient le résultat suivant : $\texttt{curry}(f(\bar{t}_n)) = (\dots ((f,\, t_1), \dots),\, t_n)$. Pour retrouver la forme non curryfiée, à partir de la forme curryfiée il suffit d'appliquer la fonction comme suit : $\texttt{uncurry}((\dots ((f,\, t_1), \dots),\, t_n)) = f(\bar{t}_n)$. Les détails sur l'implémentation sont donnés dans le chapitre 4.

Exemple 3.6.1. Soient $f : \tau \times \tau \to \tau$, $g, h : \tau \to \tau$ et $a : \tau$, considérons l'ensemble de contraintes $\{f(a) \simeq g,\ f(a,a) \not\simeq g(a),\ g(a) \simeq h(a)\}$. Notre algorithme de fermeture de congruence raisonne à partir de l'ensemble de contraintes curryfiées $\{(f,\, a) \simeq g,\ ((f,\, a),\, a) \not\simeq (g,\, a),\ (g,\, a) \simeq (h,\, a)\}$. •

L'ensemble des contraintes curryfiées est donc formellement défini comme suit. Étant donné un ensemble E de contraintes non curryfiées, l'ensemble des contraintes curryfiées E_{curr} s'exprime comme l'ensemble de contraintes suivant

$$E_{\text{curr}} = \{\texttt{curry}(f(\bar{t}_n)) \simeq \texttt{curry}(g(\bar{s}_m)) \mid f(\bar{t}_n) \simeq g(\bar{s}_m) \in E\}$$

On note $\mathbf{T}(E_{\text{curr}})$, l'ensemble de tous les sous-termes de l'ensemble de contraintes curryfiées E_{curr}. Pour simplifier la présentation donnée ci-dessus on suppose que l'ensemble E_{curr}, contient aussi l'ensemble des inégalités $\{\texttt{curry}(f(\bar{t}_n)) \not\simeq \texttt{curry}(g(\bar{s}_m)) \mid f(\bar{t}_n) \not\simeq g(\bar{s}_m) \in E\}$.

Exemple 3.6.2. Considérons l'ensemble de contraintes $E = \{f(a, g(b,c)) \simeq a\}$, avec $a, b, c : \tau$ et $f, g : \tau \times \tau \to \tau$, l'ensemble de contraintes curryfiées E_{curr} est

$$\begin{aligned} E_{\text{curr}} &= \texttt{curry}(f(a, g(b,c))) \simeq \texttt{curry}(a) \\ &= ((f,\, a),\, ((g,\, b),\, c)) \simeq a \end{aligned}$$

Et l'ensemble des sous-termes curryfiés est le suivant :

$$\mathbf{T}(E_{\text{curr}}) = \{((f,\, a),\, ((g,\, b),\, c)), (f,\, a), (g,\, b), ((g,\, b),\, c), f, g, a, b, c\}$$

Notons que l'ensemble $\mathbf{T}(E_{\text{curr}})$ est calculé à partir de E_{curr}. •

$$\frac{t \in \mathbf{T}(E_{\text{curr}})}{t \simeq t}\ \text{Refl}_{\text{curr}} \qquad \frac{t \simeq u}{u \simeq t}\ \text{Sym}_{\text{curr}} \qquad \frac{s \simeq t,\, t \simeq u}{s \simeq u}\ \text{Trans}_{\text{curr}}$$

$$\frac{t \simeq u \quad t' \simeq u' \quad (t,\, t'), (u,\, u') \in \mathbf{T}(E_{\text{curr}})}{(t,\, t') \simeq (u,\, u')}\ \text{Cong}_{\text{curr}} \qquad \frac{t \simeq u,\, t \not\simeq u}{\bot}\ \text{Conflict}$$

$$\frac{t,\, u \in \mathbf{T}(E_{\text{curr}}) \qquad t,\, u : \bar{\tau}_n \to \tau,\, n > 0}{\forall F, G : \bar{\tau}_n \to \tau.\, F \not\simeq G \Rightarrow F(sk_1,\, \ldots,\, sk_n) \not\simeq G(sk_1,\, \ldots,\, sk_n)}\ \text{Ext-Ax}$$

où $sk_1, \ldots, sk_n$ sont de nouveaux symboles de type respectif $\tau_1, \ldots, \tau_n$.

FIGURE 3.3 – Règles curryfiées pour EUF.

Les règles de la figure 3.2 sont modifiées pour raisonner à partir de termes curryfiés. La figure 3.3 exprime les règles de calcul pour la fermeture de congruence où l'ensemble des contraintes sont curryfiées. En particulier, on remarque que la règle APP-ENCODE n'est plus nécessaire dans cette version du calcul, puisque toutes les déductions peuvent être réalisées pareillement dans sa version curryfiée. On suppose que tous les symboles de théorie on été pré-traités, comme vu dans la section 3.4, de sorte qu'il n'y a pas d'application partielle de ces symboles. La combinaison de théories dans veriT s'appuie sur l'échange d'égalités. En principe un solveur de théorie peut déduire des égalités qui doivent être partagées avec la fermeture de congruence, et inversement. Néanmoins, il est possible qu'au sein de la fermeture de congruence certains termes partagés avec d'autres théories apparaissent comme partiellement appliqués. Dans ce cas de figure précis il est important de ne pas propager ces sous-termes dans les autres solveurs de théorie. Pour cette raison le module de fermeture de congruence ne propage aucune contrainte contenant des sous-termes partiellement appliqués, mais seulement les contraintes contenant des termes partagés et déjà présents dans les autres théories, en s'appuyant encore une fois sur un système d'interface (`curry`, `uncurry`). Cette restriction permet de garantir le bon fonctionnement du processus de combinaison de théorie. Dans l'exemple ci-dessous on présente un cas concret d'interaction avec la théorie de l'arithmétique où la fermeture de congruence, manipule des contraintes sur des termes partagés.

Exemple 3.6.3. Soient f : `Int`→`Int` , p :`Int`→ $o, a, b, c_1, c_2, c_3, c_4$:`Int`, l'ensemble de contraintes $E = \{a \leq b,\, b \leq a,\, p(f(a,c) - f(b,c)),\, \neg p(0),\, c_1 \simeq c_3 - c_4,\, c_2 \simeq 0, c_3 \simeq f(a,c),\, c_4 \simeq f(b,c)\}$. L'ensemble de contraintes curryfié envoyé à la fermeture de congruence après abstraction des termes liées à la théorie de l'arithmétique est $E_{\text{curr}} = \{(p,\, c_1),\, \neg(p,\, c_2),\, c_3 \simeq ((f,\, a),\, c),\, c_4 \simeq ((f,\, b),\, c)\}$. Les égalités $c_3 \simeq ((f,\, a),\, c)$ et $c_4 \simeq ((f\, b),\, c)$ gardent une trace des termes partagés $((f,\, a),\, c)$ et $((f,\, b),\, c)$. A partir de l'ensemble de contraintes E la procédure de décision pour l'arithmétique peut déduire que $a \simeq b$. Le nouvel ensemble de contraintes est donc $E' = E \cup \{a \simeq b\}$. A partir de E'_{curr} la fermeture de congruence déduit par congruence que $(f,\, a) \simeq (f,\, b)$, ce qui implique que $((f,\, a),\, c) \simeq ((f,\, b),\, c)$. On voit donc que la première contrainte ne doit pas nécessairement être partagée avec le module d'arithmétique, cependant la

deuxième est importante car elle rend deux termes partagés égaux. Il faut donc propager l'égalité $\texttt{uncurry}(((f,\, a),\, c)) \simeq \texttt{uncurry}(((f,\, b),\, c))$. Grâce à cette nouvelle contrainte la procédure de décision pour l'arithmétique est capable de déduire que $c_1 \simeq c_2$, ce qui a pour conséquence de rendre l'ensemble de contraintes $E' \cup \{\texttt{uncurry}(((f,\, a),\, c)) \simeq \texttt{uncurry}(((f,\, b),\, c)), c_1 \simeq c_2\}$ non satisfaisable. •

Extensionnalité. Contrairement, à la règle d'extensionnalité définie dans la figure 3.2, dans veriT nous avons choisi d'utiliser une version quantifiée de l'axiome. La règle EXT-AX donnée figure 3.3, permet de dériver pour chaque terme de type fonctionnel une formule quantifiée, qui peut par la suite être instanciée avec tous les termes du bon type contenus dans $\mathbf{T}(E_{\text{curr}})$. Cette règle est suffisante pour assurer la propriété d'extensionnalité au niveau du solveur ground. Cependant, elle a l'inconvénient de recourir à l'instanciation, ce qui fait perdre au calcul sa propriété de terminaison. Aussi surprenant que cela puisse paraître cela fonctionne mieux en pratique dans veriT pour les problèmes qui nous intéressent (c.-à-d. majoritairement provenant d'assistants de preuve) que la version proposée plus tôt dans l'approche pragmatique (figure 3.2). La raison est donc assez simple : en pratique la majorité des problèmes sur lesquels nous avons évalué notre approche contiennent des quantificateurs. Introduire des formules quantifiés dans un solveur SMT présente toujours des risques, en particulier de terminaison, cependant comme le montre l'exemple 3.6.4, utiliser un axiome peut parfois être plus avantageux. Dans veriT les deux stratégies d'axiomes sont implémentées et peuvent être utilisées indépendamment. Le calcul présenté dans cette section est correct et préserve les propriétés de correction 3.5.2 et 3.5.3. La démonstration de ces propriétés peut se faire en deux parties. Une première partie consiste à prouver par simulation que, sous l'hypothèse que la version pragmatique du calcul est correcte, le calcul 3.3 permet d'obtenir les même solutions sans l'extensionnalité, que le calcul défini dans la figure 3.2. Puis dans un second temps montrer qu'en ajoutant la règle EXT-AX, on ne perd pas la correction, mais seulement la complétude.

Exemple 3.6.4. Considérons l'ensemble de contraintes $E = \{h(f) \simeq b,\, h(g) \not\simeq b,\, \forall x.\, f(x) \simeq a,\, \forall x.\, g(x) \simeq a\}$, avec $h : \tau \to \tau \to \tau$, $f, g : \tau \to \tau$, $a, b : \tau$. L'approche pragmatique pourrait théoriquement résoudre ce problème en utilisant la procédure de décision ground présentée dans la section précédente (Figure 3.2), en utilisant la règle d'extensionnalité. Néanmoins, les solveurs SMT sont bien connus pour ne pas propager toutes les inégalités (principalement par soucie d'efficacité). Par conséquent la procédure présentée avant pourrait en pratique ne pas dériver $f \not\simeq g$ qui permettrait d'appliquer la règle d'extensionnalité, pour déduire $f(sk) \not\simeq g(sk)$, qui permettrait ainsi de conclure que le problème original est non satisfaisable. En revanche avec l'approche présentée ici (Figure 3.3), la formule $\forall F, G : \tau \to \tau.\, F \not\simeq G \Rightarrow F(sk) \not\simeq G(sk)$ est automatiquement dérivé puisque les termes $f : \tau \to \tau$ et de $g : \tau \to \tau$, font partis du problème ground. Par conséquent après instantiation de cette formule la clause ajoutée au ground solveur est la suivante $f \simeq g \vee f(sk) \not\simeq g(sk)$, qui correspond à un case split que le SAT solveur traitera en deux branches. La première $E \cup \{f \simeq g\}$ qui permet de reporter un conflit au niveau équationnel. Tandis que la seconde $E \cup \{f(sk) \not\simeq g(sk)\}$ permet au module d'instanciation de dériver l'instance $f(sk) \simeq a$, $g(sk) \simeq a$ rendant le problème E non satisfaisable au niveau propositionnel. •

Proposition 3.6.1 (Équivalence)**.** Les deux systèmes de dérivation sont équivalents, sans l'ex-

tensionnalité :

Démonstration. Par induction structurelle sur les arbres de dérivation.
Si la dernière règle appliquée est $\textsc{Refl}_{\textsc{curr}}$, on montre que $\textsc{Refl}_{\textsc{curr}} \Rightarrow \textsc{Refl}$: si $t \in \mathbf{T}(E_{\text{curr}})$ alors il existe une injection `uncurry` de $\mathbf{T}(E_{\text{curr}})$ à $\mathbf{T}(E)$ telle que $\texttt{uncurry}(t) \in \mathbf{T}(E)$. D'où $\texttt{uncurry}(t) \simeq \texttt{uncurry}(t)$. Les cas $\textsc{Sym}_{\textsc{curr}}$ et $\textsc{Trans}_{\textsc{curr}}$ sont similaires.

Si la dernière règle appliquée est $\textsc{Cong}_{\textsc{curr}}$ on doit montrer que $\textsc{Cong}_{\textsc{curr}} \Rightarrow \textsc{Cong}$. Pour ce cas, il faut démontrer qu'il est possible de simuler une application de la règle Cong grâce à une ou plusieurs applications de la règle $\textsc{Cong}_{\textsc{curr}}$. Intuitivement, si deux termes $f(\bar{t}_n)$, et $f(\bar{u}_n)$ sont congruents par l'application de Cong, cela signifie que $\bar{t}_n \simeq \bar{u}_n$ et donc par conséquent que $(f,\ t_1) \simeq (f,\ u_1), ((f,\ t_1),\ t_2) \simeq ((f,\ u_1),\ u_2), \ldots, ((\ldots(f,\ t_1),\ t_2), \ldots),\ t_n) \simeq ((\ldots(f,\ u_1),\ u_2),\ \ldots),\ u_n)$. En partant de cette observation, il suffit donc d'appliquer n fois (n le nombre d'arguments) la règle $\textsc{Cong}_{\textsc{curr}}$ pour simuler une application de Cong.

Formellement, on définit $\phi_k(f(\bar{t}_n))$, la fonction qui applique dans le terme $f(\bar{t}_n)$ les k premiers sous-termes à gauche, avec $k \leq n$. Il existe donc un séquence de dérivations successives, dont les n derniers règles sont n applications de la règle $\textsc{Cong}_{\textsc{curr}}$ avec :

$$\dfrac{\dfrac{\Pi_{n-2}}{\phi_{n-1}(f(\bar{t}_n)) \simeq \phi_{n-1}(f(\bar{u}_n))}\,\textsc{Cong}_{\textsc{curr}} \qquad \phi_{k_n}(t_n) \simeq \phi_{l_n}(u_n)}{\phi_n(f(\bar{t}_n)) \simeq \phi_n(f(\bar{u}_n))}\,\textsc{Cong}_{\textsc{curr}}$$

où Π_{n-2} est le sous-arbre

$$\dfrac{\dfrac{\phi_0(f(\bar{t}_n)) \simeq \phi_0(f(\bar{u}_n)) \quad \phi_{k_1}(t_1) \simeq \phi_{l_1}(u_1)}{\phi_1(f(\bar{t}_n)) \simeq \phi_1(f(\bar{u}_n))}\,\textsc{Cong}_{\textsc{curr}} \qquad \phi_{k_2}(t_2) \simeq \phi_{l_2}(u_2)}{\ldots}\,\textsc{Cong}_{\textsc{curr}}$$

De sorte que $\overline{\phi_{k_n}(t_n)} = \bar{t}_n$ et $\overline{\phi_{l_n}(u_n)} = \bar{u}_n$ on fixe $t = \phi_{n-1}(f(\bar{t}_n))$, $u = \phi_{n-1}(f(\bar{u}_n))$, $t' = \phi_{k_n}(t_n)$, $u' = \phi_{l_n}(u_n)$ alors $(t\ t') = \phi_n(f(\bar{t}_n))$ et $(u\ u') = \phi_n(f(\bar{u}_n))$ comme le montre la dérivation ci-dessus. Par hypothèse d'induction, on a que $t \simeq t'$ et que $u \simeq u'$, on a alors par application de la règle $\textsc{Cong}_{\textsc{curr}}$ que $(t\,t') \simeq (u\,u')$, or $(t\,t') = \phi_n(f(\bar{t}_n))$ et $(u\,u') = \phi_n(f(\bar{u}_n))$ donc $f(\bar{t}_n) \simeq f(\bar{u}_n)$ et par conséquent $\textsc{Cong}_{\textsc{curr}} \Rightarrow \textsc{Cong}$.
Tous les autres cas sont simples.
$\Leftarrow$ Trivial en utilisant la traduction `curry`. □

La correction du calcul 3.3 avec l'axiome d'extensionnalité est un peu plus subtile. En effet l'ajout de l'axiome d'extensionnalité ne rend pas le calcul "unsound", puisque c'est une tautologie. Cependant, les proposition 3.5.2 et 3.5.3, établissent la correction, et la complétude du calcul 3.2. Avec la règle d'extensionnalité le calcul perd la terminaison pour les contraintes satisfaisables, et donc la complétude (dans le cas satisfaisable). Pour prouver la correction du calcul il suffit donc de relâcher la condition de saturation de la proposition 3.5.3.

Pour la proposition 3.5.2, la preuve doit se baser sur l'hypothèse que le module d'instanciation est complet pour la réfutation (ce qui est le cas pour veriT). En d'autres termes si le problème d'entrée est non satisfaisable alors veriT termine toujours, et répond UNSAT (en théorie avec un temps de calcul non limité). Par conséquent en s'appuyant sur cette hypothèse le calcul 3.3, respecte la proposition 3.5.2. On a donc une procédure de semi-décision qui est correct.

Dans cette section nous avons présenté une semi-procédure de décision permettant de raisonner de manière native sur des termes de la logique d'ordre supérieur. Nous avons expliqué comment ce calcul permettait de traiter les problématiques liées à la combinaison de théories ainsi que celles liées à l'extensionnalité. Nous avons terminé cette sous-section en montrant la correction de ce calcul par simulation et détaillant les subtilités liée à l'extensionnalité.

3.7 Étendre le module d'instanciation pour HOSMT

Les principales techniques utilisées pour résoudre le problème de l'instanciation des quantificateurs dans SMT sont l'instantiation par triggers [45], par conflits [9, 97], par construction de modèle [55, 98], et énumérative [95]. L'extension de ces approches représente un défi, en particulier, lorsqu'il est nécessaire d'y intégrer un algorithme d'unification d'ordre supérieur. Le problème de *matching* est une variante, plus facile, du problème d'unification. Plus précisément on parle de matching lorsque un des termes à unifier est ground. Le problème dit de *E-matching* [41] consiste donc à trouver une substitution σ telle qu'étant donné un terme ground u, et un terme contenant des variables libres t, $E \models u \simeq t\sigma$. La solution à ce problème d'unification est une substitution qui peut être utilisée pour instancier la formule qui contient t.

Dans ce travail nous nous concentrons essentiellement sur l'algorithme de E-matching, qui est l'élément central utilisé pour l'instanciation par triggers, et qui est, de surcroît, la technique la plus couramment utilisée dans les solveurs SMT, pour l'instanciation des quantificateurs. Intuitivement, cette technique choisit des instances pour chaque formule quantifiée φ à partir de sous-termes extraits de φ, appelés *triggers*. Ce trigger est un terme (ou un ensemble de termes) contenant les variables libres apparaissant dans φ. Pour déterminer une instance, cette approche met en correspondance un trigger avec des termes ground qui apparaissent dans l'ensemble de contraintes E.

La présence de contraintes d'ordre supérieur pose certains défis pour l'algorithme d'E-matching. Dans le cas de l'approche pragmatique, ou lorsque l'on souhaite tout simplement utiliser une approche par encodage applicatif, nous avons tout d'abord à considérer que le symbole @ est un opérateur surchargé au niveau de ses arguments. Par conséquent, les applications de ce symbole peuvent être sélectionnées en tant que termes apparaissant dans les triggers. Il convient donc de veiller tout particulièrement à ce que les applications de @ ne soient pas associées à des applications ground de @ dont les arguments ont des types différents. Deuxièmement, les symboles fonctions peuvent être mis en équation dans la logique d'ordre supérieur. Par conséquent, une correspondance entre deux termes peut impliquer un terme de trigger et un terme ground avec

différents symboles de tête. Ce qui n'est pas le cas au premier ordre puisque pour être unifiable, deux termes qui sont des fonctions doivent avoir le même symbole de tête. Troisièmement, notre ensemble de termes ground peut contenir un mélange d'applications de fonctions partiellement et entièrement appliquées. L'exemple suivant illustre les deux derniers défis.

Exemple 3.7.1. Soit E contenant l'égalité $@(f, a) \simeq g$ et le terme $f(a, b)$ où $f : \tau \times \tau \to \tau$ et $g : \tau \to \tau$. Supposons que le trigger $g(x)$ apparaisse dans notre problème initial. D'après l'égalité $@(f, a) \simeq g$ on peut déduire que $g(x)$ est équivalent modulo E au terme $f(a, b)$ sous la substitution $x \mapsto b$. Une telle correspondance est trouvée en indexant tous les termes qui sont des applications soit de $@(f, a)$, soit de g, en commun dans l'index des termes. Cela permet de garantir, lors de la mise en correspondance de $g(x)$, que le terme $f(a, b)$, où sa forme applicative est $@(@(f, a), b)$, est considéré. •

Dans l'approche pragmatique l'algorithme de E-matching est adapté dans le solveur cvc4 de sorte à prendre en considération les cas vus ci-dessus. Les extensions d'autres approches d'instanciation, telles que l'approche par construction de modèle, sont laissées en tant que travaux futurs.

Dans l'approche *redesign* nous proposons une extension de l'algorithme de E-matching permettant de répondre aux problématiques posées par la syntaxe de la logique d'ordre supérieur à savoir : la gestion des variables fonctionnelles, et du mélange d'applications partielles et totales. Dans veriT la représentation curryfiée des termes n'est utilisée qu'à l'intérieur du E-graph de la nouvelle procédure de fermeture de congruence. Pour que l'algorithme de E-matching fonctionne efficacement, il est impératif de disposer d'une bonne structure d'indexation des termes, permettant un accès rapide aux termes ground présents dans E.

Pour choisir l'ensemble des *candidats*, les termes grounds, à mettre en correspondance avec un trigger donné, le module d'instanciation s'appuie sur un *index*. L'index est une structure de données généralement construite avant l'instanciation qui permet d'optimiser les temps d'accès aux potentiels candidats. Il existe plusieurs approches pour indexer des termes : les arbres discriminants [78], utilisés par exemple dans le cadre de la superposition, et les arbres code [41], utilisés par les solveurs SMT (technique développé spécifiquement pour le E-matching dans z3). Dans le solveur veriT, la structure d'index repose sur une approche artisanale, spécifiquement développée pour le framework CCFV. L'ensemble des candidats est indexé par la *signature* des termes dans la fermeture de congruence. La signature d'une fonction $f(t_1, \ldots, t_n)$, correspond à une paire contenant le symbole de tête de la fonction et la liste $([t_1], \ldots, [t_n])$ de toutes les classes d'équivalence de ses arguments. L'implémentation détaillée de cette approche peut être consultée dans la thèse de Haniel Barbosa [6].

Malheureusement à l'ordre supérieur la distinction entre symbole de tête et argument n'est plus explicite, étant donné que chaque application est binaire et qu'une fonction peut être un sous-terme et pas seulement un symbole. Par conséquent la signature telle que définie plus haut ne peut plus être utilisée pour indexer des termes curryfiés de manière efficace. Deux solutions peuvent être proposées à ce problème : utiliser deux structures d'index, une efficace, pour les

termes de premier ordre et une seconde plus coûteuse pour les constructions d'ordre supérieur. Une deuxième solution consiste à repenser toute la conception du module d'instanciation de sorte à contourner ce problème. Cette dernière solution n'est pas présentée dans cette thèse car elle est encore en étude. La solution proposée consiste à encoder le problème dans SAT [116]. Dans cette thèse nous présentons la première solution, qui représente un bon compromis entre efficacité et difficulté d'implémentation. L'idée est la suivante : étendre l'algorithme de E-matching dans une version non curryfiée, ce qui permet de conserver la structure d'index premier ordre pour toutes les contraintes qui ne contiennent pas de variables fonctionnelles (variable apparaissant en tête d'un trigger). Pour traiter les triggers dont la tête est une variable fonctionnelle, on utilise une table d'association qui, pour chaque type fonctionnel, associe un ensemble de termes grounds de E, du même type. Pour limiter la taille de cette table, seuls les représentants des classes d'équivalence sont introduits. Formellement, à partir d'un ensemble de triggers $\{u_1, \ldots, , u_n\}$, nous construisons l'ensemble de candidats suivants où $\mathtt{Ty}(t)$ renvoie le type de t :

$$\mathbf{T}^{\mathtt{uncurry}}(E_{\text{curr}}) = \left\{ \mathtt{uncurry}(t) \mid t \in \mathbf{T}(E_{\text{curr}}) \wedge \bigvee_{i=0}^{n} \mathtt{Ty}(u_i) = \mathtt{Ty}(t) \right\}$$

Exemple 3.7.2. Considérons l'ensemble de contraintes $E = \{f(a, g(b, c)) \simeq a, \forall F.\, F(a) \simeq h, \forall y.\, h(y) \not\simeq a\}$, et l'ensemble de triggers $T = \{F(a), h(y)\}$ avec $a, b, c : \tau$, $h : \tau \to \tau$ et $f, g : \tau \times \tau \to \tau$, l'ensemble des sous-termes curryfiés comme défini dans la section 3.6.2 nous permet de calculer l'ensemble de sous-termes suivant :

$$\mathbf{T}(E_{\text{curr}}) = \{((f,\, a),\, ((g,\, b),\, c)), (f,\, a), (g,\, b), ((g,\, b),\, c), f, g, a, b, c\}$$

A partir de l'ensemble obtenu ci-dessus, nous calculons l'ensemble des candidats potentiels pour les triggers $F(a)$ et $h(y)$ comme suit

$$\mathbf{T}^{\mathtt{uncurry}}(E_{\text{curr}}) = \{f(a), g(b), f(a, g(b, c)), g(b, c), a, b, c\}$$

Notre procédure construit sur cet ensemble deux index, un premier basé sur le système de signature, et un second sur le type. L'ensemble des candidats du trigger $F(a)$ est le sous-ensemble $\{f(a), g(b)\}$ associé au type $\tau \to \tau$. •

La procédure présentée ne synthétise pas de fonctions (c.-à-d. qu'elle ne peut pas créer de nouvelles fonctions), même si une heuristique est donnée plus bas pour énumérer des fonctions. Par conséquent, pour instancier les variables de type fonctionnelles, et en particulier les variables de tête, la procédure essaie d'associer autant que possible ces variables à des symboles, ou sous-termes présents dans E. Dans l'exemple ci-dessus 3.7.2, si l'on matche le trigger $F(a)$ avec le terme $f(a)$, on obtient la substitution $\{F \mapsto f\}$. Cette approche est incomplète mais donne de bons résultats en pratique.

Formellement, la procédure d'E-matching, désignée par $e(s, t, S)$, est composée d'un trigger s dans la première composante, d'un terme ground t et d'un ensemble S de substitutions. La fonction e liste toutes les correspondances possibles entre son premier argument s et le deuxième argument t. Un appel à $e(s, t, S)$ permet d'obtenir toutes les substitutions grounds dans S,

$$
\begin{aligned}
e(x,t,S) &= \{\sigma \cup \{x \mapsto t\} \mid \sigma \in S,\, x \notin dom(\sigma))\} \cup \{\sigma \mid \sigma \in S,\, E \models x\sigma \simeq t\} \\
e(t',t,S) &= \begin{cases} S & \text{si } E \models t' \simeq t \\ \varnothing & \text{sinon} \end{cases} \\
e(a(\overline{s_n}),t,S) &= \bigcup_{\substack{a(\overline{t_n}) \in \mathbf{T}^{\mathtt{uncurry}}(E_{\mathrm{curr}}) \\ E \models a(\overline{t_n}) \simeq t}} e(s_n,t_n,e(s_{n-1},t_{n-1},(\ldots,e(s_1,t_1,S)\ldots))) \\
e(F(\overline{s_n}),t,S) &= \bigcup_{\substack{\mathtt{Ty}(F)=\mathtt{Ty}(a) \\ a(\overline{t_n}) \in \mathbf{T}^{\mathtt{uncurry}}(E_{\mathrm{curr}}) \\ E \models a(\overline{t_n}) \simeq t}} e(s_n,t_n,e(s_{n-1},t_{n-1},(\ldots,e(s_1,t_1,S)\ldots))) \cup \{F \mapsto a\}
\end{aligned}
$$

FIGURE 3.4 – Règles étendus pour le E-matching

qui sont solutions de la contrainte de matching $E \models s\sigma \simeq t$, où σ est une substitution qui est contenue dans S et est obtenue modulo les littéraux dans E. $e(s,t,S)$ renvoie donc un ensemble de substitutions complètes (c'est-à-dire que pour chaque substitution contenu dans S, chaque variable de s est associée à un terme ground), si la contrainte de matching admet une solution, et renvoie la substitution vide dans le cas contraire. Pour traiter le cas où s est une application fonctionnelle, c.-à-d. $e(F(\bar{s}_n),t,S)$ où F est une variable fonctionelle, nous ajoutons une règle supplémentaire. Pour la contrainte $e(F(\bar{s}_n),t,S)$, dans la figure 3.4, le calcul met en correspondance le trigger $F(\bar{s}_n)$ avec des termes ground dans E de même type, sélectionnés comme ci-dessus. Par exemple, si un terme $f(\bar{t}_n)$ appartient à l'ensemble des termes indexés par le type du terme $F(\bar{s}_n)$ la règle $e(F(\bar{s}_n),t,S)$ remplacera la variable libre F par le symbole de fonction f trouvé par l'index et tente de faire correspondre tous les arguments $\bar{s}_n$ avec les arguments de la série de termes ground $\bar{t}_n$.

Exemple 3.7.3. Soit $E = \{f(a,g(b,c)) \simeq a, \forall F.\, F(a) \simeq h,\ \forall y.\, h(y) \not\simeq a\}$ l'ensemble de triggers $\{F(a), h(y)\}$ ou $a,b,c : \tau$, $h : \tau \to \tau$ et $f,g : \tau \times \tau \to \tau$. L'ensemble de termes ground curryfiés dans E est $\{((f,\, a\,), ((g,\, b),\, c)),\ (f,\, a),\ (g,\, b),\ ((g,\, b),\, c),\ f,\ g,\ a,\ b,\ c\}$. L'heuristique d'instanciation par triggers, peut typiquement choisir de résoudre la contrainte de E-matching entre le trigger $F(a)$ et le terme $h(y)$. Par conséquent la procédure présentée en figure va tenter de matcher $F(a)$, avec l'ensemble de sous-termes non curryfiés $\{f(a), g(b)\}$ du même type τ. •

La figure 3.4, étend les règles présentées dans [41], de l'algorithme de E-matching, pour prendre en considération les sous-termes partiellement appliqués dans la fermeture de congruence et les variables de type fonctionnel. On remarque que la règle s'appliquant pour le cas $e(a(\overline{s_n}),t,S)$ est modifiée de sorte à mettre en correspondance les termes dans une formulation non curryfiée, à partir d'un ensemble de termes curryfiés. De surcroît cet ensemble est partitionné en signatures, ce qui permet à ce calcul d'être aussi performant que l'ancien sur un fragment purement premier ordre. En effet seul le cas $e(F(\bar{s}_n),t,S)$ est réellement coûteux car il nécessite de mettre en correspondance tous les termes de E du type de la variable F. Dans les deux exemples suivant nous déroulons l'algorithme de E-matching sur des contraintes d'ordre supérieur.

Exemple 3.7.4. Soit l'ensemble de triggers $T = \{F(a), h(y)\}$, de contraintes $E = \{b \simeq a,\ f(a, g(b, c)) \simeq a, \forall F.\, F(a) \simeq h,\ \forall y.\, h(y) \not\simeq a\}$, et l'ensemble de sous-termes : $\mathbf{T}^{\mathtt{uncurry}}(E_{\text{curr}}) = \{f(a), g(b), f(a, g(b, c)), g(b, c), a, b, c, f, g\}$. Dans le cadre de l'approche par triggers l'algorithme de E-matching calcule d'abord l'ensemble de solutions pour le trigger $F(a)$:

$$e(F(a), f(a), \emptyset) = e(F(a), g(b), \{F \mapsto f\}) = \{\{F \mapsto f\}, \{F \mapsto g\}\}$$

Supposons maintenant qu'au second tour d'instanciation le nouvel ensemble de contraintes E' est $\{b \simeq a,\ f(a, g(b, c)) \simeq a, f(a) \simeq h,\ g(b) \simeq h\}$. Si on applique à nouveau l'heuristique d'instanciation par triggers à partir de E' on obtient la solution suivante pour le trigger $h(y)$:

$$e(h(y), g(b, c), \emptyset) = e(h(y), f(a, g(b, c)), \{y \mapsto c\}) \quad = \{\{y \mapsto c\}, \{y \mapsto g(b, c)\}\}$$

Notons par ailleurs que les correspondances $e(h(y), a, S)$, $e(h(y), b, S)$, $e(h(y), c, S)$ conduisent à des ensembles de solution vides. En utilisant les solutions produites par l'approche par triggers nous pouvons déduire par raisonnement sur les égalités que l'ensemble de contraintes $\{b \simeq a,\ f(a, g(b, c)) \simeq a, f(a) \simeq h,\ g(b) \simeq h, h(c) \not\simeq a, , h(g(b, c)) \not\simeq a\}$ nouvellement obtenu est non satisfaisable. •

Exemple 3.7.5. Considérons les symboles $g : \tau \to \tau$, $f : \tau \times \tau \to \tau$, $a, b : \tau$, ainsi que l'ensemble de contraintes $E = \{f(a, b) \not\simeq g(b)\}$ et $Q = \{\forall F.\, F(a) \simeq g\}$, et l'unique trigger est $F(a)$. Dans le but de trouver une substitution pour F, l'algorithme de matching, donné figure 3.4, doit faire appel à la dernière règle. Le système d'index fait une requête sur les termes de la fermeture de congruence de type $\tau \to \tau$, qui est l'ensemble des termes $\{f(a),\ g\}$. Notons que l'ensemble de termes originaux $\mathbf{T}(E)$ ne contient pas $f(a)$ ou g, mais les termes apparaissent tous deux dans le E-graph de la fermeture de congruence. En utilisant l'algorithme de E-matching avec les appels suivants $e(F(a), f(a), S)$ et $e(F(a), g, S)$ nous obtenons la solution $\{F \mapsto f\}$ à partir de la première paire. En appliquant cette substitution nous pouvons obtenir l'instance $(f, a) \simeq g$, qui combinée à l'ensemble de contraintes E, est non satisfaisable par raisonnement équationnel. •

3.7.1 Synthétiser des fonctions pour la logique d'ordre supérieur

Nous discutons dans cette sous-section des extensions permettant d'étendre le calcul présenté ci-dessus pour synthétiser (unifier) des fonctions. Le calcul ci-dessus ne "crée" pas de nouvelles fonctions mais se contente de trouver les correspondances avec les termes déjà présents dans E. Cette approche permet d'obtenir de bon résultats mais elle reste néanmoins incomplète. Dans le but d'enrichir cette procédure nous proposons une heuristique permettant d'énumérer certaines fonctions.

Considérons $p, q : \tau \to o$ et $f : \tau \times \tau \to \tau$, avec l'ensemble de contraintes suivantes :

$$\begin{array}{lcl} E & = & \{q(f(a, b)),\ \neg p(f(a, a))\} \\ Q & = & \{\forall (F : \tau \times \tau \to \tau)\,(y, z : \tau).\, p(F(y, z)) \vee \neg q(F(b, y))\}. \end{array}$$

Concrètement le problème ci-dessus est un problème d'ordre supérieur. Il est possible pour ce problème de trouver un ensemble incohérent d'instances. Par exemple si l'on dispose d'une procédure d'unification qui nous permet de dériver la substitution suivante $\{F \mapsto \lambda w_1 w_2.\ f(a, w_1),\ y \mapsto a,\ z \mapsto a\}$, alors le problème est non satisfaisable. Concrètement, l'algorithme utilisé pour instancier des formules de la logique du premier-ordre n'est pas capable de trouver de telles substitutions, car il n'est pas capable de synthétiser de nouvelles λ-expressions. Dans le but de répondre à cette problématique nous proposons pour l'approche *redesign* une extension de l'algorithme de E-matching, basée sur l'algorithme développé par Gerard Huet [63], permettant le matching pour des contraintes équationnelles de la logique d'ordre supérieur. En particulier dans cette extension, lorsque l'algorithme rencontre le cas de figure où il doit mettre en correspondance un trigger dont la tête est une variable fonctionnelle, l'algorithme peut choisir de matcher ou d'énumérer de manière heuristique une liste de candidats-solutions possibles. Par conséquent cette astuce permet de simuler les règles d'imitation et d'identification coûteuses de l'algorithme d'unification, de manière contrôlée. Par exemple si l'on reprend la formule ci-dessus contenue dans Q, l'algorithme de E-matching présenté plus haut appelé avec en argument $F(y, z)$, $f(a, a)$ et la substitution initiale vide, tel que $e(F(y, z), f(a, a), \emptyset)$, produit la solution $\{F \mapsto f,\ y \mapsto a,\ z \mapsto a\}$. Cependant cette solution ne permet pas de dériver une instance incohérente avec E. Maintenant si on utilise l'heuristique d'énumération, il est possible de produire les solutions suivantes pour F :

$$F \mapsto \lambda w_1 w_2.\ f(w_1, w_2) \tag{3.1}$$
$$F \mapsto \lambda w_1 w_2.\ f(w_2, w_1) \tag{3.2}$$
$$F \mapsto \lambda w_1 w_2.\ f(a, w_1) \tag{3.3}$$
$$F \mapsto \lambda w_1 w_2.\ f(w_1, a) \tag{3.4}$$
$$F \mapsto \lambda w_1 w_2.\ f(a, w_2) \tag{3.5}$$
$$F \mapsto \lambda w_1 w_2.\ f(w_2, a) \tag{3.6}$$
$$F \mapsto \lambda w_1 w_2.\ f(a, a) \tag{3.7}$$

dans lesquelles $(2) - (7)$ sont des variations obtenues en permutant les arguments de la fonction avec des constantes en fonction des correspondances qui ont été trouvées. En particulier, en utilisant la substitution (3.3) pour F il est possible de dériver une instance incohérente avec E. En effet la la substitution $\{F \mapsto \lambda w_1 w_2.\ f(a, w_1),\ y \mapsto a,\ z \mapsto a\}$ générée à partir de l'heuristique d'énumération, permet de dériver l'instance $p(f(a, a)) \vee \neg q(f(a, b))$ qui est incohérente avec $\{q(f(a, b)), \neg p(f(a, a))\}$. Cette technique a été implémentée dans les deux approches, *redesign* et *pragmatique*.

Étendre l'expressivité par des axiomes Même si le fait de ne pas synthétiser les λ-abstractions nous empêche d'élever pleinement les techniques d'instanciation énumérées ci-dessus à la logique d'ordre supérieur, nous remarquons que, comme nous le voyons dans la section 3.8, l'extension pragmatique permet très souvent de prouver des problèmes de la logique d'ordre supérieur, et souvent même plus, sur certains types de problèmes, que les démonstrateurs d'ordre supérieur dédiés. Le nombre de problèmes résolus peut encore être augmenté si l'on utilise cer-

tains axiomes permettant de simuler en un certain sens l'unification d'ordre supérieur, tout en utilisant une version restreinte de cette algorithme. Utiliser de tels axiomes permet notamment de prouver des problèmes qui ne peuvent être prouvés sans synthétiser de l'λ-abstractions.

Exemple 3.7.6. Soit la formule ground $\varphi = a \not\simeq b$ avec a, b de type τ et la formule quantifiée $\psi = \forall F, G : \tau \to \tau.\, F \simeq G$. Intuitivement ψ énonce que toutes les fonctions de sorte $\tau \to \tau$ sont égales. Toutefois, cela n'est pas cohérent avec φ, qui oblige τ à contenir au moins deux éléments et donc $\tau \to \tau$ à contenir au moins quatre fonctions. Pour qu'un démonstrateur automatique puisse détecter cette incohérence, il doit appliquer une instanciation de ce type $\{F \mapsto (\lambda w.\, a),\, G \mapsto (\lambda w.\, b)\}$ à ψ, qui nécessite donc l'utilisation de l'unification d'ordre supérieur. Cependant, si l'on considère l'axiome

$$\forall F : \tau \to \tau.\, \forall x, y : \tau.\, \exists G : \tau \to \tau.\, \forall z : \tau.\, G(z) \simeq \mathsf{ite}(z \simeq x,\, y,\, F(z)) \tag{SAX}$$

alors le problème devient prouvable, sans avoir recours à la synthétisation de λ-abstractions. •

Nous désignons l'axiome ci-dessus comme l'axiome *store* (SAX) car il simule l'utilisation de l'axiome de modification d'élément utilisé dans la théorie des tableaux. Comme nous le noterons dans la section 3.8, l'introduction de cet axiome pour toutes les sortes de fonctions présentes dans le problème permet souvent à la version étendue de cvc4, de prouver des problèmes qu'il ne pourrait pas prouver autrement. Intuitivement, la raison en est que les instances, telles que celles de l'exemple ci-dessus, peuvent être générées non seulement à partir des termes du problème original, mais aussi à partir de l'ensemble plus large des fonctions représentables dans la signature de la formule.

3.8 Évaluation

Dans cette section nous montrons les principaux résultats expérimentaux obtenus sur les deux approches présentées. D'une part nous montrons expérimentalement ce que nous avons affirmé en section 3.6.1, c'est-à-dire qu'utiliser un algorithme d'une complexité algorithmique moins bonne, au bénéfice d'une structure de données plus flexible, n'a en pratique pas de conséquence importante sur les performances. D'autre part nous montrons que les approches implémentées respectivement dans cvc4 et veriT surclassent tant les approches par encodages applicatifs traditionnellement employés par les outils de type « hammer », que l'utilisation de prouveurs spécialement conçus pour la logique d'ordre supérieur.

3.8.1 Raisonnement d'ordre supérieur pour le solveur ground

Dans cette première partie de nos évaluations nous avons voulu montrer que l'utilisation d'une structure de données flexible et de complexité en temps théoriquement plus grand, n'était pas réellement moins efficace en pratique. Et pour cela nous avons mené deux expériences sur

le solveur SMT veriT, dont l'implémentation est décrite dans la section précédente. Les deux expériences ont été menées sur les benchmarks de la catégorie QF_UF de la SMT-LIB 2018. Ces problèmes sont pertinents car ce sont les problèmes utilisés par la communauté SMT pour évaluer l'évolution de l'efficacité des différents solveurs SMT au cours du temps. Chaque année une compétition, regroupant l'ensemble des solveurs SMT développés à travers le monde, est organisée dans le but d'évaluer, et de comparer ces solveurs sur différentes théories (correspondants à des catégories distinctes). À la fin de cette compétition chaque année un classement est formé par catégorie. Pour information le solveur SMT veriT est depuis quelques années, classé second, derrière le solveur Yices2, à quelques problèmes près, dans la catégorie QF_UF, qui correspond aux problèmes équationnels sans quantificateurs. L'architecture premier ordre, développée pour le raisonnement équationnel dans le solveur veriT est par conséquent portée à un niveau d'optimisation extrême, et donc un bon point de comparaison pour évaluer notre nouvelle approche. Nous avons donc comparé en premier lieu les résultats obtenus sur les problèmes de la catégorie QF_UF par notre nouvelle approche aux résultats de l'approche premier ordre de veriT. Les résultats de cette évaluation sont exposés par le graphique de droite de la figure 3.5. En deuxième lieu nous avons comparé cette approche face au solveur SMT cvc4, qui est depuis de nombreuses années, classé troisième derrière veriT dans la catégorie QF_UF. Cette évaluation est représentée par le graphique de gauche de la figure 3.5.

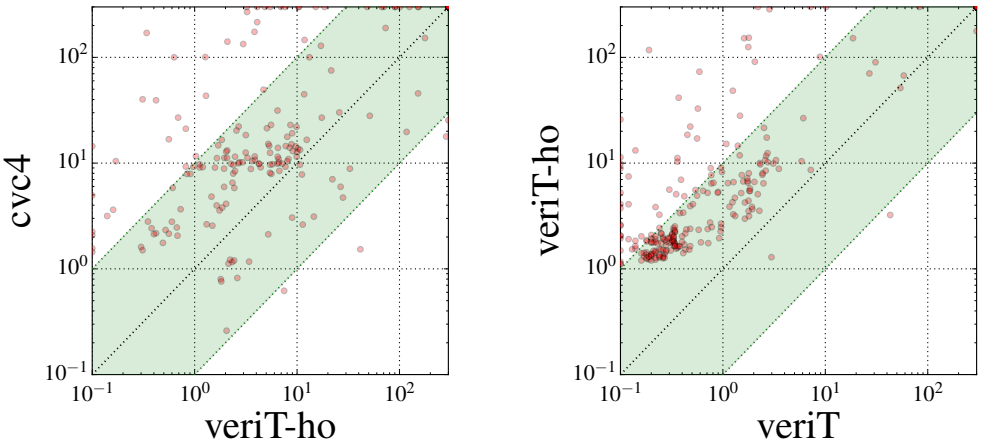

FIGURE 3.5 – Comparaisons des temps d'executions des solveurs cvc4 veriT et veriT-ho sur les problèmes QF_UF.

Ce graphique compare les temps d'exécution entre les deux versions du solveur, exécutés sur l'ensemble des problèmes de type équationnel de la catégorie QF_UF avec un time out de 60 secondes. Le graphique de droite dans la figure 3.5 se lit de la façon suivante : chaque axe correspond à une configuration du solveur : l'axe des abscisses exprime les résultats de la version originale premier ordre du solveur veriT, et l'axe des ordonnées la version de veriT dite ordre supérieur (veriT-ho). Chaque axe est gradué en secondes, et les 3 obliques représentent des fonctions de temps. Au centre la fonction $x = y$, si un point se trouve sur cette droite alors cela indique que les deux configurations ont un temps d'exécution identique, pour un problème en particulier. Au-dessus la fonction $x = 10y$, si un point se trouve sur cette courbe cela signifie que la configuration placée sur l'axe des ordonnées s'est exécutée dix fois moins vite, sur un

problème en particulier, que la configuration placée sur l'axe des abscisses, respectivement un point placé sur la courbe au-dessous $x = 10/y$, signifie que la configuration sur l'axe des abscisses s'est exécutée dix fois moins vite, sur un problème en particulier, que la configuration sur l'axe des ordonnées pour le même problème. Le graphique de gauche de cette même figure se lit de façon similaire, l'axe des abscisses exprime les résultats obtenus par la version de veriT dite ordre supérieur (veriT-ho), et l'axe des ordonnées les résultats obtenus par cvc4.

Par conséquent on observe sur ces graphiques que d'une part la nouvelle architecture ne pénalise pas fortement le solveur sur un raisonnement purement équationnel puisque l'implémentation de l'approche flexible est en moyenne 5 fois moins rapide que la version optimisée ; de plus si l'on rapproche ses résultats en comparaison de ceux obtenus face au solveur cvc4, sur le graphique de gauche de la figure 3.5, on observe que les différences ne permettent toujours pas au solveur cvc4 d'être plus rapide sur ce fragment que la version dite ordre supérieur de veriT, identifiée par veriT-ho dans la figure 3.5. Par conséquent si veriT avait participé à la compétition avec cette nouvelle architecture, cela n'aurait pas affecté le classement final, et veriT aurait conservé sa seconde place.

	SMT-LIB QF_UF benchmarks on 300 secondes time out	
	success 300s	between 0 and 50s
veriT-ho	6643	6629
cvc4	6625	6611
veriT	6646	6643

TABLE 3.1 – Nombre de problèmes résolus sur les benchmarkes QF_UF.

Pour information la table 3.1, ci-dessus reporte le nombre total de problèmes résolus par les trois solveurs comparés dans la figure 3.5. La version nommée veriT-ho dénote la version de veriT dite ordre supérieur et implémente les algorithmes décrits dans cette thèse.

3.8.2 Évaluer les nouvelles approches

Dans cette section nous donnons une évaluation des deux approches décrites dans cette thèse, l'une développée dans le solveur à titre de comparaison cvc4 par l'équipe de l'université d'Iowa, et la seconde développée dans le solveur veriT à Nancy. Pour valider nos approches nous distinguons deux versions de chaque solveur : une version qui s'appuie sur un codage applicatif, (section 3.2) premier ordre, noté @cvc et @vt, et une autre qui utilise les extensions pragmatiques (Sections 3.5) ou la version *redesign* de veriT (section 3.6) pour la logique d'ordre supérieur, notées cvc et vt. Les deux modes implémentés dans le solveur cvc4 éliminent les λ-abstractions via le λ-lifting. Cependant, pour des raisons techniques nous n'avons pas évalué les deux configurations de veriT avec le module de λ-lifting. Par conséquent aucune des versions de veriT ne prend en charge les benchmarks contenant des λ-abstractions. La version du solveur cvc4 utilisant l'axiome "store axiom" décrite en section (section 3.7) est indiquée par le suffixe -sax.

Nous utilisons les démonstrateurs automatiques d'ordre supérieur Leo-III [111], Satallax [33, 52] et Ehoh [106, 122] comme base de référence dans notre évaluation. Les deux premiers solveurs sont équipés de calculs complets pour la réfutation pour la logique d'ordre supérieur avec extensionnalité dans la sémantique de Henkin, tandis que le troisième ne supporte que la logique d'ordre supérieur sans λ, et sans fonctions booléennes de première classe. Pour les démonstrateurs Leo-III et Satallax nous utilisons les configurations employées pour la compétition CASC [115], tandis que pour le démonstrateur Ehoh nous utilisons la meilleure configuration sans portfolio utilisée dans Vukmirović et al., Ehoh [122].

Nous avons mené nos expériences sur un cluster de machines équipées de processeurs Intel E5-2637 v4 fonctionnant sous Ubuntu 16.04, utilisant un coeur, 60 secondes et 8 Go de RAM pour chaque tâche. L'ensemble des données expérimentales est accessible au public [2].

Nous considérons les ensembles de problèmes d'ordre supérieur suivants [3] : nous séparons les 3188 problèmes de type monomorphique extraits des benchmarks TPTP [114] de la catégorie ordre supérieur en trois sous-ensembles : 530 problèmes avec et sans λ-abstractions et sans variable de prédicats (TH0) ; 743 problèmes seulement sans λ-abstractions (oTH0) ; et 1915 problèmes sans λ-abstractions ni variables de prédicat (λoTH0). Le reste des problèmes sur lesquels les solveurs sont évalués sont des problèmes extraits de formalisations Isabelle, et générés depuis l'outil Sledghammer (SH). Plus particulièrement, ce sont des problèmes extraits de la banque de problèmes Judgment Day [28], qui est constituée de 1253 buts prouvables *manuellement* choisis depuis des théories de l'assistant de preuves Isabelle [89] puis encodés dans le fragment de la logique d'ordre supérieur, λ-free monomorphique, sans variable de prédicat. Les problèmes encodés sont prouvables uniquement si le but original l'est. Généralement l'outil Sledghammer utilise un certain nombre d'heuristiques pour sélectionner un sous-ensemble d'axiomes nécessaires pour démontrer un but. Généralement il est possible de spécifier à Sledghammer le nombre de ces axiomes. Dans nos évaluations ces problèmes sont donc divisés en quatre sous-ensembles, $\mathrm{JD}^{32}_{\mathrm{lift}}$, $\mathrm{JD}^{32}_{\mathrm{combs}}$, $\mathrm{JD}^{512}_{\mathrm{lift}}$, et $\mathrm{JD}^{512}_{\mathrm{combs}}$ suivant le nombre (32 ou 512) de lemmes sélectionnés par Sledghammer. Les λ-abstractions sont supprimées via λ-lifting ou via des combinateurs de type SK. Le dernier ensemble $\lambda o\mathrm{SH}^{1024}$, contient 832 problèmes. Ces 832 problèmes, correspondent à 832 buts prouvables extraits aléatoirement depuis différentes théories d'Isabelle. Ces problèmes sont accompagnés de 1024 lemmes et incluent toutes les fonctionnalités de la logique d'ordre supérieur, c'est-à-dire les λ-abstractions et les variables de prédicats. En considérant un nombre variable de faits dans les benchmarks SH, on reproduit les besoins engendrés par des problèmes de plus en plus importants produits dans le cadre de la vérification interactive. De surcroît, les différents systèmes de traitement de λ permettent de mesurer de quelles alternatives chaque solveur bénéficie le plus.

Nous soulignons que nos extensions de cvc4 et veriT n'altèrent pas significativement leur performance sur des problèmes de la logique du premier ordre. L'extension pragmatique du

2. `http://homepage.divms.uiowa.edu/~hbarbosa/papers/hosmt/`

3. Étant donné que veriT ne supporte pas la syntaxe TPTP, ces résultats sont exécutés sur des problèmes traduits depuis le solveur cvc4 dans le langage SMT-LIB pour l'ordre supérieur décrit en début de thèse et dans [7].

solveur cvc4 se comporte presque de la même façon que le solveur original sur les problèmes premier ordre de la SMT-LIB [12], comme le montrent les résultats obtenus dans [11]. Pour la nouvelle version de veriT les résultats obtenus dans la sous-section 3.8.1, montrent que les performances sont tout de même un peu dégradées, mais de façon acceptable.

3.8.3 Prouver des théorèmes d'ordre supérieur

Solver	Total	TH0	*o*TH0	λoTH0	$\text{JD}^{32}_{\text{lift}}$	$\text{JD}^{32}_{\text{combs}}$	$\text{JD}^{512}_{\text{lift}}$	$\text{JD}^{512}_{\text{combs}}$	$\lambda o\text{SH}^{1024}$
#	9032	530	743	1915	1253	1253	1253	1253	832
@cvc	4318	384	344	940	457	459	655	667	**412**
@cvc-sax	4348	390	373	937	456	457	655	**668**	**412**
cvc	4232	389	342	865	463	447	**667**	654	405
cvc-sax	4275	389	376	883	458	443	**667**	654	405
Leo-III	**4410**	**402**	452	1178	**491**	**482**	609	565	231
Satallax	3961	392	**457**	**1215**	394	390	407	404	302
@vt		370	332		404	396	525	529	
vt		369	346		426	424	550	556	
Ehoh		394			489	481	637	630	

TABLE 3.2 – Théorèmes prouvés par ensemble de benchmarks. Les meilleurs résultats se trouvent en **gras**.

Le nombre de théorèmes prouvés est donné pour chaque configuration, et chaque ensemble de benchmarks dans la Table 3.2. Les cellules grisées représentent les benchmarks non supportées. La Figure 3.6 compare le nombre total de benchmarks résolus en fonction du temps. Cela inclut seulement les benchmarks supportés par tous les solveurs (c'est à dire les benchmarks TH0 et JD).

Comme attendu, les résultats varient considérablement d'un ensemble de benchmarks à l'autre. Leo-III et Satallax ont un net avantage sur les problèmes TPTP, qui contiennent un nombre important de petits problèmes logiques destinés à exercer les démonstrateurs purement conçus pour des problèmes d'ordre supérieur. De plus si l'on considère les benchmarks TPTP des moins expressifs aux plus expressifs, c'est-à-dire ceux autorisant variables de prédicats puis ceux autorisant les λs, on constate que l'avantage de ces systèmes ne fait qu'augmenter. Nous observons également que les configurations @cvc et cvc, et en particulier celles bénéficiant de l'axiome SAX révèlent que l'absence de synthétisation des λ-expressions peut parfois être compensée par des axiomes bien choisis. Néanmoins, les résultats sur λoTH0 montrent que cet axiome seul est loin d'être suffisant pour compenser l'écart entre @cvc et cvc, cvc abandonnant plus souvent par manque d'instances purement "d'ordre supérieur".

Les problèmes engendrés par Sledghammer proviennent de formalisations de différentes applications dans les assistants de preuve. Comme le font remarquer [113,122], le goulot d'étranglement dans la résolution de ces problèmes est souvent l'extensionalité, et le raisonnement efficace au niveau premier ordre, plutôt que le raisonnement d'ordre supérieur en lui-même, en particulier

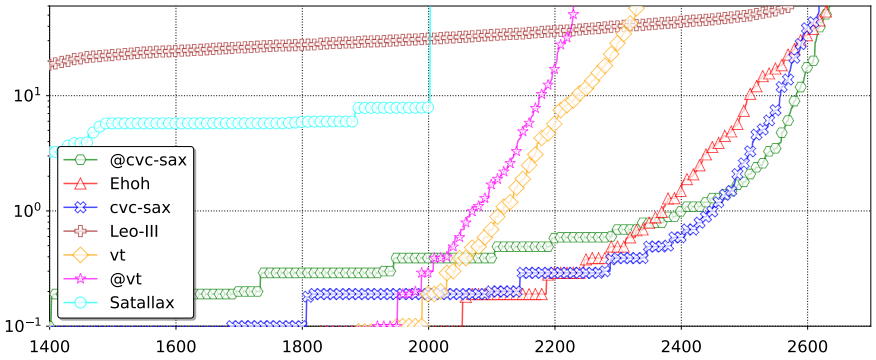

FIGURE 3.6 – Temps d'exécutions en secondes sur 5543 benchmarks, provenant de TH0 et JD, supporté par tous les solveurs.

lorsque il est nécessaire d'inclure un nombre important de lemmes pour montrer le but. Par conséquent la capacité à synthétiser des fonctions n'est pas un élément suffisant pour en pratique résoudre des problèmes provenant d'applications concrètes, type assistant de preuve. En particulier des extensions de Ehoh et cvc4 surpassent des démonstrateurs purement d'ordre supérieur. Sur le plus gros ensemble de benchmarks, $\lambda o\text{SH}^{1024}$, on observe que les deux configurations @cvc et cvc ont un net avantage. Intuitivement, nous pensons que les raisons pour lesquelles, @cvc est plus efficace que cvc sur les benchmarks $\lambda o\text{TH0}$, $\lambda o\text{SH}^{1024}$ est dû au fait que cette version permet de déduire plus d'instances, puisque dans cette configuration les types fonctionnels sont considérés comme des types simples. Cependant, cvc est souvent plus rapide (15 % en moyenne) que @cvc (voire figure 3.6).

La table 3.3 reporte les résultats obtenus en utilisant des stratégies de type portefeuille. Elle indique le nombre de problèmes résolus uniquement par chaque entrée en fonction de chaque ensemble de benchmarks.

Solver	Total	TH0	oTH0	λoTH0	$\text{JD}^{32}_{\text{lift}}$	$\text{JD}^{32}_{\text{combs}}$	$\text{JD}^{512}_{\text{lift}}$	$\text{JD}^{512}_{\text{combs}}$	$\lambda o\text{SH}^{1024}$
#	9032	530	743	1915	1253	1253	1253	1253	832
all-cvc-port	**4616**	**408** (8)	385 (2)	1001 (26)	482	482 (5)	**703** (38)	**702** (38)	**453** (137)
vt-port	2746	376 (1)	351 (3)		444 (3)	441 (3)	565	569	
Ehoh-port	2749	399 (1)			**494** (2)	**485** (1)	690 (31)	681 (31)	
Leo-III	4410	402 (1)	452 (21)	1178 (53)	491 (5)	482 (3)	609 (1)	565 (2)	231 (5)
Satallax	3961	392	**457** (18)	**1215** (101)	394	390	407 (8)	404 (3)	302 (24)

TABLE 3.3 – Théorèmes prouvés par des configurations portefeuille de [@]cvc[−ui][−sax], [@]veriT et Ehoh[a|as|b|hb], par ensemble de benchmarks. Les meilleurs résultats sont en **gras**. Le nombre de benchmarks résolus de manière unique se trouve entre parenthèses.

À la différence de l'extension pragmatique de cvc4, qui offre une alternative à l'encodage applicatif, la nouvelle approche, surnommée *redesign*, offre une nette amélioration, vt résout systématiquement plus de problèmes, et plus rapidement que @vt, en particulier pour les problèmes

plus difficiles, comme le montre la séparation plus importante entre les deux versions du solveur après 10 s dans la figure 3.6. Dans l'ensemble, les performances des deux approches implémentées dans veriT, qui est un solveur moins abouti si on le compare à E ou cvc4, sont à la traîne à mesure que les benchmarks contiennent plus de lemmes. Cependant, en tenant compte du niveau d'élaboration du solveur veriT, globalement moins performant déjà au premier ordre, ces résultats sont respectables en comparaison aux performances de Leo-III et Satallax. Ces résultats sont prometteurs, et nous encouragent à penser que les prochaines générations du solveur SMT devraient s'appuyer sur une architecture similaire à celle présentée dans cette thèse. Cette nouvelle génération de solveur offrirait par conséquent tous les avantages des solveurs premier ordre avec des capacités de raisonnement accrues pour la logique d'ordre supérieur. Ce qui en tout état de cause contribuerait grandement à réduire l'écart entre les plateformes de vérification formelles, et les outils automatiques. Par ailleurs, bien que ce ne soit pas mentionné dans le tableau 3.2, l'utilisation de l'extension de l'algorithme de E-matching d'ordre supérieur (section 3.7.1) s'appuyant sur l'heuristique d'énumération de fonctions donne aussi de bon résultats. Elle permet notamment de résoudre davantage de problèmes difficiles, mais trop peu pour qu'il soit utile d'en rendre compte dans cette section.

3.9 Conclusions et directions futures

Nous avons présenté dans ce chapitre des extensions pour les solveurs SMT permettant de traiter des problèmes de la logique d'ordre supérieur. L'extension pragmatique implémentée dans le solveur SMT cvc4 peut être mise en œuvre dans d'autres solveurs SMT avec un niveau d'effort équivalent. Cette approche fonctionne de manière similaire à l'approche standard basée sur un encodage applicatif, et ce malgré un support limité pour l'instanciation d'ordre supérieur. En outre, l'approche pragmatique permet de résoudre de nombreux nouveaux problèmes grâce à une stratégie portefeuille, les performances sont très compétitives et souvent supérieures à celles des démonstrateurs automatiques d'ordre supérieur. D'autre part, l'approche *redesign* implémentée dans le solveur SMT veriT surpasse constamment son homologue standard basé sur un encodage applicatif classique. Ces résultats montrent en outre que cette approche peut servir de base vers une automatisation plus poussée et plus systématique de la logique d'ordre supérieur dans d'autres solveurs SMT.

Les directions futures pour ce travail concernent donc principalement l'intégration d'un algorithme de E-matching d'ordre supérieur permettant de synthétiser de façon efficace des fonctions, de façon similaire aux travaux de Vukmirovic et al [121]. Et bien sûr dans le but d'étendre toutes les stratégies en place, il est indispensable d'étendre de façon gracieuse le framework CCFV à la logique d'ordre supérieur. Ce dernier point n'a pas été discuté dans cette thèse car il est à l'heure actuelle toujours en cours de développement. Cependant nous pouvons déjà affirmer qu'une extension syntaxique du calcul, c'est-à-dire avec une présentation curryfiée des règles n'est pas efficace. En effet cette approche a été implémentée dans veriT. Cependant cette approche a rapidement montré ses limites et n'a pas surpassé les résultats obtenus par l'approche pré-

sentée dans ce chapitre. Le problème majeur de cette extension est la structure d'index, et cela semble être une limitation nécessitant un profond remaniement des structures de données et algorithmes. Récemment en collaboration avec Sophie Tourret, Pascal Fontaine et Haniel Barbosa une approche basée sur un encodage SAT [116] a été développée. CCFV étant un problème NP-complet, il semble donc judicieux d'essayer de déléguer la recherche combinatoire à un solveur SAT optimisé. En particulier, cette approche semble être prometteuse car elle permet de mieux gérer l'ensemble des contraintes liées au grand nombre de candidats unifiables.

4

Implémentation du solveur ground pour la logique d'ordre supérieur

Sommaire

Dans ce chapitre nous nous intéressons à la partie pratique de l'approche adoptée dans le solveur SMT veriT. En particulier nous détaillons les choix d'implémentation utilisés pour la représentation des termes, la structure de données utilisée pour représenter les classes d'équivalences, et la gestion des explications. Ce chapitre a pour objet de donner une intuition sur le fonctionnement de la fermeture de congruence implémentée dans veriT, néanmoins nous nous efforcerons de donner une description de haut niveau, pouvant servir de point d'ancrage pour toute investigation plus poussée. Le code source est accessible depuis le dépôt [4].

4.1 Les termes curryfiés

La structure de données de la figure 4.1 permet de représenter les termes. La constitution des termes est simple, puisqu'elle se compose d'un champ `type` spécifiant si le type du terme : une constante, variable, application ou bien une λ-expression. En fonction de son type, le terme admet, 0 (constante et variable), 1 (λ-expression), ou 2 (application) sous-termes, représentés dans le champ `sons` qui est un tableau de termes. Le champ `sort` permet de stocker le type de

4. `https://github.com/delouraoui/these`

chaque terme. De manière globale les termes sont stockés dans un tableau de `TSnode` global, et chaque terme est associé à un entier ici renommé comme `Tnode`.

```
1  typedef enum {EMPTY, VAR, CONST, APP, LAM} Tnode_type;
2
3  typedef unsigned Tnode;
4
5  typedef struct TSnode {
6    Tnode_type type:3;
7    Tnode sons[2];
8    Tsort sort;
9  } TSnode;
10
11 typedef struct Tnode_app_infos {
12   unsigned position;
13   TDAG DAG;
14   unsigned key;
15   Tnode node;
16 } Tnode_app_infos;
```

FIGURE 4.1 – Structure de termes

Pour pouvoir assurer l'interaction entre les termes de premier ordre du solveur (de type `TDAG` dans le code de veriT), nous utilisons une structure de données supplémentaire appelée `Tnode_app_infos`. Cette structure permet d'assurer le fonctionnement de manière interne des fonctions décrites plus haut `curry` et `uncurry`. Intuitivement, chaque terme de type application est décomposé en plusieurs sous-termes. De manière à retrouver à partir de chacun de ces sous-termes son homologue de premier ordre (le `TDAG`) nous construisons un index stockant pour chaque sous-terme : sa `position` (c.-à-d. dans le `TDAG` le numéro de l'argument), le `TDAG` correspondant, la clé de hachage `key` permettant de stocker chaque sous-terme de manière unique, et le dit terme `node`. Pour construire chaque terme nous nous appuyons sur une fonction appelée `new_node(TDAG u)`, qui à partir d'un `TDAG` construit l'ensemble des sous-termes curryfiés, de type `Tnode`, de `u` (voir construction de $\mathbf{T}(E_{\text{curr}})$ dans la section 3.6.2, et exemple 3.6.2). Imageons le principe d'index de sous-termes et de position par l'exemple suivant :

Exemple 4.1.1. Si l'on considère le `TDAG` $f(a, b, c)$, la fonction `new_node(TDAG u)` va stocker dans l'index les applications-termes suivants : $(((f,\ a),\ b),\ c)$; $(f,\ a),\ b)$; $(f,\ a)$; et bien sûr les constantes f, a, b, c. Ainsi pour qu'à partir de chaque application nous soyons capable de retrouver le terme original, le pointeur `DAG`, pour chaque application donnée ici pointera vers $f(a, b, c)$. En plus chaque application se voit attribuée une position, ainsi $(((f,\ a),\ b),\ c)$ a la position 2 dans le terme original, $((f,\ a),\ b)$ la position 1 et $(f,\ a)$ la position 0. Par conséquent si l'on souhaite à partir du `DAG` $f(a, b, c)$ accéder au sous-terme $f\ a$ il suffit d'utiliser l'index, au travers d'une fonction d'accès en temps constant, `get_sub_DAG(DAG, position)`, via l'appel suivant `get_sub_DAG(`$f(a, b, c)$`, 0)` $= (f,\ a)$. •

L'utilité d'une telle construction est multiple. Premièrement cela permet de stocker chaque sous-terme de manière efficace. Deuxièmement, grâce à la combinaison position-`DAG` il est facile à partir d'un terme curryfié de retrouver son homologue, de déterminer les sous-termes ou d'introduire de nouveaux sous-termes dans le solveur quand c'est nécessaire, en particulier au moment de l'instanciation.

```
1  typedef enum Tboolean_value
2    {BOOL_FALSE = 0, BOOL_TRUE = 1, BOOL_UNDEFINED} Tboolean_value;
3
4  typedef struct Tcc {
5    bool visited:1;
6    Tboolean_value boolean_value:2;
7    unsigned nbparents:16;
8    unsigned nbedges:16;
9    Tnode  repr;
10   Tnode *parents;
11   Tedge *edges;
12 } Tcc;
```

FIGURE 4.2 – Représentation des nœuds du graphe d'égalités

4.2 Le graphe d'égalités

La structure de données utilisée pour représenter les classes d'équivalences dans notre algorithme de fermeture de congruence est une structure de graphe. Les figures 4.2 et 4.3, décrivent de manière simplifiée la structure de graphe. La figure 4.2, décrit les informations contenues dans chaque nœud du graphe. Pour parcourir ce graphe nous utiliserons majoritairement deux fonctions de parcours qui permettent de récolter l'ensemble des termes d'une classe d'équivalence. Ces parcours peuvent être en largeur ou en profondeur. Deux fonctions génériques `DFS(Tnode node)` et `BFS(Tnode node)` permettent à partir d'un terme d'obtenir, dans l'ordre du parcours en largeur et respectivement en profondeur, la liste des termes de sa classe d'équivalence. Pour effectuer ces opérations chaque nœud contient un champ booléen `visited`, qui permet de savoir si pendant un parcours le nœud a déjà été visité. Chaque nœud contient un pointeur `repr` vers le représentant de la classe d'équivalence à laquelle il appartient. De façon assez classique, chaque nœud dispose d'un pointeur vers l'ensemble de ses voisins ici incarné par `edges`, et de façon plus spécifique à notre application une liste de parents `parents`. La liste de parents est utilisée pour déduire les congruences. Enfin chaque arête est représentée par la structure `Tedge`, qui contient le nœud cible, le littéral associé à l'arête (s'il existe) dans E, et le type de l'arête. Une arête peut être d'égalité, de diségalité ou encore de congruence. Notons que cet ensemble peut être enrichi si l'on souhaite étendre notre procédure pour les data-types par exemple en ajoutant une arête de type injectif, ou encore pour des fonctionnalités d'ajout de nouveaux termes pendant le calcul de la fermeture de congruence.

```
1 typedef enum {DISEQ, EQ, CONG} Tedge_type;
2
3 typedef struct Tedge {
4   Tedge_type type:3;
5   Tnode node;
6   Tlit lit;
7 } Tedge;
```

FIGURE 4.3 – Représentation des arêtes du graphe d'égalités

Enfin si l'on veut calculer la fermeture de congruence de manière efficace il faut traiter les prédicats différemment des autres égalités. Intuitivement, on serait tenté d'introduire deux constantes spécifiques $\top$, et $\bot$, et réécrire chaque prédicat, disons $\mathsf{P}(a)$, avec une polarité positive comme l'égalité $P(a) \simeq \top$, et inversement représenté $\neg P(a)$ $\neg P(a) \simeq \bot$. Cependant cette représentation est très coûteuse, puisque l'ensemble des prédicats doivent êtres rassemblés en seulement deux classes d'équivalence. Ainsi toute opération de parcours, ou de modification de la classe d'équivalence peut devenir extrêmement coûteuse en temps, à mesure que le nombre de prédicats augmente. Cette solution est d'autant plus inenvisageable pour des applications industrielles, ou dans le cadre de la compétition SMT. Pour pallier ce problème on introduit une notion de *polarité de classe*. Dans notre structure par le champ `boolean_value`, permet d'attribuer une polarité à chaque nœud et par transitivité à chaque classe. Par conséquent une classe peut avoir la polarité neutre représentée par la valeur `BOOL_UNDEFINED` dans le cas de fonction non booléenne, et inversement avoir la polarité positive `BOOL_TRUE`, ou négative `BOOL_FALSE` en fonction de la polarité du prédicat considéré. Par défaut, à la création de la signature de la fermeture de congruence tous les termes ont une polarité neutre `BOOL_UNDEFINED`.

Dans les algorithmes donnés ci-dessous, nous utilisons une structure de données de liste un peu spéciale, les `Tstack`. Ce type de liste permet un usage assez flexible des objets. En interne, cette liste est représentée comme un tableau à taille variable. Il permet l'accès et la modification en temps constant. Ces listes peuvent être utilisées à la fois comme une pile, comme une liste ou un tableau classique, les fonctions `stack_get`, `stack_push`, `stack_pop`, `stack_size` permettent respectivement d'accéder à un élément, d'ajouter un élément en tête, de supprimer l'élément en tête, et d'accéder à la taille de la pile. Enfin on peut construire des listes de tous les types, par exemple pour exprimer une liste de `Tnode`, on déclare `Tstack_node`.

4.3 Calculer la fermeture de congruence

Le calcul de la fermeture de congruence est généralement effectué de manière incrémentale dans un solveur SMT. Cela signifie que pour un ensemble conjonctif $E = \{l_1, \ldots, l_n\}$ de n littéraux, chaque littéral est introduit tour à tour, et dans le contexte plus particulier de veriT, via une fonction appelée `CCS_assert`. On doit donc imaginer une boucle qui fait appel incrémentalement

à la fonction `CCS_assert(`l_i`)`. Le cœur de cette fonction `CCS_assert` repose essentiellement sur la fonction `union`, présentée par la figure 4.4, dans une version simplifiée. Dans le contexte SMT, et plus particulièrement de la théorie de l'égalité, l'ensemble des contraintes se présente sous la forme d'une liste de littéraux, qui sont soit des (dis)égalités, soit des prédicats. Par conséquent si l'on désire calculer la fermeture de congruence sur cette liste de littéraux, il faut considérer 4 cas : le cas d'une égalité, le cas d'une diségalité (égalité avec polarité négative), puis les cas des prédicats positifs ou négatifs.

Égalité avec polarité négative $u \not\simeq v$. Le cas d'égalité avec polarité négative $u \not\simeq v$, est assez trivial puisque il suffit dans notre structure de données de créer une arête de diségalité, entre les deux membres u, et v dans notre graphe d'égalité comme ceci `create_edge(u, v, DISEQ,` $u \not\simeq v$`);`. Bien évidement nous devons nous assurer avant d'effectuer cette opération que u et v ne sont pas dans la même classe d'équivalence, c.-à-d. qu'il n'ont pas le même représentant, auquel cas l'ensemble de contraintes est incohérent et un conflit doit être reporté via la fonction `conflict_rec(u, v)`.

Prédicat avec polarité négative u. Le cas d'un prédicat avec polarité négative est traité de façon similaire à celui d'une diségalité. Pour détecter toute incohérence entre prédicats il suffit d'observer la polarité de la classe d'équivalence du prédicat asserté au moment t. Par conséquent dans le cas d'un prédicat asserté avec une polarité négative, il suffit de considérer deux cas de figure : si aucune polarité n'a encore été attribuée à ce prédicat (polarité `BOOL_UNDEFINED`), alors cela signifie que ce prédicat n'a pas encore été introduit dans la fermeture de congruence, il suffit donc d'assigner une polarité négative à sa classe d'équivalence ; si au contraire ce prédicat apparaît avec une polarité positive au moment de l'assertion alors cela signifie que l'ensemble E est contradictoire puisque il existe un prédicat équivalent au prédicat asserté avec une polarité différente, un conflit est donc reporté avec un appel à la fonction `conflict_rec(u, PREDICATE_HOLE)` (`PREDICATE_HOLE` est une constante utilisée pour signifier la présence d'un prédicat).

Plus haut nous avons considéré les deux cas où les littéraux assertés ont une polarité négative, nous devons maintenant traiter les deux autres cas à savoir les égalités et les prédicats assertés avec une polarité négative.

Égalité avec polarité positive $u \simeq v$. C'est probablement le cas le plus complexe de cet algorithme, qui fait appel à plusieurs mécanismes. Pour traiter le cas d'une égalité positive il faut d'abord supposer l'existence d'une file de priorité globale de contraintes `merge_queue`, permettant de placer en attente l'ensemble des congruences déduites tout au long de ce processus de construction de la fermeture de congruence. Lorsqu'une égalité $u \simeq v$ est assertée, on fait donc appel à la fonction `CCS_union` de la manière suivante `CCS_union(u, v,` $u \simeq v$`, EQ)`. Au premier ordre les égalités entre prédicats ne sont pas autorisés, cependant ce n'est pas le cas pour la logique d'ordre supérieur. Par conséquent nous utilisons la fonction `set_bvalue`, ligne 5, qui nous permet de propager la polarité des prédicats lors de la fusion de deux classes

```
1  static void
2  CCS_union(Tnode u, Tnode v, Tlit lit, Tedge_type type)
3  {
4    Tstack_node ccpar_u, ccpar_v;
5    set_bvalue(u, v, (lit)?EQ:CONG, lit);
6    if (CCS_status == UNSAT || v == PREDICATE_HOLE) return;
7    stack_INIT(ccpar_u); stack_INIT(ccpar_v);
8    assert(get_repr(u) != get_repr(v));
9    /**< update the new class [u] U [v] or throw conflict if any **/
10   if(!update_class(u, v, get_repr(u), &ccpar_u, &ccpar_v)) return;
11   /**< create a new edge between u and v **/
12   create_edge(u, v, type, lit);
13   assert(get_repr(u) == get_repr(v));
14   CCS_deduce_cong(ccpar_u, ccpar_v);
15   stack_free(ccpar_u);
16   stack_free(ccpar_v);
17 }
```

FIGURE 4.4 – Procédure de fusion de deux classes d'équivalence

d'équivalence. Cette fonction est un peu complexe, mais une intuition sur son fonctionnement est esquissée dans le paragraphe suivant. Lors du processus de propagation de polarité il est possible qu'une incohérence soit détectée. Dans ce cas la valeur de la constante `CCS_status` est assignée à la valeur `UNSAT`, et il n'est pas nécessaire de continuer les calculs. En effet, puisqu'un conflit est reporté il faut donc redonner la main au solveur, car la condition ligne 6 est remplie. La deuxième condition, ligne 6, concerne le cas d'un prédicat asserté seul, et est expliquée dans le paragraphe suivant. Les listes `ccpar_u`, et `ccpar_v`, doivent contenir respectivement tous les parents des termes u et v. On remarque notamment l'invariant ligne 8, exigeant que les classes n'ont jamais été fusionnées auparavant. C'est une condition de non redondance. Ligne 10 la fonction `update_class`, procède à un double parcours qui est celui de la classe de u et de v. Dans cette fonction on élit un représentant de la nouvelle classe, par défaut celui de u. Dans ce parcours la fonction effectue plusieurs opérations, de manière totalement incrémentale. Pour chaque noeud visité on ajoute l'ensemble de ses parents dans la liste `ccpar_u`, si ce terme apparaît dans la classe de u, et inversement dans la liste `ccpar_v` s'il apparaît dans la classe de v, puis on met à jour le représentant. Ensuite on examine toutes les arêtes de ses voisins. Avant de connecter les deux classes d'équivalence, représentées dans notre structure de données sous forme de graphe, il faut s'assurer que la fusion des deux classes n'implique aucune contradiction. En d'autres termes, si l'ajout d'une arête d'égalité entre les deux graphes connectés implique que E est incohérent il faut être capable de le détecter avant l'ajout de cette arête. Pour s'assurer que la nouvelle égalité n'implique aucune incohérence nous allons lors du parcours examiner les voisins de chaque nœud du graphe d'égalité, respectivement de u et de v. Intuitivement la fusion de ces deux classes implique une incohérence si il existe une arête de type diségalité entre les deux classes comme représenté à la figure 4.5.

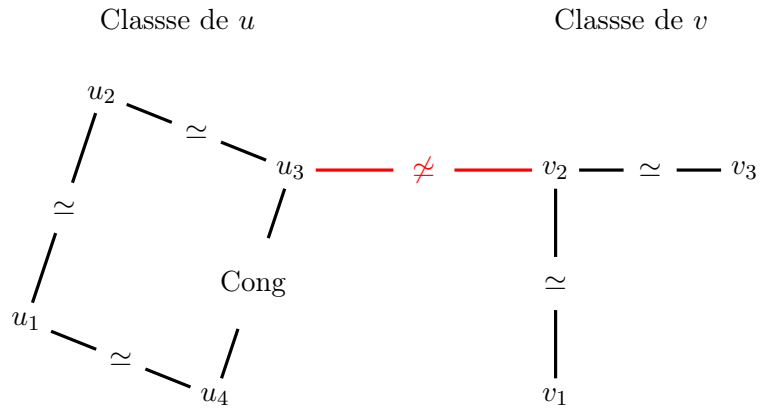

FIGURE 4.5 – Configuration de conflit si les classes u et de v doivent être fusionnées

Par conséquent pour détecter une telle situation il suffit de s'assurer que pour chaque nœud de la classe d'équivalence u (resp. v), aucun de ses voisins directs n'est à la fois un élément de la classe d'équivalence v (resp. u), et à la fois connecté par une arête de diségalité. Lorsqu'une telle incohérence est détectée par l'algorithme, la fusion est abandonnée, et un conflit est reporté. Autrement si la fusion s'est déroulée sans erreur, alors les deux classes peuvent êtres connectées par une arête d'égalité, ligne 12. Ligne 14, la fonction `CCS_deduce_cong`, décrite dans la figure 4.6, s'occupe de propager toutes les congruences induites par l'opération de fusion des deux classes. Intuitivement cette fonction parcoure les deux listes de parents, et ajoute une nouvelle contrainte de congruence dans la file de priorité `merge_queue`, au travers de la fonction `create_cong_deduction`, ligne 15 à la figure 4.6, lorsque les deux termes sont congruents. La représentation curryfiée des termes simplifie grandement la propagation des congruences à ce niveau. En effet tous les tests de congruence se font systématiquement en temps constant. Deuxièmement les deux formes de congruences nécessaires au raisonnement d'ordre supérieur sont automatiquement propagées par le test de congruence ligne 14, puisqu'il suffit de tester l'équivalence entre chaque sous-terme.

Prédicat avec polarité positive u. Le cas d'un prédicat positif asserté consiste à discriminer la polarité de la classe du prédicat, de façon similaire au cas où le prédicat est asserté avec une polarité négative. Si la classe du prédicat est neutre alors on lui assigne une polarité positive, et dans le cas ou la polarité de la classe est négative, on déclenche un conflit.

Fusion de deux prédicats. Il s'agit d'un cas plus complexe à traiter, car il nécessite en plus des opérations décrites plus haut pour le cas d'une égalité positive, de discriminer la polarité des deux classes à fusionner. Intuitivement, c'est à ce moment là que la fonction ligne 5 de la figure 4.4 entre en action. En effet par défaut tous les termes ont une polarité neutre, cependant il peut arriver que sous l'effet d'une contrainte propagée par une relation de congruence, une classe avec une polarité non neutre doive être fusionnée avec une classe neutre. Ceci implique

```
1  static void
2  CCS_deduce_cong(ccpar_u, ccpar_v)
3  {
4    unsigned i, j;
5    Tnode curU,curV;
6    if(CCS_status != UNSAT && stack_size(ccpar_v) &&
       stack_size(ccpar_u))
7      {
8        for (i = 0; i < stack_size(ccpar_u); i++) /**< Deduce rule */
9          {
10           for (j = 0; j < stack_size(ccpar_v); j++)
11             {
12               curU = stack_get(ccpar_u, i);
13               curV = stack_get(ccpar_v, j);
14               if (cong(curU,curV) && !are_connected(curU,curV))
15                 create_cong_deduction(curU, curV);
16             }
17         }
18       stack_reset(ccpar_u);
19       stack_reset(ccpar_v);
20     }
21 }
```

FIGURE 4.6 – Représentation des arêtes du graphe d'égalités

donc une propagation complexe des polarités entre les classes qui doit être contrôlée dans le but de détecter toute incohérence. Par conséquent, la fonction `set_bvalue` joue un double rôle, le premier consiste à assigner la polarité de chaque prédicat asserté positivement, le second de détecter toute incohérence entre les polarités. Ainsi, avant de fusionner deux classes d'équivalence dont les polarités sont contraires, la fonction `set_bvalue` va discriminer deux cas. Le premier cas correspond à la propagation des polarités, c'est lorsque l'une des deux classes à fusionner est neutre. Dans ce cas de figure il suffit simplement de propager la polarité. Le second cas est le cas problématique où les polarités des deux classes sont non neutres, et différentes l'une de l'autre. En d'autre termes c'est le cas où l'on cherche à fusionner une classe avec une polarité négative avec une classe dont la polarité est positive (ou vice versa), et dans ce cas le l'ensemble E est incohérent. Il faut donc reporter un conflit.

Dans le but de simplifier au maximum la présentation nous n'avons pas parlé de la notion de retour en arrière, qui dans le solveur est indispensable. Intuitivement, pour rendre l'algorithme présenté backtrackable, il faut introduire une notion de trace. Cette trace est une pile, dans laquelle toutes les opérations effectuées sur le graphe d'égalité sont stockées. En substance, si l'on souhaite rendre la procédure présentée dans ce chapitre backtrackable, il faudra ajouter une fonction qui à chaque opération mémorise l'action effectuée dans la pile. Par exemple, à chaque fusion de classe, il faut retenir l'ordre dans lequel les classes sont fusionnées, la polarité, et les

représentants respectifs de chaque classe.

Nous rappelons aussi que plusieurs vérifications et optimisations sont effectuées dans la version implémentée par veriT dans le but de rendre les calculs plus efficaces. Le code peut être consulté pour plus de détails.

4.4 Calculer des explications

Pour s'intégrer au framework CDCL($\mathcal{T}$), présenté dans la section 2.2.2, une procédure de décision doit être capable de fournir des explications. Ces explications se présentent sous la forme de clauses de conflits qui incriminent les littéraux à l'origine de l'incohérence dans E. Le calcul de la plus petite clause de conflit est un problème NP-complet [53]. Pour cette raison l'algorithme que nous donnons dans cette section ne calcule pas la plus petite explication, mais une petite explication, qui, sans être nécessairement de taille minimale, permet en pratique au solveur de fonctionner de façon efficace.

Dans l'algorithme proposé par Nieuwenhuis et Oliveras [85], le calcul des explications fait appel à une seconde structure de données maintenue en parallèle à la structure d'union-find nécessaire au calcul de la fermeture de congruence. Cette architecture a l'avantage de préserver une complexité intéressante ($\mathcal{O}(n \log n)$) pour le calcul des explications, cependant elle a l'inconvénient d'être complexe à maintenir en particulier lorsque l'on souhaite y ajouter des fonctionnalités particulières. La structure de données présentée dans cette thèse offre une alternative équilibrée entre flexibilité et performance. En particulier l'algorithme ici présenté n'a besoin que d'une seule structure de données pour effectuer à la fois le calcul de la fermeture de congruence et des explications, ce qui permet de grandement réduire la difficulté du code.

En pratique lorsqu'un conflit est reporté par l'algorithme de fermeture de congruence, les deux termes à l'origine du conflit sont mémorisés. Ensuite la clause de conflit est retrouvée grâce à un calcul de plus court chemin sur le graphe d'égalité entre les deux termes en conflit. Il faut considérer deux types de conflits : les conflits liés à des polarités incompatibles, et les conflits liés à la présence d'une arête de diségalité entre deux classes équivalentes. Pour l'intuition, il est suffisant, dans cette thèse, de ne se concentrer que sur le dernier type de conflit. Intuitivement, lorsque une arête de diségalité apparaît entre deux classes il faut être capable de remonter à l'origine de l'incohérence, et pour cela il suffit de suivre les arêtes d'égalité ou de congruence qui connectent les deux termes à l'origine de ce conflit. Étant donné que l'on désire une seule explication, il est suffisant de calculer le plus court chemin entre ces deux termes. L'algorithme permettant de calculer les explications est donné en figure 4.7. Cet algorithme utilise une pile `explain_pending`, pour stocker les nœuds pour lesquels il est nécessaire de calculer une explication. À l'initialisation cette fonction est appelée avec les deux termes mémorisés à l'apparition du conflit. La fonction `explain_eq_aux` appelée ligne 12, permet de construire la clause de conflit. La clause de conflit finale est obtenue lorsque la pile `explain_pending` est vide, c'est à dire, lorsque toutes les égalités, y compris les congruences, ont été expliquées à partir des égalités du

problème intial.

```
1  static void
2  explain_eq(Tnode u, Tnode v)
3  {
4    Tstack_node explain_pending;
5    stack_INIT(explain_pending);
6    stack_push(explain_pending, u);
7    stack_push(explain_pending, v);
8    while (stack_size(explain_pending))
9      {
10       v = stack_pop(explain_pending);
11       u = stack_pop(explain_pending);
12       if (u != v) explain_eq_aux(u, v, &explain_pending);
13     }
14   stack_free(explain_pending);
15 }
```

FIGURE 4.7 – Algorithme d'explication

La clause de conflit est construite incrémentalement, par la fonction `explain_eq_aux` représentée à la figure 4.8. Notons que chaque arête d'égalité mémorise le littéral associé dans E. Mais lorsque l'on traverse une arête de congruence il faut être capable aussi d'expliquer l'égalité des deux termes à partir des littéraux dans E. Pour calculer le pus court chemin la fonction fait appel à une fonction `BFS`, qui remplit le tableau de prédécesseurs global `pred_explain`. Ce tableau permet de remonter l'ensemble des prédécesseurs entre v et u, où `stack_get(pred_explain, u) == -1`. Ligne 9, on regarde l'arête en cours. Si cette arête est de type égalité alors ligne 10, on ajoute ce littéral à la clause de conflit contenue dans `veriT_conflict`. Autrement cela signifie que l'arête visitée est une arête de congruence, et donc, ligne 13, l'appel à la fonction `explain_cong` permet d'extraire une séquence de littéraux impliquant cette congruence. Ligne 14, on passe au prédécesseur suivant, dans le tableau `pred_explain`.

L'algorithme donné dans la figure 4.9 permet d'extraire les séquences qui impliquent une arête de congruence. Intuitivement, si deux termes $u = u_1\ u_2$ et $v = v_1\ v_2$ sont reliés dans le graphe d'égalité par une arête de congruence, cela signifie que $u_1 \simeq v_1$ et $u_2 \simeq v_2$. Par conséquent l'arête de congruence est la conséquence de ces équivalences. En d'autres termes cela signifie qu'il faut pouvoir expliquer au SAT solveur au moyen d'une clause de conflit et des litéraux présents dans E pourquoi u et v sont équivalents, et pour cela il suffit de retrouver les séquences de littéraux associés à $u_1 \simeq v_1$ et $u_2 \simeq v_2$. C'est exactement ce que la fonction `explain_cong` réalise. Elle empile deux contraintes supplémentaires dans la pile `explain_pending`, la première $u_1 \simeq v_1$, ligne 6 et 7 ; puis la seconde $u_2 \simeq v_2$, ligne 11 et 12. Ces contraintes seront traitées plus tard par la fonction `explain_eq`, donnée plus haut figure 4.7. En pratique nous prenons bien sûr des précautions supplémentaires notamment, nous faisons attention de ne pas introduire d'explications redondantes.

```
1  static void
2  explain_eq_aux(Tnode u, Tnode v, Tstack_node *explain_pending)
3  {
4    Tlit lit;
5    Tnode traveler = v;
6    BFS(u);
7    while (stack_get(pred_explain, traveler) != -1)
8    {
9      lit = get_eq_lit(traveler, stack_get(pred_explain, traveler));
10     if (lit) stack_push(veriT_conflict, lit);
11     else if (type_node(traveler) == APP &&
12              type_node(stack_get(pred_explain, traveler)) == APP)
13       explain_cong(traveler, stack_get(pred_explain, traveler),
             explain_pending);
14     traveler = stack_get(pred_explain, traveler);
15   }
16   clean_visited_node();
17 }
```

FIGURE 4.8 – Construction de la clause de conflit

4.5 conclusion

Dans ce chapitre, nous avons présenté une partie de l'implémentation du ground solveur pour la théorie de l'égalité en place dans le solveur SMT veriT. Cet algorithme permet en outre de traiter des formules sans quantificateur de la logique d'ordre supérieur sans λ-expression. Cette implémentation a une complexité quadratique dans le pire des scénarios et comme nous l'avons observé dans le chapitre précédent (section 3.8) dans la pratique cet algorithme est cinq fois moins rapide que l'implémentation optimale de la fermeture de congruence initialement utilisée dans veriT pour traiter des formules de la logique du premier ordre. La description de l'ensemble des algorithmes donnée dans cette section permet de s'assurer intuitivement de leur correction par construction. En effet dans la présentation de l'algorithme donnée ci-dessus nous avons décrit chacune des entrées possible et le traitement appliqué sur ces entrées. Ainsi pour chaque nouveau littéral introduit dans la fermeture de congruence l'algorithme est capable de déterminer si l'ensemble des littéraux en cours de traitement est satisfaisable ou non. En effet la construction de classe d'équivalence en utilisant des graphes permet de se convaincre de la correction de l'algorithme de manière assez intuitive. Cette implémentation permet donc d'exprimer des expressions de la logique d'ordre supérieur de manière assez naturel, mais peut aussi être une base solide pour des logiques plus riches. En particulier, il est possible d'implémenter assez facilement la théorie des data-types, mais aussi de traiter les symboles de fonction injective de manière efficace. Enfin cet algorithme permet une meilleure collaboration avec le module d'instanciation et peut notamment être utilisé pour effectuer l'algorithme de E-unification utilisé par le calcul CCFV. Une implémentation du solveur SMT veriT s'appuyant sur cet algorithme pour effectuer

```
1  static void
2  explain_cong(Tnode u, Tnode v, Tstack_node *explain_pending)
3  {
4    if (!already_seen(get_left(u), get_left(v)))
5      {
6        stack_push(*explain_pending, get_left(u));
7        stack_push(*explain_pending, get_left(v));
8      }
9    if (!already_seen(get_right(u), get_right(v)))
10     {
11       stack_push(*explain_pending, get_right(u));
12       stack_push(*explain_pending, get_right(v));
13     }
14 }
```

FIGURE 4.9 – Expliquer les congruences

l'instanciation des quantificateurs (via une version de CCFV modifiée) est aussi disponible sur le dépôt [5].

5. https://github.com/delouraoui/these

5

Améliorer l'instanciation via des méthodes d'apprentissage

Sommaire

5.1 Introduction

Comme vu au chapitre 2, les solveurs SMT sont des outils particulièrement efficaces pour raisonner, avec des théories, sur des formules de la logique du premier ordre. Lorsque les problèmes contiennent des quantificateurs, les solveurs SMT s'appuient sur le processus d'instanciation. L'instanciation dérive un ensemble d'instances à partir des formules quantifiées. Plus précisément, les variables liées aux quantificateurs, dans les formules, sont remplacées par des termes

qui apparaissent généralement dans la fermeture de congruence. Pour instancier ces formules plusieurs stratégies peuvent être employées : l'approche par énumération [95], l'approche par trigger [41, 45], l'approche par conflit [97], ou encore l'approche par construction de modèle [55]. Parmi ces stratégies, seule l'approche par conflit permet de calculer efficacement un ensemble d'instances permettant d'éliminer le modèle E, produit en collaboration avec le SAT solveur et les procédures de décision. Cette approche est néanmoins incomplète, et nécessite d'être utilisée en combinaison avec d'autres méthodes comme les approches par trigger, ou par énumération. Malheureusement ces dernières sont très heuristiques, et génèrent un grand nombre d'instances. En conséquence, l'espace de recherche du solveur peut vite exploser. Ainsi, pour résoudre rapidement des problèmes avec quantificateurs, il faut être capable de trouver rapidement les bonnes instances, c'est-à-dire réduire le nombre d'instances inutiles données au solveur ground. Il est donc impératif d'améliorer l'efficacité du module d'instanciation, et pour cela nous proposons dans ce chapitre de sélectionner les instances produites par les approches par trigger et énumération, à l'aide d'un algorithme de classification entraîné en amont sur plusieurs exemples de bonnes et mauvaises instances produites par le module d'instanciation.

L'apprentissage automatique est un domaine en plein essor. Popularisé par les algorithmes d'apprentissages profonds, et notamment très utilisé pour les systèmes de vision, l'apprentissage automatique peut aussi être appliqué à de nombreux autres domaines. Polyvalents, ces algorithmes ont la particularité d'apprendre une tache sans aucune connaissance préalable du domaine d'application, pour peu que ce dernier soit mathématiquement représentable. En effet à l'aide d'encodages spécifiques, ces algorithmes sont capables de comprendre et d'apprendre le fonctionnement d'un système. On distingue d'ailleurs deux types d'apprentissages : supervisé qui consiste à entraîner un modèle à partir d'un certain nombre d'observations étiquetées, et non supervisé où l'algorithme doit découvrir la structure du problème sans indication préalable. Récemment plusieurs solveurs automatiques, incorporant dans leur processus de recherche des algorithmes d'apprentissage, ont démontré leur efficacité face à des solveurs de dernière génération. En particulier des systèmes comme Enigma et ATPBoost [67, 94] permettent de modérer le processus de preuve, et de suggérer des choix au solveur. Ces approches ont été majoritairement appliquées à des architectures basées sur des systèmes de preuves de type superposition ou tableaux. Dans ce travail nous nous inspirons de ces travaux pour concevoir une approche par instanciation coopérant avec des algorithmes d'apprentissage supervisé, et dont le but est d'améliorer l'instanciation des quantificateurs au sein du SMT solveur veriT.

Le travail présenté dans ce chapitre est initialement motivé par la visite à Nancy de Cezary Kaliszyk. Ce travail est donc le fruit d'une collaboration étroite avec mes encadrants Jasmin Blanchette et Pascal Fontaine pour la partie raisonnement automatique et SMT, et Cezary Kaliszyk qui a apporté son expertise dans le domaine de l'apprentissage automatique, en particulier pour la preuve automatique. L'idée générale de notre approche consiste à invoquer un classifieur après chaque cycle d'instanciation pour évaluer l'utilité potentielle de chaque instance. En fonction des prévisions, les instances sont soit introduites au problème original, retardées ou rejetées. Étant donné que l'instanciation des quantificateurs implique plusieurs milliers d'instances, le prédicteur doit être rapide. Par conséquent, il est important de choisir une approche simple à mettre en

œuvre, avec des temps d'apprentissage et de prédictions rapides. XGBoost [34] est un algorithme de classification binaire qui répond à ces critères, et dont le principe de fonctionnement repose sur des arbres de décision. Nous proposons donc dans ce chapitre un prototype encodant le problème de l'instanciation comme un vecteur d'entiers pouvant être exploité par l'algorithme de classification XGBoost. Les prédictions sont ensuite utilisées pour déterminer la pertinence des instances dérivées par le module d'instanciation.

Nous avons mené nos expériences dans veriT [31], un solveur SMT qui met en œuvre les techniques d'instanciation mentionnées ci-dessus, à l'exception de l'approche par construction de modèles. Une implémentation, avec le code source, sous une licence permissive a été développée. Une évaluation expérimentale, du prototype est présentée en section 5.4. Ces évaluations montrent notamment que le nombre d'instances introduites dans le solveur est considérablement réduit grâce à cette approche. Nous montrons en outre que notre prototype permet d'augmenter le taux de réussite pour les problèmes SMT-LIB (la plus grande bibliothèque de problèmes pour SMT), permettant à notre prototype mis en œuvre dans veriT d'égaler en efficacité, après apprentissage, les meilleurs solveurs de sa catégorie.

Les informations transmises au prédicteur sont appelées *features*, et sont décrites section 5.3.2. L'encodage du problème de l'instanciation dans le SMT solveur veriT est décrit en section 5.3.3. Dans la section 5.3.6, nous précisons quand, où, et dans quelle mesure ces techniques sont utilisées ; ce sont des détails importants pour une intégration harmonieuse et productive des techniques dans le solveur.

5.2 Contexte

La notation $t[\bar{x}_n]$ représente un terme dont les variables libres sont incluses dans le tuple de variables distinctes $\bar{x}_n$. Le terme $t[\bar{s}_n]$ est alors le terme obtenu à partir de t par une substitution simultanée de $\bar{x}_n$ par $\bar{s}_n$. Le symbole x désigne un vecteur de features, et la notation $\mathsf{x}[i]$ représente le i-ème élément du vecteur.

5.2.1 Petits rappels sur SMT

Comme vu dans le chapitre 2, le cœur d'un solveur SMT (Fig. 1.1) est constitué d'un solveur SAT [24], et de procédures de décision lui permettant de travailler avec des logiques plus expressives. (Voir [13] pour plus d'informations sur SMT.) Schématiquement, le solveur SMT prend en entrée une formule de la logique du premier ordre. Cette formule est tout d'abord abstraite en une formule booléenne, puis donnée à un solveur SAT. Le solveur SAT fournit un modèle pour cette abstraction booléenne, correspondant au premier ordre à un ensemble de littéraux, dont la consistance est ensuite vérifiée par les procédures de décision. Cette architecture permet aux procédures de décision de n'avoir à vérifier la satisfaisabilité que d'ensembles conjonctifs

de littéraux, plutôt que de combinaisons booléennes arbitraires de littéraux. Si l'ensemble des littéraux n'est pas satisfaisable, l'abstraction booléenne est affinée par l'ajout d'une clause propositionnelle de conflit dans le solveur SAT, et le processus se répète. Si l'ensemble des littéraux est satisfaisable, alors la formule est également satisfaisable, et un modèle peut être produit. Si la théorie est décidable, le solveur de théorie termine toujours, et si nous supposons en outre que les clauses de conflit ne contiennent que des variables booléennes abstraites de la formule d'entrée, l'ensemble du processus s'achève après l'ajout d'un nombre fini de clauses de conflit. Si la formule est non satisfaisable, une preuve peut être fournie.

Exemple 5.2.1. Si l'on considère une formule de la logique du premier ordre sans quantificateur telle que

$$\mathsf{a} \simeq \mathsf{b} \wedge \big(\mathsf{f}(\mathsf{a}) \not\simeq \mathsf{f}(\mathsf{b}) \vee (\mathsf{R}(\mathsf{a}) \wedge \neg \mathsf{R}(\mathsf{b}))\big).$$

La formule ci-dessus est abstraite en la formule propositionnelle suivante par le solveur SMT :

$$p_{\mathsf{a}\simeq\mathsf{b}} \wedge \big(\neg p_{\mathsf{f}(\mathsf{a})\simeq\mathsf{f}(\mathsf{b})} \vee \big(p_{\mathsf{R}(\mathsf{a})} \wedge \neg p_{\mathsf{R}(\mathsf{b})}\big)\big).$$

Dans la formule ci-dessus, p_ℓ exprime l'abstraction booléenne de l'atome ℓ. Le SAT solveur peut déduire à partir de cette formule le modèle propositionnel suivant : $\{p_{\mathsf{a}\simeq\mathsf{b}}, \neg p_{\mathsf{f}(\mathsf{a})\simeq\mathsf{f}(\mathsf{b})}\}$. Étant donné que l'ensemble de littéraux $\{\mathsf{a} \simeq \mathsf{b}, \mathsf{f}(\mathsf{a}) \not\simeq \mathsf{f}(\mathsf{b})\}$ est non satisfaisable, la procédure de décision, pour la théorie de l'égalité, produit la clause de conflit $\neg p_{\mathsf{a}\simeq\mathsf{b}} \vee p_{\mathsf{f}(\mathsf{a})\simeq\mathsf{f}(\mathsf{b})}$ permettant d'informer le solveur SAT que le modèle propositionnel ne s'accorde avec la théorie de l'égalité. Le solveur SAT génère un nouveau modèle $\{p_{\mathsf{a}\simeq\mathsf{b}}, p_{\mathsf{R}(\mathsf{a})}, \neg p_{\mathsf{R}(\mathsf{b})}\}$, qui de la même façon sera réfuté par la théorie de l'égalité. Comme plus aucun modèle propositionnel ne peut être produit, le solveur SMT conclut que la formule n'est pas satisfaisable. •

Le processus ci-dessus décrit le fonctionnement interne d'un solveur SMT pour les problèmes sans quantificateur, c'est à dire le solveur *ground*. Il n'est pas utile de creuser ici plus en détail mais il est important pour se rappeler que le solveur ground abstrait les atomes en propositions booléennes. Si la formule d'entrée est satisfaisable, le solveur ground propose une conjonction satisfaisable de littéraux. Lorsque les formules contiennent des quantificateurs, ces formules quantifiées sont également abstraites en tant que propositions booléennes. Mais pour les traiter, il est nécéssaire d'ajouter une boucle de raffinement autour du solveur ground.

5.2.2 Instanciation dans SMT

Le raisonnement avec quantificateurs est donc traité par une couche supplémentaire (voir le côté gauche de Fig. 2.5). Plus exactement, le module d'instanciation, est placé au dessus du solveur SMT sans quantificateur. Le solveur sans quantificateur ne voit qu'une abstraction sans quantificateur de l'entrée, c'est-à-dire, l'entrée où les formules quantifiées sont réduites à de nouvelles formules propositionnelles, après la skolemisation. Le module d'instanciation dans un solveur SMT se base sur les travaux fondateurs de Skolem, et de Herbrand pour instancier les formules, historiquement introduits dans les systèmes [39, 56]. Le solveur ground produit d'abord

une conjonction de littéraux où les formules quantifiées apparaissent comme des propositions. Ensuite, le module d'instanciation produit des instances des formules quantifiées avec des termes provenant de l'univers de Herbrand (c'est-à-dire tous les termes ground possibles sur la signature de la formule). Ces instances permettent par conséquent d'affiner l'abstraction ground de la formule. Formellement, étant donné un ensemble de littéraux ground E, et un ensemble de formules quantifiées Q, le *problème de l'instanciation* consiste à trouver un ensemble d'instances I de Q telles que $E \cup I \models \bot$. Il s'agit d'un problème semi-décidable dans la logique du premier ordre avec l'égalité. De nombreuses approches ont été développées pour s'attaquer à ce problème, et le solveur SMT veriT en implémente quelques-unes : l'approche par énumération [95], par triggers [41, 45] et par conflits [97] (voir 2.3 pour plus de détails).

5.2.3 À la recherche des bonnes instances

L'instanciation par triggers et par énumération sont des approches très heuristiques, qui génèrent un nombre important d'instances. Malheureusement, la taille du problème que le solveur SMT doit résoudre, ainsi que le nombre de modèles propositionnels possibles croissent proportionnellement au nombre de nouvelles instances, qui ne sont pas des instances de conflits, introduites par ces techniques. Évidemment ces approches peuvent dériver quelquefois des instances de conflits, cependant la proportion des instances qui sont de conflit est faible. Ainsi, réduire le nombre d'instances non pertinentes, générées à partir de ces approches, permettrait de prévenir ces effets, avec pour conséquence une amélioration de l'efficacité, et une augmentation du nombre de problèmes résolus. L'apprentissage supervisé est une méthode applicable pour le filtrage de ces instances.

Pour être employé, l'apprentissage supervisé, et en particulier pour des problèmes de classification binaires, nécessite une base d'apprentissage qui se compose d'observations étiquetées comme bonnes ou mauvaises. Pour construire cette base d'observations, dans notre contexte, il est nécessaire de classer chaque instance produite par le module d'instanciation. Pour déterminer si une instance est pertinente — c'est-à-dire si elle est utile pour résoudre un problème — la preuve produite par le solveur peut être inspectée a posteriori. En effet, le solveur veriT produit des preuves avec un haut niveau de détail. Il est par ailleurs facile d'élaguer de ces preuves les lemmes ou les instances qui n'ont pas été utilisées pour déduire l'inconsistance. Et par conséquent, il est donc facile, en examinant une preuve, de déterminer avec une bonne approximation quelles instances sont utiles.

Statistiquement, si l'on compare le nombre d'instances produites lors d'une exécution typique du solveur, et le nombre d'instances qui figurent dans la preuve élaguée, on constate que seulement 10% des instances produites (et seulement 1% des instances générées le sont par la stratégie d'instanciation par conflit) font partie de la preuve élaguée. En d'autres termes, en moyenne, seulement 10% des formules ajoutées par le module d'instanciation sont réellement utiles. Un bon algorithme de classification pourrait donc aider à éliminer une partie des 90% de formules générées inutiles. Illustrons cela sur un problème simple — une version simplifiée du problème

`UF/misc/set10.smt2`, provenant de la SMT-LIB — qui servira également d'exemple courant.

Exemple 5.2.2. Considérons l'ensemble d'axiomes suivant, où $\sqcup, \sqcap, \sqsubset, \in$ sont des symboles non interprétés représentant abstraitement les fonctions d'union, d'intersection, et prédicats d'inclusion et d'adhésion. Pour des raisons de lisibilité, la notation infixe est utilisée.

$$\begin{aligned}
\varphi_1 &= \forall xyz\,.\, x \in y \wedge y \sqsubset z \Rightarrow x \in z \\
\varphi_2 &= \forall xy\,.\, \neg(x \sqsubset y) \Rightarrow \exists z\,.\, z \in x \wedge \neg(z \in y) \\
\varphi_3 &= \forall xyz\,.\, x \in (y \sqcup z) \simeq (x \in y \vee x \in z)
\end{aligned}$$

Supposons en outre un ensemble de littéraux ground $E = \{\neg((\mathsf{a} \sqcup \mathsf{b}) \sqsubset \mathsf{c}), \mathsf{a} \sqsubset \mathsf{c}, \mathsf{b} \sqsubset \mathsf{c}\}$ où $\mathsf{a}, \mathsf{b}, \mathsf{c}$ sont des symboles constants, et un ensemble de formules quantifiées $Q = \{\varphi_1, \varphi_2, \varphi_3\}$. Nous voulons résoudre le problème d'instanciation $E \cup I \models \bot$ où I est l'ensemble des instances de base de Q qui doivent être générées. Il n'est pas difficile de montrer que $E \cup Q$ est insatisfaisant, mais c'est déjà un cas intéressant pour analyser le comportement général du solveur. Le tableau 5.1 décrit les tours successifs d'instanciation qui pourraient se produire dans le solveur SMT.

Tour	E_i	φ_i	σ		I
1	$\{\neg((\mathsf{a} \sqcup \mathsf{b}) \sqsubset \mathsf{c}),\ \mathsf{a} \sqsubset \mathsf{c},\ \mathsf{b} \sqsubset \mathsf{c}\}$	φ_2	$\sigma_{2,1}$	$x \mapsto \mathsf{a},\ y \mapsto \mathsf{c}$	$\varphi_2\sigma_{2,1}$
			$\sigma_{2,2}$	$x \mapsto \mathsf{b},\ y \mapsto \mathsf{c}$	$\varphi_2\sigma_{2,2}$
			$\sigma_{2,3}$	$x \mapsto (\mathsf{a} \sqcup \mathsf{b}),\ y \mapsto \mathsf{c}$	$\varphi_2\sigma_{2,3}$
2	$E_1 \cup \{\mathsf{sk} \in (\mathsf{a} \sqcup \mathsf{b}),\ \neg(\mathsf{sk} \in \mathsf{c})\}$	φ_2	$\sigma_{2,4}$	$x \mapsto \mathsf{a},\ y \mapsto \mathsf{c}$	$\varphi_2\sigma_{2,4}$
			$\sigma_{2,5}$	$x \mapsto \mathsf{b},\ y \mapsto \mathsf{c}$	$\varphi_2\sigma_{2,5}$
			$\sigma_{2,6}$	$x \mapsto (\mathsf{a} \sqcup \mathsf{b}),\ y \mapsto \mathsf{c}$	$\varphi_2\sigma_{2,6}$
		φ_3	$\sigma_{3,1}$	$x \mapsto \mathsf{sk},\ y \mapsto \mathsf{a},\ z \mapsto \mathsf{b}$	$\varphi_3\sigma_{3,1}$
3	$E_2 \cup \{\neg(\mathsf{sk} \in \mathsf{b}),\ \mathsf{sk} \in \mathsf{a}\}$	φ_1	$\sigma_{1,1}$	$x \mapsto \mathsf{sk},\ y \mapsto \mathsf{a},\ z \mapsto \mathsf{c}$	$\varphi_1\sigma_{1,1}$
4	$E_2 \cup \{\mathsf{sk} \in \mathsf{b},\ \neg(\mathsf{sk} \in \mathsf{a})\}$	φ_1	$\sigma_{1,2}$	$x \mapsto \mathsf{sk},\ y \mapsto \mathsf{b},\ z \mapsto \mathsf{c}$	$\varphi_1\sigma_{1,2}$

TABLE 5.1 – Tours d'instanciation de $E \cup Q$

La colonne E_i correspond à l'ensemble de littéraux produits par le solveur ground au tour i, φ_i est une formule quantifiée provenant de Q, σ correspond à la substitution dérivée au cours du processus d'instanciation, et I correspond aux instances produites à partir des substitutions dérivées par le module d'instanciation. Les instances générées à partir des substitutions $\sigma_{1,1}$ et $\sigma_{1,2}$ sont des instances de conflit, toutes les autres sont générées soit par l'approche par trigger ou par énumération. Les instances associées à la formule φ_2 contiennent un quantificateur existentiel, ces formules sont donc skolemisées avant d'être passées au ground solveur. La constante sk provient de la skolemisation de l'instance $\varphi_2\sigma_{2,3}$; pour simplifier, nous ne mentionnons pas les autres constantes de Skolem qui sont générées pour les autres instances (inutiles). Il est aussi important de souligner que les tours 3 et 4 sont déclenchés par un branchement conditionnel du solveur SAT sur les littéraux : $\mathsf{sk} \in \mathsf{b}$, $(\mathsf{sk} \in \mathsf{a})$. Finalement, au quatrième tour d'instanciation, l'instance $\mathsf{sk} \in \mathsf{b} \wedge \mathsf{b} \sqsubset \mathsf{c} \Rightarrow \mathsf{sk} \in \mathsf{c}$ est produite, rendant le problème incohérent au niveau du solveur ground.

En regardant la colonne des instances à droite, on peut observer que certaines instances sont redondantes (par exemple, $\sigma_{2,4}, \sigma_{2,5}, \sigma_{2,6}$) ou inutiles (par exemple, $\sigma_{2,1}, \sigma_{2,2}$) pour résoudre le problème. Une preuve élaguée ne contient que les instances $\sigma_{2,3}, \sigma_{3,1}, \sigma_{1,2}, \sigma_{1,1}$ (voir tableau 5.2). Un solveur SMT idéal ne ferait donc qu'instancier les formules du tableau 5.2, et éviterait d'inonder le solveur ground avec des instances inutiles ou redondantes. •

TABLE 5.2 – Tours d'instanciation (preuve élaguée) de $E \cup Q$

Round	E_i	φ_i	σ		I
1	$\{\neg((\mathsf{a} \sqcup \mathsf{b}) \sqsubset \mathsf{c}), \mathsf{a} \sqsubset \mathsf{c}, \mathsf{b} \sqsubset \mathsf{c}\}$	φ_2	$\sigma_{2,3}$	$x \mapsto (\mathsf{a} \sqcup \mathsf{b}), y \mapsto \mathsf{c}$	$\varphi_2\sigma_{2,3}$
2	$E_1 \cup \{\mathsf{sk} \in (\mathsf{a} \sqcup \mathsf{b}), \neg(\mathsf{sk} \in \mathsf{c})\}$	φ_3	$\sigma_{3,1}$	$x \mapsto \mathsf{sk}, y \mapsto \mathsf{a}, z \mapsto \mathsf{b}$	$\varphi_3\sigma_{3,1}$
3	$E_2 \cup \{\neg(\mathsf{sk} \in \mathsf{b}), \mathsf{sk} \in \mathsf{a}\}$	φ_1	$\sigma_{1,1}$	$x \mapsto \mathsf{sk}, y \mapsto \mathsf{a}, z \mapsto \mathsf{c}$	$\varphi_1\sigma_{1,1}$
4	$E_2 \cup \{\mathsf{sk} \in \mathsf{b}, \neg(\mathsf{sk} \in \mathsf{a})\}$	φ_1	$\sigma_{1,2}$	$x \mapsto \mathsf{sk}, y \mapsto \mathsf{b}, z \mapsto \mathsf{c}$	$\varphi_1\sigma_{1,2}$

Bien qu'il soit assez facile de filtrer les instances redondantes, en vérifiant simplement si l'instance a déjà été ajoutée au solveur ground, reconnaître et éliminer les instances non pertinentes n'est pas une tâche triviale. Nous montrons dans la suite de ce chapitre comment entraîner, et utiliser un modèle prédictif, grâce à un algorithme d'apprentissage, pour résoudre ce problème.

5.3 Une approche par apprentissage pour l'instanciation

Dans le but d'appliquer, au mieux, les algorithmes d'apprentissage automatique au problème de l'instanciation dans le solveur SMT veriT, il nous faut répondre à plusieurs problématiques. Premièrement, les données récoltées pour entraîner les modèles prédictifs sont très déséquilibrées : généralement il y a dix fois plus d'instances non pertinentes que d'instances utiles. Deuxièmement, le nombre d'exemples disponibles est assez faible, une centaine de milliers d'exemples extraits d'un petit ensemble de problèmes, ce qui complique considérablement le processus d'apprentissage. L'ensemble des données récoltées pour entraîner notre modèle prédictif est réduit aux problèmes disponibles dans le SMT-LIB. Troisièmement, le coût en temps de chaque prédiction doit être faible. En effet le module d'instanciation peut être amené à introduire un nombre important d'instances, à chaque cycle d'instanciation, il est donc nécessaire que l'étape de filtrage soit la plus rapide possible afin de ne pas passer plus de temps à trier les instances qu'à résoudre le problème en lui-même.

Une contrainte supplémentaire, intrinsèquement liée aux contraintes énoncées précédemment, nous oblige à restreindre le domaine de notre encodage vers l'algorithme d'apprentissage, aux entiers naturels. En effet parmi les nombreux algorithmes d'apprentissage supervisés existants, ce sont les approches par arbres de décision, et plus précisément la méthode XGBoost [34], qui répond le mieux à ces exigences. Cette approche à d'ailleurs déjà démontré son efficacité, et son utilité dans d'autres proveurs automatiques [67, 94].

Nous allons donc dans la suite décrire comment un algorithme d'apprentissage, de type XG-

Boost, peut être intégré pour guider le module d'instanciation dans un processus de sélection d'instances, dans notre solveur SMT veriT.

5.3.1 Encoder SMT comme un problème de classification

Les algorithmes traditionnels d'apprentissage travaillent à partir de connaissances, appelées *features*, des valeurs numériques qui caractérisent les entrées. Le rôle de l'algorithme d'apprentissage est donc, essentiellement, d'identifier, et de classer en régions chaque entrée. Les features sont des éléments cruciaux qui relient l'application, ici le problème de l'instanciation, à l'algorithme d'apprentissage automatique. Adapter ces algorithmes d'apprentissage à de nouvelles applications se résume donc à représenter la somme des connaissances du domaine d'application à un vecteur de features dans un espace euclidien, et à une recherche de paramètres appropriés pour les algorithmes. L'objectif des features est de fournir une représentation aussi fidèle que possible de notre problème d'entrée pour l'algorithme d'apprentissage. Parfois, il n'est pas possible d'être fidèle au problème original, et ce pour de nombreuses raisons techniques. Dans ce cas, il est nécessaire d'approcher le problème en utilisant des données qui approximent le comportement de l'entrée. Par exemple, il n'existe pas d'approche discrète permettant d'encoder de manière fidèle les formules de la logique du premier ordre. Le coeur de cette problématique avec cette approche réside dans le fait qu'il ne faudrait pas que deux formules similaires soient considérées totalement différentes une fois encodées (comme deux features distinctes). Ce qui en pratique est très difficile à assurer. Idéalement pour éviter tout effet de sur-apprentissage, il serait plus efficace de transmettre à l'algorithme des informations sur les termes telles que la profondeur, la taille des termes, la longeur des symboles, etc. plutôt que les termes tels qu'ils apparaîssent dans la formule. Malheureusement, et ce après de multiples tentatives, aucune combinaison efficace de tels paramètres n'a pu être déterminée lors de nos expérimentations, pour le filtrage des instances, dans le solveur SMT veriT. Par conséquent nous avons choisi d'utiliser une approche hybride se basant principalement sur un encodage syntaxique, et une combinaison de paramètres génériques. Dans la section suivante nous développons cette idée plus en profondeur, et suggérons une approche permettant d'encoder les formules provenant du solveur SMT comme des vecteurs d'entiers.

5.3.2 Concevoir les features

La caractérisation, par l'usage de vecteur de *features*, du problème de l'instanciation vers l'algorithme d'apprentissage nécessite un travail préliminaire de modélisation. De précédents travaux [69] ont eu pour objet l'étude des différents types de features, dans le contexte de la preuve automatique, qui ont notamment été utilisées en combinaison avec des approches d'apprentissage basées sur des arbres de décision [94]. Bien qu'efficaces, ces approches ne peuvent pas être directement utilisées pour les solveurs SMT, car elles sont principalement développées pour des architectures de solveurs basées sur des systèmes de saturations, tels que les calculs de tableaux ou encore de superposition. L'architecture des solveurs SMT étant assez éloignée de

ces solveurs, il est nécessaire de repenser la modélisation du problème, tout en s'inspirant des précédentes études. Et pour cause, les features développées dans ce travail sont plus particulièrement inspirées de celles utilisées dans ENIGMA [67] et RLCoP [68]. Cette section présente l'encodage des termes, des formules, et des instances satisfaisables du problème original dans un espace vectoriel des features, à traiter par le classifieur.

Un solveur SMT manipule des termes de la logique du premier ordre, qui de manière interne sont représentés sous forme de graphes acycliques orientés, aussi appelés DAG (directed acyclic graphs). Cette représentation à l'avantage d'optimiser la représentation en mémoire des problèmes. Via cette représentation, certains termes peuvent partager le même espace mémoire, notamment lorsque certains sous-termes sont dupliqués : c'est en quelque sorte une représentation compressée des arbres de syntaxe abstraite. Le problème d'instanciation se compose naturellement de termes de la logique du première ordre. Ainsi, pour nourrir l'algorithme d'apprentissage des informations provenant du solveur SMT il est nécessaire de déterminer une représentation permettant d'exprimer les termes dans un formalisme adapté à l'algorithme. Malheureusement cette étape n'est pas sans obstacle. Le choix de la méthode d'apprentissage, comme évoqué précédemment, est conditionné par plusieurs contraintes provenant du solveur SMT, et même si l'approche par arbre de décision semble parfaitement répondre aux exigences énoncées plus haut, il y a une difficulté non négligeable qui est la représentation des termes au renommage près sous forme vectorielle. La difficulté peut en effet être contournée lorsque l'encodage permet l'usage de vecteurs réels, comme par exemple avec des systèmes de réseaux de neurones. Dans notre cas les termes doivent pouvoir être représentés de façon plus ou moins uniformes sous forme de vecteurs d'entiers. La contrainte s'exerce donc principalement au niveau des symboles des termes manipulés. Par conséquent, notre représentation ne permet pas à l'algorithme d'apprentissage de déterminer l'équivalence de deux termes au renommage prés. C'est assez dommage car cela prive l'algorithme d'informations utiles, mais c'est une limitation qui est intrinsèquement dépendante de la méthode d'apprentissage choisie.

Malgré cette contrainte importante au niveau des symboles il est possible d'utiliser des méthodes d'apprentissage via des modélisations basées sur l'encodage syntaxique des termes, lorsque les benchmarks de problèmes utilisent un domaine de symboles *cohérents*. C'est-à-dire que la famille de problèmes utilise un domaine de symboles commun, et qu'il n'y a pas (ou très peu de fois) plusieurs définitions du même symbole dans une famille de problèmes. Ce qui est a priori le cas des bibliothèques de preuves, des assistants de preuve comme Isabelle, ou encore Mizar, une fois traduites dans la syntaxe de la SMT-LIB, ou TPTP. Ce sont d'ailleurs les approches utilisées dans le prototype ENIGMA.

L'idée de notre encodage est d'extraire de chaque terme un ensemble de séquences de symboles. Ces séquences de symboles sont obtenues au moyen du parcours en profondeur de chaque terme (on retient les symboles de tous les nœuds visités). Par la suite il est possible de trier ces séquences par longueur. Et c'est en pratique avec des séquences de longueurs un, deux et trois que nous obtenons les meilleurs résultats lors de nos évaluations. Techniquement la combinaison de ces séquences, de trois longueurs différentes, représentent le meilleur compromis en termes

de représentation mémoire, de recouvrement des termes, et de modélisation pour l'algorithme d'apprentissage. C'est un résultat qui d'ailleurs corrobore les travaux de Jakubův et Urban [67], qui utilisent uniquement des séquences de longueur trois.

Les symboles de variables et de Skolem sont souvent sujet à de multiples renommages, et sans notion de portée, ces éléments sont peu informatifs sur la structure des termes. Pour ces raisons, l'encodage obfusque les variables par l'usage d'un symbole particulier $\circledast$. De la même façon, les constantes de Skolem sont remplacées par le symbole $\odot$.

Chacune de ces séquences représentent donc une feature, et le nombre d'occurrences d'une séquence représente la valeur associée à la feature. Une séquence apparaissant plus qu'une autre sera donc considérée comme plus importante ou au contraire comme moins importante par l'algorithme en fonction des différentes observations auxquelles elle sera soumise. Par conséquent, le classifieur utilisera le nombre d'occurrences de chaque feature, plus quelques features supplémentaires, pour classer une instance comme utile ou non. Par exemple, l'encodage du terme $\mathsf{f}(\mathsf{a},\mathsf{b})$ se traduit par l'ensemble de features $\{\mathsf{f},\mathsf{a},\mathsf{b},(\mathsf{f},\mathsf{a}),(\mathsf{f},\mathsf{b})\}$, et par exemple, la valeur de l'élément (f,a) est 1. Les exemples ci-dessous présentent, concrètement, l'encodage pour des termes de la logique du premier ordre.

FIGURE 5.1 – Représentation arborescente du littéral $(x \sqcup \mathsf{sk}) \sqsubset \mathsf{c}$

Exemple 5.3.1. La figure 5.1 montre la représentation arborescente du terme $(x \sqcup \mathsf{sk}) \sqsubset \mathsf{c}$ où x est une variable et sk est un symbole de Skolem. L'arbre de gauche est le terme original et celui de droite est l'arbre traité, après avoir remplacé les variables et symboles de Skolem par leurs symboles respectifs.

Le tableau suivant présente les features extraites de ce terme avec leurs valeurs. La ligne k donne la longueur de chaque séquence de symbole. La valeur est le nombre de fois que la séquence de symboles apparaît dans toutes les séquences de longueur au maximum 3.

k	1	valeur	2	valeur	3	valeur
`features`	$\sqsubset$	1	$(\sqsubset,\sqcup)$	1	$(\sqsubset,\sqcup,\circledast)$	1
	$\sqcup$	1	$(\sqsubset,\mathsf{c})$	1	$(\sqsubset,\sqcup,\odot)$	1
	$\odot$	1	$(\sqcup,\circledast)$	1		
	$\circledast$	1	$(\sqcup,\odot)$	1		
	c	1				

•

FIGURE 5.2 – Représentation arborescente du littéral g (f x) (f y)

Exemple 5.3.2. La figure 5.2 et le tableau suivant montrent un autre exemple de représentation arborescente et de valeurs de features.

k	1	valeur	2	valeur	3	valeur
features	f	2	(g, f)	2	(g, f, ⊛)	2
	g	1	(f, ⊛)	2		
	⊛	2				

•

Pour transformer chaque séquence de symboles, features, en une valeur entière, nous utilisons la fonction de hachage djb2 (voir la figure 5.3), naïve mais suffisante, qui permet d'associer à chaque symbole un entier naturel.

```
1  Int64 hash(char *str)
2  {
3    Int64 hash = 5381;
4    while (*str)
5      hash = ((hash << 5) + hash) + *(str++);
6    return hash;
7  }
```

FIGURE 5.3 – Fonction de hachage.

La dimension des vecteurs de features est fixe, en pratique l'expérience a montrer, que des vecteurs de dimension 2310705, soit 2 MB, est un bon compromis. Cette taille permet de stocker suffisamment de features pour les problèmes rencontrés dans nos expérimentations, ce paramètre peut-être réévaluer pour des problèmes différents. La transformation des séquences de symboles en features est calculée par la formule récursive suivante :

$$u_{n+1} = (u_n \bmod 2310705) * k^{n+1} + s.$$

où u_n est une séquence de longueur n, avec n compris entre 0 et 2, k est un nombre grand premier, et s la valeur de hachage du symbole n de la séquence. Concrètement, lorsqu'une séquence est de longueur 1, de type (f_0), la feature calculée correspond à la valeur de hachage du symbole modulo la dimension. Pour une séquence de longueur 2, de type (f_0, f_1) on calcule la feature comme suit :

$u_1 = (\texttt{hash}(f_0) \bmod 2310705) * k + \texttt{hash}(f_1))$. Et le calcul d'une séquence de symboles de longueur 3, de type (f_0, f_1, f_2), est : $((u_1 \bmod 2310705) * k^2 + \texttt{hash}(f_2))$. Une fois le calcul terminé chaque feature est ramenée dans l'espace de dimension des vecteurs, par l'application de l'opérateur modulo.

En pratique, l'espace de features est segmenté de sorte que les features provenant, par exemple de E, des instances, ou des paramètres génériques (taille, profondeurs des termes) soient distinguées entre elles. Pour cela on utilise un simple décalage lors du calcul des features. Par exemple si la séquence u_1 est extraite de l'ensemble d'égalités E, nous ajoutons un décalage fixe, d_1, multiplié par un certain intervalle : $u_1 + d_1 * 262139$ (ce paramètre a été déterminer par l'expérience pour les problèmes SMT, il peut être réévalué pour d'autres benchmarks). Si au contraire la séquence provient d'une instance, alors la valeur de décalage sera différente. Cette astuce permet donc de partitionner chaque vecteur de façon à ce que l'algorithme puisse identifier la provenance des informations. Dans la section suivante nous étudions plus en détail la modélisation du problème de l'instanciation dans SMT.

5.3.3 Description du problème de l'instanciation

Considérant la caractérisation des termes par des features donnée dans la section précédente, nous allons maintenant étudier la modélisation du problème de l'instanciation pour SMT. L'entrée de l'algorithme d'apprentissage est un vecteur de features. Dans notre contexte nous avons choisi de modéliser le problème en distinguant les trois parties essentielles de notre problème, et pour cela il faut s'intéresser au mode de fonctionnement du module d'instanciation. Comme nous l'avons étudié dans la section 5.2.3, le module d'instanciation se base sur la concrétisation du modèle propositionnel E, fournit par le solveur ground, et la formule quantifiée à instancier. Grâce à ces deux éléments le module d'instanciation est capable de générer une substitution, permettant, une fois appliquée à la formule quantifiée, d'obtenir une instance ground. Afin de déterminer si cette instance est utile ou pas ; l'algorithme d'apprentissage doit, à partir des mêmes éléments, trouver des similitudes, ou des éléments de comparaison. Concrètement, nous passons en entrée du prédicteur un vecteur de features, disjoint (comme vu précédemment), qui contient ces trois éléments : la formule quantifiée $\psi[\bar{x}_n]$ à instancier, l'ensemble de littéraux $E_i = l_1, \ldots, l_m$ produit au $i^{\text{ème}}$ tours d'instanciation par le solveur ground, et la substitution, $\bar{x}_n \mapsto \bar{t}_n$, produite par le module d'instanciation pour la formule $\psi[\bar{x}_n]$, formant le vecteur $x = (\texttt{features}(E_i), \texttt{features}(\bar{x}_n \mapsto \bar{t}_n), \texttt{features}(\psi[\bar{x}_n]))$, appelé état. Au travers de la fonction `features`, chaque brique d'information est traduit en un vecteur de features qui encode les occurrences de chaque séquence de symboles nécessaires au module d'instanciation pour dériver une instance particulière d'une formule quantifiée. L'idée ici est donc de donner à l'algorithme d'apprentissage approximativement la même information utilisée par chaque stratégie d'instanciation.

En pratique, chaque vecteur de features est de longueur fixe, généralement un grand entier naturel (voir section 5.3.2), cependant la plupart des vecteurs calculés pour une instance ne

contiennent qu'un petit sous-ensemble de séquences de symboles. En effet la plupart des valeurs correspondantes aux séquences n'apparaissent pas dans la modélisation de l'état x, et prennent, par conséquent, une valeur nulle dans le vecteur de features. Il est donc indispensable d'utiliser une représentation en mémoire plus adaptée à ces objets non denses. Le vecteur sparse est une représentation idéale dans cette configuration, car il a la particularité de ne stocker, en mémoire, que les valeurs non nulles du vecteur, le vecteur original pouvant être aisément reconstruit.

Théoriquement, l'encodage des états peut se faire, au moyen de vecteur sparse, par la traduction littérale de E_i, de la formule quantifiée, et de la substitution en vecteur de features, sous forme d'état comme décrit plus haut. Au départ, la première version de notre prototype se basait sur cette modélisation. Malheureusement, bien que cette modélisation semble traduire de manière fidèle l'état du solveur au moment d'instancier la formule, elle est volumineuse, et encode un nombre trop important d'informations. La conséquence directe est la production de modèle trop gros par l'algorithme d'apprentissage, et produire de gros modèles est contraignant. Car premièrement, ça rallonge les temps d'apprentissage et donc par conséquent limite les possibilités de paramétrage du modèle prédictif. Deuxièmement pour être intégré à veriT, le modèle est traduit vers du code C. Par conséquent augmenter la taille du modèle réduit la portabilité du code, mais rallonge aussi les temps de compilation. Enfin, plus on augmente la taille des modèles prédictifs, et plus les temps de prédiction se rallongent, ce qui n'est pas non plus un comportement désiré.

Afin de réduire la quantité d'informations, nous n'allons plus prendre la formule quantifiée complète $\psi[\bar{x}_n]$, mais seulement les triggers associés à la formule quantifiée. En outre, plutôt que de prendre en compte tous les termes dans E_i $l_1, \ldots, l_m$, nous ne sélectionnons que les termes de E_i dont les classes d'équivalences sont non vides, ou explicitement différentes, dans la fermeture de congruence. Toutefois, nous ne faisons pas de distinction entre les triggers extraits des formules quantifiées par le solveur, et ceux fournis par l'utilisateur. Cependant, on pourrait très bien imaginer un système dans lequel chaque trigger se verrait attribué une priorité, une information supplémentaire qui pourrait être exploitée pour l'apprentissage des instances. Cette nouvelle modélisation des états nous permet d'associer pour chaque instance produite par le module d'instanciation, un vecteur de features dont la taille est considérablement réduite, et qui contient les informations essentielles pour l'instanciation des formules quantifiées. D'un point de vue expérimental cette modélisation permet le passage à l'échelle de l'approche dans le solveur SMT veriT sur des problèmes de grande taille tel que ceux de la catégorie UF de la librairie SMT-LIB (voir les évaluations en section 5.4).

Afin d'améliorer les prédictions du classifieur, un certain nombre d'informations génériques portant sur la structures des termes sont ajoutées à chaque vecteur de features. Plus spécifiquement, pour chaque état on calcule : la taille, la profondeur moyenne et maximale des termes apparaissant dans la substitution, le nombre de constantes de Skolem, le nombre de termes total, et le nombre de triggers, la profondeur moyenne des triggers, et des termes en relation avec la substitution, plus exactement les termes équivalents ou explicitement différents, dans la fermeture de congruence, aux termes présents dans la substitution, ainsi que la somme des tailles des classes

id	features	valeur
1	a	1
2	c	1

TABLE 5.3 – Substitution

id	features	valeur
3	a	1
4	c	1

TABLE 5.4 – Termes egaux

id	features	valeur
5	$\circledast$	2
6	$\neg$	1
7	$\sqsubset$	1
8	$(\neg, \sqsubset)$	2
9	$(\sqsubset, \circledast)$	2
10	$(\neg, \sqsubset, \circledast)$	2

TABLE 5.5 – Triggers

d'équivalence de chaque terme de la substitution. L'exemple ci-dessous illustre, concrètement, la caractérisation des états en vecteur de features.

Exemple 5.3.3. Considérons l'instance $\sigma_{2,1}$ de l'exemple 5.2.2. La formule quantifiée est φ_2, et la substitution est $\sigma_{2,1}$ comme représentée par le tableau 5.3. Il n'y a pas d'égalité dans E_2. Ainsi, il n'existe pas d'autres termes égaux à a ou c : le vecteur de features correspondant aux littéraux ne contient des features que pour a et c, et le vecteur de features est représenté par le tableau 5.4.

On remarque que le tableau 5.3, et le tableau 5.4 diffèrent par les identifiants des features, puisque ces identifiants pour la partie substitutions et littéraux sont disjoints. Le seul trigger dans φ_2 est $\neg(x \sqsubset y)$ comme représenté par le tableau 5.5. Le vecteur de features contient donc toutes les tableaux 5.3–5.5, ainsi que les informations supplémentaires mentionnées ci-dessus.

•

id	features	valeur
11	a	1
12	b	1

TABLE 5.6 – Termes non égaux

id	features	valeur
3	a	2
4	c	1
13	$\odot$	1

TABLE 5.7 – Termes égaux

Exemple 5.3.4. Pour illustrer l'usage des features correspondantes aux littéraux nous considérons le deuxième tour de l'exemple 5.2.2, avec les littéraux $\mathsf{a} \simeq \mathsf{sk}$ et $\mathsf{b} \not\simeq \mathsf{a}$ ajoutés à E_2. Les termes égaux aux termes substitués donnent les features du tableau 5.7 en raison de l'égalité $\mathsf{a} \simeq \mathsf{sk}$. En outre, il existe maintenant une diségalité qui produit les features du tableau 5.6. Le tableau de triggers est inchangée. •

5.3.4 Apprentissage automatique

Précédemment, nous avons défini le contexte dans lequel nous souhaitons utiliser un algorithme d'apprentissage automatique, l'instanciation dans le SMT solveur veriT. Plus précisément

l'instanciation pour la résolution de problèmes non satisfaisables, veriT n'implémentant pas d'approche d'instanciation pour les problèmes satisfaisables. Pour cela nous avions premièrement besoin de définir une traduction, des termes de la logique du premier ordre, vers des vecteurs de features. Nous avions aussi besoin de caractériser les exemples que nous souhaitions suggérer à l'algorithme d'apprentissage, ce qui est défini dans la section juste ci-dessus. Maintenant que nous avons une entrée qui satisfait le formalisme supporté par les algorithmes d'apprentissage automatique, nous allons nous intéresser à comment apprendre à partir des exemples construits en section 5.3.3.

Les algorithmes d'apprentissage automatique sont des outils basés sur des approches mathématiques, et statistiques. Initialement développés pour résoudre un large éventail de problèmes à partir d'un certain nombre d'observations, ces approches permettent d'apprendre des tâches sans avoir de réelles connaissances préalables du domaine d'application. Le processus d'apprentissage automatique se compose généralement de deux phases. La première est la phase appelée d'apprentissage. C'est le moment où, à partir d'un ensemble fini d'observations récoltées depuis l'application que l'on souhaite améliorer, l'algorithme d'apprentissage tente de construire un modèle prédictif. Dans notre contexte cela consiste à récolter un certain nombre d'exemples d'instances, telles que définies dans la section précédente, puis à entraîner un modèle prédictif sur ces observations. La deuxième partie est la phase d'évaluation, le modèle prédictif est déterminé, et peut être utilisé comme un classifieur, pour évaluer de nouvelles observations. Cette phase est aussi appelée phase de prédiction.

En fonction des données à disposition, il est possible d'utiliser plusieurs méthodes d'apprentissage. Dans le cas où l'on dispose d'un ensemble d'observations annotées, suffisamment grand, il est possible d'utiliser une approche par apprentissage supervisé. Dans ce cas de figure, l'algorithme d'apprentissage tente généralement de construire un modèle prédictif à partir de ces observations. Les méthodes d'apprentissage les plus populaires qui utilisent ce principe sont : l'approche par support de vecteur machine (SVM) [119], la méthode des k plus proches voisins, les systèmes de réseaux de neurones artificiels, appelés aussi apprentissage profond lorsque ils sont constitués de plusieurs sous-couches [74], le boosting [54], ou encore les arbres de décision [38] (que nous allons utiliser).

Une autre méthode d'apprentissage automatique est l'apprentissage par renforcement. Cette approche fonctionne à partir d'un système de récompense. Chaque choix effectué par l'algorithme est évalué par une fonction de récompense, qui valorise les choix ayant des conséquences positives. Cette approche est non supervisée, c'est-à-dire qu'aucune observation préalable n'est nécessaire. Le système est intégré au programme, ou à l'application, et est conçu pour apprendre les "meilleures" décisions tout au long de son exécution. Un exemple très populaire de cette approche est le système AlphaGo [109] connu pour avoir battu, en mai 2017, le champion du monde Ke Jie de jeu de Go après avoir joué virtuellement un grand nombre de parties. Cette approche a aussi été intégrée avec succès dans le prouveur automatique leanCoP [91] donnant lieu au système rlCop [68], qui après plusieurs utilisations est capable de battre son homologue.

Dans le domaine de l'apprentissage automatique il existe deux types d'apprentissage possibles,

le premier est appelé la classification. Dans ce cas de figure l'algorithme d'apprentissage tente de ranger chaque observation dans une classe. La deuxième approche consiste à apprendre une fonction à partir d'une certain nombre d'observations. Dans ce cas les annotations correspondent à la valeur attendue par la fonction recherchée. Typiquement l'algorithme d'apprentissage va essayer d'approximer la fonction à partir des informations données.

Ainsi, quel que soit l'objectif, ou le type d'application sur lequel on souhaite utiliser une méthode d'apprentissage automatique il est essentiel de disposer d'un nombre important d'observations différentes. Plus on dispose d'exemples de qualité, et plus le modèle produit sera précis, et efficace dans ses prédictions, sur des nouvelles observations. Lorsque le modèle prédictif semble répondre correctement sur un large gamme de nouvelles observations, on dit que ce modèle généralise le problème. À l'inverse, on souhaite à tout prix éviter le phénomène de sur-apprentissage, qui se produit généralement lorsque l'ensemble des observations, utilisées pour entraîner le modèle, est trop petit ou pas assez varié. Lorsque le modèle sur-apprend, on obtient généralement de très bons résultats sur l'ensemble des observations utilisées pour entraîner le modèle. Néanmoins, lorsque le modèle est utilisé en production, les prédictions sur de nouvelles observations ne sont pas bonnes.

5.3.5 Les arbres de décision

Dans le contexte de cette thèse, nous utilisons une approche d'apprentissage supervisé, basé sur les arbres de décision. Plus précisément nous utilisons l'algorithme XGBoost [34], qui est une approche relativement sophistiquée combinant deux méthodes d'apprentissage : les arbres de décision, et le boosting. De plus le processus d'apprentissage est amélioré par une approche par descente de gradient permettant d'optimiser la recherche de modèles prédictifs. Nous donnons dans cette section une intuition, sans rentrer dans les détails, sur le fonctionnement de cet algorithme.

Globalement, les arbres de décision sont des outils probabilistes très simples permettant de faire un choix en fonction de plusieurs facteurs. Les arbres sont généralement constitués de noeuds, de branches, et de feuilles. Chaque nœud de l'arbre décrit la distribution de la variable aléatoire à prédire, chaque branche correspond à un choix, et chaque feuille à une valeur pour la prédiction. Il y a donc autant de branches que de valeurs à prédire. L'arbre est ensuite construit en fonction des observations dont on dispose. Supposons dans notre contexte que l'on souhaite prédire si une instance particulière est utile ou non. Pour cela nous allons construire un arbre binaire, dont la profondeur dépend du nombre de features recueillies dans les observations. Supposons comme dans la section précédente que l'on dispose d'un certain nombre de critères (features), qui pour simplifier l'exemple correspondent simplement aux caractéristiques des termes, profondeur, tailles, etc.

Exemple 5.3.5. Supposons que l'on souhaite prédire si une instance est utile, à partir de la taille maximale des termes obtenus dans la substitution, du nombre de symboles qui apparaissent dans E_i, du nombre de symboles différents, et du nombre de Skolems. Si l'on souhaite en particulier

prédire l'instance $\varphi_2\sigma_{2,1}$ de l'exemple 5.2.2, et que notre ensemble de features est le suivant $\{\texttt{max\_size}, \texttt{nb\_symb\_E}, \texttt{nb\_symb}, \texttt{nb\_sk}\}$, alors il est possible de construire l'arbre de décision suivant :

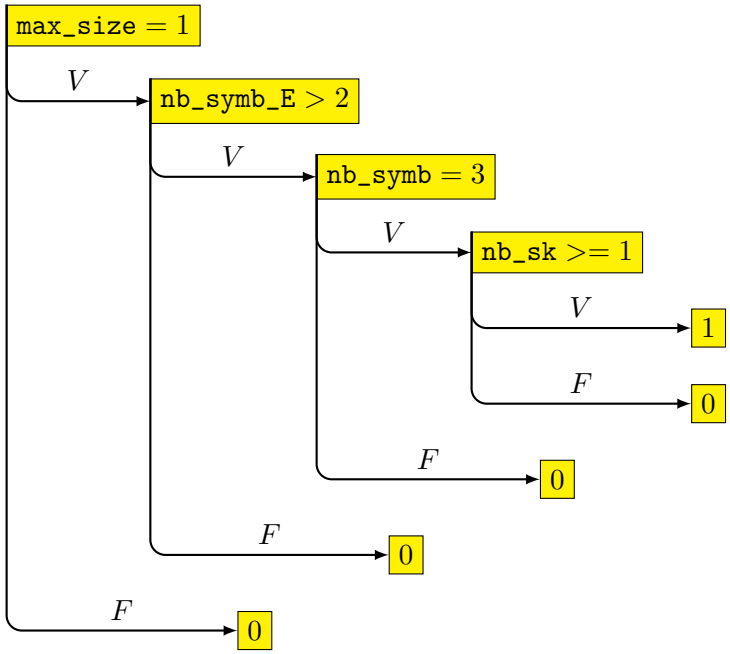

FIGURE 5.4 – Arbre de décision

D'après l'arbre ci-dessus, obtenu grâce aux observations recueillies à partir de l'exemple 5.2.2, on peut conclure que l'instance $\varphi_2\sigma_{2,1}$ n'est pas utile. En effet si l'on récupère les features de cette instance on obtient $\texttt{max\_size} = 1$, $\texttt{nb\_symb\_E} = 2$, $\texttt{nb\_symb} = 2$, $\texttt{nb\_sk} = 0$. Évidement ceci n'est qu'un exemple, et l'on remarque immédiatement que ce modèle n'est pas très robuste. •

L'algorithme XGBoost se base donc sur ce principe pour construire ses modèles. La particularité de cet algorithme est qu'il ne construit pas un seul arbre mais plusieurs. C'est le principe du Boosting, qui s'appuie sur l'hypothèse suivante : un ensemble de classifieurs faibles permet d'obtenir un classifieur fort. Un classifieur faible est un classifieur qui seul n'est pas capable de produire de prédictions exploitables. Généralement, chaque classifieur est spécialisé sur un sous domaine restreint de features. Cependant lorsque l'on réunit plusieurs de ces classifieurs, le principe de Boosting affirme que l'interprétation de la somme de ces prédictions est plus efficace que celle d'un unique classifieur fort, c'est-à-dire un classifieur capable d'interpréter l'ensemble du domaine de features du problème d'entrée. En pratique, la prédiction finale est donc obtenue par l'intermédiaire d'un système de vote entre les classifieurs. Dans l'algorithme XGBoost, les classifieurs faibles sont des arbres de décision.

Une petite subtilité lors de la construction des arbres, rend cet algorithme encore plus efficace. En effet l'algorithme XGBoost s'appuie sur une descente de gradient pour optimiser chacun des nouveaux arbres produits, ce qui permet de rendre les prédictions de chaque nouvelle forêt

d'arbres encore plus précise que la précédente. Pour plus de détails il est intéressant de consulter le papier original de XGBoost [34], mais aussi l'article fondateur sur les forêts aléatoires [32].

5.3.6 Intégration du classifieur

La section précédente explique le fonctionnement de l'algorithme d'apprentissage XGBoost. Dans cette section nous allons nous intéresser à l'intégration de l'algorithme dans le solveur SMT veriT.

Évaluer les instances. Nous avons vu précédemment que l'algorithme d'apprentissage utilisé dans ce travail se base sur des modèles prédictifs qui sont des arbres de décision. Techniquement chaque noeud de ces arbres effectue une comparaison entre la valeur d'une feature et une constante déterminée par l'algorithme lors du processus d'apprentissage, dans la phase d'entraînement. Le modèle prédictif produit par l'algorithme XGBooost, est une collection d'arbres de décision. La précision des prédictions dépend donc de deux paramètres : le premier est la profondeur de chacun de ces arbres, et le second le nombre d'arbres. La distribution des features pour chacun des arbres est aléatoire, cela signifie qu'un arbre seul ne contient pas l'ensemble des features du problème, cependant l'algorithme distribue les features de sorte que la forêt d'arbres soit capable d'évaluer n'importe quelle entrée contenant les features vues pendant la phase d'entraînement. Idéalement, afin de minimiser les temps de prédiction, il serait souhaitable d'entraîner des modèles avec des arbres peu profonds, et peu nombreux, proportionnellement au nombre total de features contenues dans l'ensemble de nos observations. Dans notre cas la contrainte sur les temps de prédiction est forte. En effet puisque l'on cherche à évaluer l'utilité de chaque instance, il est important de ne pas retarder le solveur lors de cette phase de filtrage, autrement les gains seraient négligeables. La question est : de combien d'arbres le modèle peut-il disposer, et de quelle profondeur maximale peuvent-ils être, sans perturber le solveur. Les expériences ont montré que l'utilisation de modèles composés de 110 arbres, de profondeur maximale de 20 était un bon compromis. Dans notre cas il est d'autant plus important de maximiser ces chiffres car les features récoltées se basent sur des symboles, il est donc important que l'algorithme connaisse le maximum de symboles pour évaluer le plus de problèmes. Chaque arbre $\mathcal{M}_k$ renvoie un score $\mathcal{M}_k(\mathtt{x})$ qui est une valeur réelle dans l'intervalle $(-\infty, \infty)$. Pour exploiter les prédictions du modèle prédictif il faut donc premièrement sommer l'ensemble des valeurs retournées par chaque arbre, puis passer cette valeur dans une fonction "d'activation" non linéaire, permettant de ramener cette somme dans un intervalle réel $(0, 1)$. La fonction sigmoïde pour calculer la prédiction à partir d'une forêt de n arbres de décision pour le vecteur de features $\mathtt{x}$ est la suivante :

$$\mathtt{xgb\_predict}(\mathtt{x}) = \frac{1}{1 \; + \; e^{-\sum_k \mathcal{M}_k(\mathtt{x})}}$$

qui donne une valeur réelle comprise entre 0 et 1, 0 correspondant à une instance non utile, et 1 à une instance utile. C'est donc cette fonction qui est utilisée pour évaluer chaque vecteur de features qui lui-même contient l'instance à évaluer.

Exploiter les prédictions. Nous avons décrit les mécanismes pour évaluer une instance au travers d'un vecteur de features. Il importe maintenant de comprendre comment exploiter ces prédictions, de façon à aider au maximum le solveur à résoudre de nouveaux problèmes. Il faut garder à l'esprit que l'approche présentée se base sur les symboles, ce qui signifie que cette approche n'apportera aucune valeur ajoutée si les problèmes à évaluer n'ont aucun symbole en commun.

Par conséquent afin d'exploiter pleinement les capacités du classifieur, il faut mettre en place un mécanisme capable de déterminer si les symboles présents dans l'instance que l'on souhaite évaluer sont des symboles connus du classifieur. Pour aller plus loin il est possible de classer l'ensemble des features, apparaissant dans le modèle, qui sont majoritairement des symboles, par ordre d'importance. Une feature est plus importante qu'une autre si elle apparaît plus souvent, et plus haut dans les arbres du modèle. La librairie XGBoost permet d'extraire à partir d'un modèle une table d'association de valeur d'*importance* pour chaque feature du modèle. Ainsi, sur une nouvelle observation, le vecteur de features calculé à partir de la méthode décrite plus haut, contiendra un certain nombre peut-être nul de features présentes dans le modèle. En utilisant la table d'association générée à partir du modèle, nous pouvons calculer pour chaque observation recueillie une valeur d'importance. Cette valeur peut être ensuite utilisée pour évaluer la pertinence de chaque prédiction. Ainsi, si l'importance des features d'un vecteur correspondant à une observation est faible, la prédiction sera de mauvaise qualité, et par conséquent ces observations ne doivent donc pas être évaluées par l'algorithme.

Étant donné un vecteur de features `x` correspondant à une instance, nous calculons son importance `imp(x)` comme la moyenne de toutes les valeurs de l'importance des features dans le vecteur `x`. Si cette valeur est inférieure à un certain paramètre λ, la prédiction calculée avec `xgb_predict(x)` est considérée comme non informative, et l'instance n'est pas filtrée. La fonction de prédiction qui est utilisée pour le filtrage est la suivante :

$$\texttt{predict(x)} = \begin{cases} \texttt{xgb\_predict(x)} & \text{if } \texttt{imp(x)} \geq \lambda \\ 1 & \text{otherwise} \end{cases}$$

où λ est un facteur qui dépend du nombre de constantes de Skolem présent dans l'instance, plus exactement 80 fois le nombre de constantes de Skolem. Cette valeur a été déterminée expérimentalement et donne de bons résultats dans la pratique. Intuitivement le nombre de Skolem est une information très importante, celle-ci est aussi d'ailleurs utilisée suite à ce travail en pratique pour filtrer des instances. Plus de détails sont donnés sur ce paramètre dans le chapitre suivant.

Filtrage des instances Nous allons maintenant présenter l'approche implémentée dans veriT, pour filtrer les instances produites par le module d'instanciation, générées depuis les stratégies par trigger ou par énumération. À chaque cycle d'instanciation, toutes les instances produites sont stockées dans une file d'attente, ainsi que leur score, calculé comme décrit ci-dessus. Les instances dont le score est supérieur à $0,5$ sont ajoutés au solveur ground. Les autres sont stockées, pour être réévaluées au prochain tour. Dans certains cas de figure, il peut arriver que les prédictions attribuées soient faibles pour toutes les instances. Par conséquent, si la valeur limite de filtrage

n'est pas réévaluée, le solveur risque tout simplement d'abandonner le problème par manque d'instances. Nous avons d'ailleurs observé que dans ces cas il est préférable de recalibrer le filtre de sélection, et de relancer la sélection. Ainsi en pratique lorsque le score moyen des instances est trop faible, il est possible de recalculer une valeur de filtrage permettant de réévaluer les instances dans ce contexte. On parle donc dans ce cas de figure de récupération, ou de sauvetage d'instances, dans le sens où pour une raison inconnue l'ensemble des instances produites, n'est pas évalué de façon cohérente par le prédicteur. En pratique, la valeur de récupération est calculée comme ci-dessous,

$$\texttt{rescue\_value}(I) = \left|\texttt{mean}(I) - 0.26\frac{\sigma(I)}{10}\right|$$

où $\texttt{mean}(I)$ et $\sigma(I)$ sont respectivement la moyenne et l'écart-type de l'ensemble des scores des instances, et 0.26 et 10 sont deux constantes ajustables. L'algorithme 1 donne un aperçu de la procédure. D est une file d'attente globale contenant toutes les instances qui ont été générées lors des précédents cycles, mais qui n'ont pas encore été sélectionnées pour être ajoutées aux solveur ground. Chaque nouvelle série d'instances est produite en utilisant la fonction $\texttt{TriggerEnum}(Q)$, qui incarne le module d'instanciation utilisant les stratégies de triggers, et d'énumération.

Si la file D est petite (lignes 3–4), ou si les instances ne sont pas pertinentes, l'algorithme demande au module d'instanciation de produire de nouvelles instances via l'appel à $\texttt{TriggerEnum}(Q)$. Plus précisément la condition $\texttt{irrelevant}(D)$ dans l'algorithme 1 (ligne 3) est vraie si le cycle d'instanciation précédent n'a pas produit de nouvelles instances, et qu'aucune des instances de D n'a été sélectionnée. Autrement, cela signifie que les instances générées lors du cycle précédent peuvent être utilisées, et qu'il n'est pas encore nécessaire d'en générer de nouvelles. L'algorithme va ensuite essayer de filtrer les instances. En premier lieu, l'algorithme évalue le score de chaque instance (lignes 5 à 7), en utilisant la fonction $\texttt{predict}(\texttt{features}(\varphi))$, où $\texttt{features}(\varphi)$ transforme une instance en un vecteur de features. Si une instance obtient un score supérieur à $0,5$, l'algorithme stocke l'instance. Finalement, si aucune instance n'a été sélectionnée par les passes de filtrage précédentes, l'algorithme déclenche le processus de sauvetage (lignes 8–11) qui filtre les instances dont la condition est plus faible, $\texttt{rescue\_value}(I)$ est généralement inférieur à $0,5$. À la ligne 11, toutes les instances générées qui n'ont pas été sélectionnées sont mises dans D.

Dans cette section nous avons présenté l'algorithme de filtrage utilisé dans veriT pour évaluer chacune des instances produites par les stratégies de triggers et d'énumération. Nous allons dans la section suivante présenter les résultats obtenus avec l'implémentation présentée ci-dessus.

5.4 Evaluation

Pour évaluer l'intérêt des techniques présentées ici, nous comparons d'abord le nombre d'instances nécessaires pour résoudre le problème avec le nombre d'instances, total, utilisées par le solveur. Ensuite, nous étudions le temps, et les taux de réussite avec quelques variantes de notre prototype. Les expériences ont été menées dans le solveur SMT veriT, sur des machines équipées de deux processeurs Intel Xeon Gold 6130 avec 16 cœurs par processeur et 192 GiB RAM. Nous avons

Input: Q set of quantified formulas
Output: S selected instances

$I = D$
$S = \emptyset$
if $|D| < 100 \vee$ `irrelevant`(D) **then**
    $I = I \cup$ `TriggerEnum`(Q)
foreach $\varphi \in I$ **do**
    if `predict`(`features`$(\varphi)) > 0.5$ **then**
        $S = S \cup \{\varphi\}$
if $S = \emptyset$ **then**
    foreach $\varphi \in I$ **do**
        if `predict`(`features`$(\varphi)) >$ `rescue_value`(I) **then**
            $S = S \cup \{\varphi\}$
$D = I \setminus S$
return S

Algorithm 1: Instance selection

réalisé nos expériences en utilisant les benchmarks de la catégorie UF de l'édition 2019 de la SMT-LIB, en comptant 7572 formules (771 sont étiquetées comme satisfaisantes, 3442 comme insatisfaisantes et 3359 comme inconnues). Toutes les informations nécessaires pour reproduire les expériences sont disponibles sur la page web `https://members.loria.fr/DElOuraoui/smtml.html`. Nous utilisons pour toutes nos évaluations une version de veriT qui produit des preuves. Dans la phase d'apprentissage, nous comparons la preuve complète avec la preuve dite élaguée. Les preuves dites élaguées sont les preuves qui ne contiennent que les instances utiles qui ont permis de résoudre le problème. Ainsi une instance produite lors de l'exécution du solveur est marquée comme utile si elle apparaît dans la preuve élaguée. Comme nous voulons essentiellement filtrer les instances générées par triggers et par énumération, nous ignorons les instances (utiles) générées depuis la stratégie de conflits (qui représentent environ 30% des cas). Dans la suite de la discussion, nous ne considérons ici que les chiffres sans les instances de conflit. Environ 90% des cas restants sont inutiles. Plutôt que de sur-échantillonner les instances utiles, nous utilisons un taux d'apprentissage (le paramètre XGBoost `scale_pos_weight`) plus élevé afin d'équilibrer l'ensemble des données (nous remercions Jan Jakubův et Josef Urban de nous l'avoir recommandé).

Nous exécutons tout d'abord veriT sans apprentissage sur tous problèmes de la catégorie UF, avec une limite de temps de 60 secondes. On traite ensuite les données obtenues en ne retenant que les observations obtenues à partir des problèmes résolus dans le temps imparti, et éliminons tous les problèmes qui ne font pas appel aux stratégies par triggers ou énumération. Le nombre de problèmes restant après filtrages est de 1865. Cet ensemble de problèmes est ensuite aléatoirement séparé en un ensemble d'entraînement (70% des observations) et un ensemble de test (30%).

L'approche présentée se base principalement sur l'évaluation de la fréquence d'apparition de certains motifs de symboles apparaissant de manière plus récurrente dans un problème. Une telle approche peut paraître faible au premier abord, cependant elle peut se révéler très utile dans le cadre de la preuve assistée par ordinateur, et plus particulièrement si elle est utilisée comme une assistance pour l'utilisateur. En effet, lorsque l'on travaille sur des preuves formelles, via des outils comme Isabelle, Coq ou encore l'Atelier B, l'ensemble de symboles est généralement assez restreint, les prédicats, fonctions, et constantes sont fixes. L'utilisateur doit résoudre un grand nombre d'obligations de preuves, en faisant régulièrement appel à des solveurs automatiques de type SMT. On imagine donc ici un système capable de s'intégrer à ce type d'outils, et capable d'enregistrer les observations extraites à partir de chaque obligation de preuve résolue grâce à un solveur SMT. Lorsque le nombre d'observations est suffisamment important pour produire un modèle robuste, le modèle est utilisé et le filtrage peut être ainsi activé dans le solveur SMT. Incrémentalement, toute nouvelle observation peut être ajoutée à la base de connaissance pour produire un nouveau modèle plus robuste que le précédent, de sorte que le solveur soit de plus en plus efficace sur de nouveaux problèmes. Les évaluations présentées dans cette section simulent ce comportement. Nous gérons de façon incrémentale deux modèles. Un premier modèle est produit à partir d'un ensemble d'observations extraites via les problèmes que le solveur SMT veriT est capable de résoudre sans filtrage d'instance. Dans un second temps, un deuxième modèle est produit à partir des observations extraites des problèmes résolus depuis une version du solveur veriT qui utilise le filtrage d'instance, et dont les prédictions se basent sur celles du premier modèle.

La version de veriT qui utilise l'algorithme de sélection d'instances entraîné sur les problèmes générés depuis veriT est appelée veriT($\mathcal{M}$). De même, veriT($\mathcal{M}^2$) est la version de veriT qui utilise l'algorithme de filtrage d'instance avec un modèle entraîné, cette fois, avec les problèmes résolus par veriT($\mathcal{M}$). Il y a 1914 benchmarks utilisés dans ce second cycle d'entraînement.

Nous considérons aussi une approche dite portfolio veriT($\mathcal{M} + \mathcal{M}^2$), qui est un script permettant de hiérarchiser les appels à veriT par configurations, en priorisant les configurations de veriT qui résolvent le plus de problèmes rapidement : premièrement veriT($\mathcal{M}^2$) sans apprentissage puis, veriT($\mathcal{M}$) et finalement veriT, en attribuant à chacun $1/3$ de temps. Un point dans la figure 5.5 est lu, depuis l'axe x, comme le nombre d'instances générées par veriT($\mathcal{M}$), et, depuis l'axe y, comme le nombre d'instances générées par veriT.

Moins il y a d'instances générées, meilleur est le solveur. Cependant il faut garder à l'esprit que si une instance nécessaire est filtrée, le solveur pourrait ne pas être en mesure de prouver la formule. La figure montre les résultats des solveurs sur l'ensemble des données (d'entraînement et de test). Sur la figure, un groupe de points se forme le long de la ligne correspondant à l'équation $f(x) = 2x$ (notez que $g(x) = x/2$). La comparaison sur l'ensemble de test, uniquement, a produit un graphique similaire à la Figure 5.5 mais avec moins de points. Cette figure est disponible sur la page web `https://members.loria.fr/DElOuraoui/smtml.html`. En moyenne, le filtrage d'instances permet au solveur de trouver des preuves avec environ la moitié du nombre d'instances nécessaires par rapport à la version de veriT sans le filtrage, c'est-à-dire que le nombre

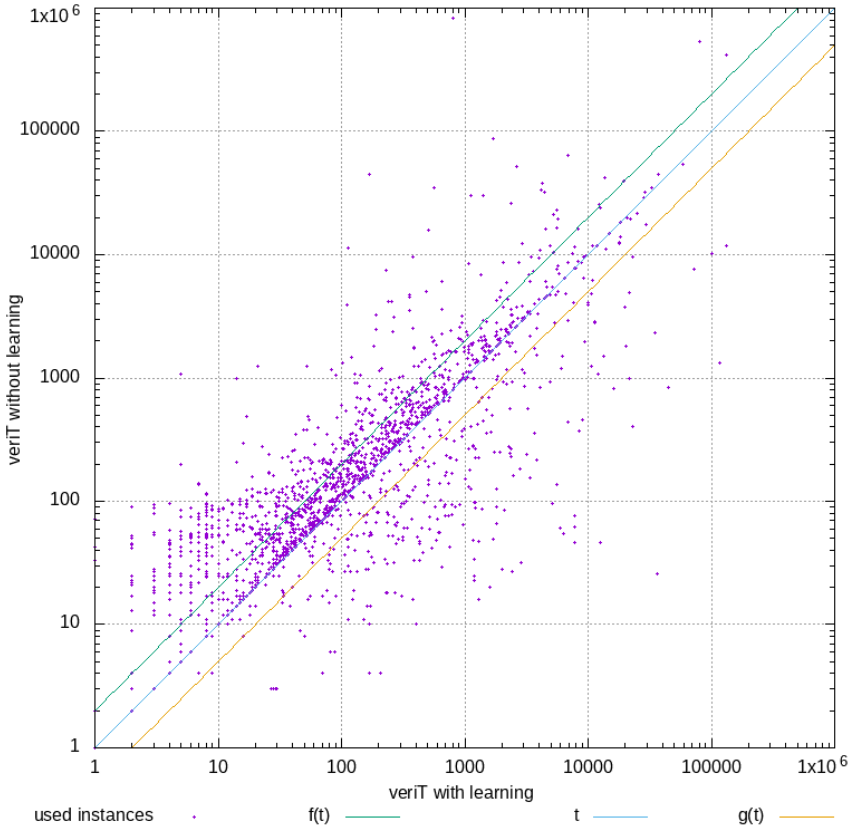

FIGURE 5.5 – Comparaison du nombre d'instances générées par les configurations veriT avec et sans apprentissage sur les benchmarks SMT-LIB UF (les 1914 benchmarks)

d'instances inutiles est réduit d'un facteur deux. Ces résultats montrent que notre approche permet de réduire considérablement le nombre d'instances inutiles. Nous montrons maintenant qu'elle permet également au solveur de prouver davantage de formules.

Pour nos évaluations nous présentons d'abord les résultats obtenus sur les benchmarks de la catégorie UF, par sous-catégories. Dans ces comparaisons, seule les sous-catégories de UF contenant un nombre conséquent de problèmes sont présentées dans les tables ci-dessous. Toutes ces tables comparent le nombre de problèmes non satisfaisables résolus, dans un temps limite, pour les différentes versions du SMT solveur veriT. Nous comparons ces différentes versions de veriT aux résultats obtenus par les deux solveurs ayant obtenus les meilleurs résultats, avant veriT, à la SMT-COMP de 2019 et 2020, dans cette catégorie, qui sont respectivement le SMT solveur cvc4 [44] et le prouveur de théorèmes Vampire [73]. Nous utilisons la version 4.2.2 de Vampire avec l'option dédiée au mode SMT-COMP, et sans l'option Z3 SMT solveur, puisque les évaluations n'impliquent pas de raisonnement théorique. Enfin les lignes intitulées "portefeuille" font état d'une stratégie portfolio, en utilisant veriT avec diverses configurations (comme celles utilisées à la SMT-COMP), avec, et sans les stratégies d'apprentissage. Vampire et cvc4 utilisent également l'approche par portfolio : l'exécution de nombreuses stratégies différentes sur le même problème dans une fraction du temps alloué donne en effet généralement de bien meilleurs résultats que l'exécution d'une seul configuration pour l'ensemble du temps alloué.

	Sledg (3675 pbs)	Grass (204 pbs)	Barrett (2371 pbs)	Total
veriT	609	196	822	1627
veriT($\mathcal{M}$)	612	195	829	1636

TABLE 5.8 – Résultats sur l'ensemble de problèmes sans les problèmes utilisés pour former veriT($\mathcal{M}$).

Pour une meilleure interprétation des résultats, nous donnons les résultats de chaque configuration de veriT utilisant l'approche par apprentissage, contre la version sans apprentissage, où l'ensemble des problèmes utilisés pour entraîner les modèles sont retirés des benchmarks pour la comparaison. Ainsi, la table 5.8 ci-dessus compare veriT à veriT($\mathcal{M}$) avec une limite de temps de 180 secondes sur les benchmarks de UF, et en ayant retiré les problèmes utilisés pour former le modèle utilisé par veriT($\mathcal{M}$). La table 5.9 ci-dessous compare avec une limite de temps de 180 secondes veriT à veriT($\mathcal{M}^2$) sur les benchmarks de UF, et en ayant retiré les problèmes utilisés pour former le modèle utilisé par veriT($\mathcal{M}^2$).

	Sledg (3669 pbs)	Grass (192 pbs)	Barrett (2379 pbs)	Total
veriT	609	183	836	1628
veriT($\mathcal{M}^2$)	614	184	842	1640

TABLE 5.9 – Résultats sur l'ensemble de problèmes sans les problèmes utilisés pour former veriT($\mathcal{M}^2$).

Finalement la table 5.10, ci-dessous, compare les résultats de l'approche veriT($\mathcal{M}^2$), avec une limite de temps de 180 secondes, combinant les 2 modèles vu précédemment, en utilisant une approche par portfolio décrite plus haut. Pour cette évaluation, nous avons retiré, dans un souci d'équité, les problèmes utilisés pour entraîner le modèle de veriT($\mathcal{M}$) et le modèle de veriT($\mathcal{M}^2$). Cette configuration est celle qui montre les meilleurs résultats. On peut en outre observer que veriT($\mathcal{M}+\mathcal{M}^2$) est capable de résoudre 28 problèmes de plus que veriT. Des résultats encore plus encourageants sont observés avec la version portfolio de veriT($\mathcal{M}+\mathcal{M}^2$), qui résout 80 problèmes de plus que l'approche classique par portfolio utilisé lors de la compétition SMT-COMP.

	Sledg (3529 pbs)	Grass (122 pbs)	Barrett (2180 pbs)	Total
veriT	470	114	638	1222
veriT($\mathcal{M} + \mathcal{M}^2$)	482	116	652	1250
veriT + portfolio	667	121	727	1515
veriT($\mathcal{M} + \mathcal{M}^2$) + portfolio	721	121	752	1595

TABLE 5.10 – Résultats sur l'ensemble de problèmes moins les problèmes utilisés pour former veriT($\mathcal{M} + \mathcal{M}^2$).

Le tableau 5.11, et la table 5.12, montrent les résultats obtenus sur l'ensemble des problèmes de la catégorie UF (sur ces évaluations aucun problème n'a été enlevé). La première table 5.11, donne une comparaison de toutes les configurations de veriT, avec les deux solveurs cvc4, et Vampire, dans les sous-intervalles de temps 30 s, 60 s, 120 s et 180 s.

Étant donné que l'application visée par cette approche est les assistants de preuves, il est

	30 s	60 s	120 s	180 s
veriT	2896	2913	2923	2929
veriT($\mathcal{M}$)	2907	2917	2925	2936
veriT($\mathcal{M}^2$)	2916	2927	2935	2944
veriT($\mathcal{M} + \mathcal{M}^2$)	2936	2959	2969	2975
veriT + portfolio	3181	3215	3228	3234
veriT($\mathcal{M} + \mathcal{M}^2$) + portfolio	3190	3247	3312	3322
Vampire smtcomp mode	3154	3165	3175	3197
CVC4 portfolio	3311	3345	3393	3404

Table 5.11 – Résultats sur les benchmarks dans la catégorie SMT-LIB UF.

important de montrer que cette approche peut aussi produire de bon résultats avec de petits intervalles de temps. La table 5.12 ci-dessous apporte un niveau de granularité supplémentaire dans l'interprétation de ces résultats, puisqu'elle décrit les résultats obtenus par sous-catégories. On peut notamment remarquer que dans les résultats obtenus, le nombre de problèmes résolus, en particulier, dans la catégorie de Sledgehammer, augmente considérablement, à mesure où l'on utilise veriT($\mathcal{M}$), veriT($\mathcal{M}^2$), et veriT($\mathcal{M} + \mathcal{M}^2$).

	Sledgehammer	misc	grass	Barrett
veriT	1072	14	431	1412
veriT($\mathcal{M}$)	1073	14	431	1418
veriT($\mathcal{M}^2$)	1079	14	431	1420
veriT($\mathcal{M} + \mathcal{M}^2$)	1093	14	434	1434
veriT + portfolio	1276	14	439	1505
veriT($\mathcal{M} + \mathcal{M}^2$) + portfolio	1333	14	440	1535
Vampire smtcomp mode	1389	15	436	1481
CVC4 portfolio	1366	14	440	1584

Table 5.12 – Résultats par sous-catégorie de SMT-LIB UF.

La conclusion de ces évaluations montre qu'un solveur utilisant une approche par apprentissage peut résoudre beaucoup plus efficacement les problèmes que les solveurs SMT classiques, surtout lorsque l'intervalle de temps de chaque exécutions est long. La comparaison avec d'autres solveurs n'est faite qu'à titre d'information ; d'ailleurs, utiliser un solveur SMT avec ce type d'approche pour le concours SMT-COMP serait déloyal, car le solveur entraîné se souvient en quelque sorte des formules qu'il a déjà vues, et prend des raccourcis dans son espace de recherche, en sélectionnant les bonnes instances. Le fait est plutôt qu'un solveur SMT doté d'une méthode apprentissage automatique peut tirer des leçons de ses précédents succès et par conséquent résoudre encore plus de problèmes. Finalement, l'approche par apprentissage compense quelque peu, dans un solveur modérément efficace, l'absence d'un grand nombre d'heuristiques et de stratégies soigneusement ajustées à la main, comme celles que l'on trouve dans les meilleurs solveurs actuels.

	veriT	veriT($\mathcal{M}$)	veriT($\mathcal{M}^2$)	veriT($\mathcal{M} + \mathcal{M}^2$)
veriT	0	39	32	15
veriT($\mathcal{M}$)	47	0	33	1
veriT($\mathcal{M}^2$)	48	41	0	10
veriT($\mathcal{M} + \mathcal{M}^2$)	64	42	44	0

TABLE 5.13 – Résultats comparatifs sur la catégorie SMT-LIB UF, avec une limite de temps de 60 secondes.

Le tableau 5.13 montre le gain et la perte par rapport au nombre de problèmes résolus. Une cellule du tableau compare deux configurations de solveurs différents, ainsi deux informations peuvent être obtenues à partir de la même configuration de solveur. Une cellule donne le nombre de problèmes perdus par la configuration sur la colonne par rapport à la configuration sur la ligne. Dans l'autre sens, une cellule peut également être lue comme le nombre de problèmes gagnés par la configuration sur la ligne par rapport à la configuration sur la colonne. Par exemple, veriT($\mathcal{M}$) résout 47 problèmes que veriT ne résout pas, tandis que veriT résout 39 problèmes que veriT($\mathcal{M}$) ne résout pas. De même, il est important de garder à l'esprit que les 47 problèmes sont des problèmes qui n'ont pas été utilisés pour former le modèle de veriT($\mathcal{M}$), puisque seuls les problèmes résolus par veriT sont utilisés pour entraîner veriT($\mathcal{M}$). Ainsi, l'utilisation de l'approche portfolio est un moyen d'atténuer le fait que l'algorithme d'instanciation a inévitablement des effets négatifs pour certaines formules.

5.5 Travaux similaires

Un certain nombre d'autres approches par apprentissage ont été étudiées pour la démonstration automatisée et interactive. Ainsi, la sélection de prémisses [1] ou le filtrage de pertinence [26] est le problème de sélection d'un sous-ensemble minimal d'une base de connaissances plus large nécessaire à un ATP pour prouver une conjecture donnée. Diverses approches par apprentissage, dont les arbres à gradient renforcé [94], et l'apprentissage profond [66] ont été essayées dans le cadre du raisonnement automatique. D'ailleurs, un certain nombre de solveurs modernes utilisent des stratégies de preuves automatiques par filtrage. L'approche par apprentissage a également été appliquée pour prédire la satisfaisabilité et l'équivalence des expressions [2] et même pour synthétiser directement des preuves dans des logiques plus simples [107]. Enfin, l'approche par apprentissage a été utilisée pour guider directement les procédures automatisées de preuves. Cela a d'abord été fait pour la sélection des étapes d'extension dans les preuves de tableaux [118], puis plus récemment combiné avec la recherche de preuves basée sur l'algorithme de Monte-Carlo [68]. Cette approche a aussi été adaptée pour le calcul de superposition, et la stratégie de sélection de clauses, et plus particulièrement dans le solveur E [76], en incluant notamment une approche par apprentissage pour la gestion des watchlist [57]. D'un autre coté, des méthodes de synthèse de substitutions combinant des approches par apprentissage profond, ont aussi été développées ; on peut en particulier citer le prouver Holophrasme [123]. Cependant seuls de très petits termes utiles ont à l'heure actuelle pu être générés à partir de ces méthodes. À notre

connaissance, aucune des approches existantes ne considère l'approche par apprentissage pour le filtrage d'instance dans les solveur SMT.

5.6 Conclusion

Lorsqu'un solveur SMT cherche à prouver des formules quantifiées, il utilise des techniques d'instanciation qui créent de nombreuses instances, dont seulement un très petit nombre sont utiles. Et plus important encore, ces instances inutiles nuisent à l'efficacité du solveur. Nous avons présenté dans ce chapitre une méthode permettant de combiner les techniques d'instanciation avec l'apprentissage machine, en l'occurrence pour le filtrage. Cette première expérience démontre que les approches par apprentissage peuvent être utilisées avec succès dans un solveur SMT, et permettent d'améliorer l'efficacité du solveur sur un ensemble cohérent de problèmes. Nous pensons donc que les méthodes présentées dans ce chapitre sont utiles, et donnent la capacité à un solveur SMT de tirer des leçons de ses précédents succès. Ainsi, lorsque le solveur est utilisé comme back-end d'une application de type assistant de preuves, ou tout autre application reposant sur la résolution de contraintes logiques, le solveur peut s'améliorer, et utiliser les observations précédentes pour se consolider dans le but de résoudre toujours plus de nouveaux problèmes présentés par l'utilisateur.

Les travaux futurs concernent l'amélioration de la caractérisation de l'état d'un solveur SMT. Nos premières expériences montrent que notre façon de calculer les entiers associés aux features (en utilisant des hachages) induit un grand nombre de conflits, et le fait de corriger cela conduirait à de meilleurs résultats. Nous prévoyons également d'étudier des caractéristiques plus indépendantes de la syntaxe, ce qui aiderait également un solveur SMT à transférer les connaissances acquises d'un ensemble de problèmes à d'autres problèmes provenant de différents domaines. Notons qu'il est aussi possible de résoudre cette problématique en s'appuyant sur un réseau de neurones pour trouver automatiquement une meilleure représentation. D'autres techniques d'apprentissage pourraient également être essayées pour notre application. Enfin, la sélection de bonnes instances est également un problème important pour la logique d'ordre supérieur, et les méthodes étudiées ici pourraient aussi trouver leur utilité dans un prouveur SMT qui traite l'ordre supérieur.

6

Contrôler le processus d'instanciation

Sommaire

Dans ce chapitre, nous présentons une heuristique qui permet de contrôler plus finement le processus d'instanciation des quantificateurs. Cette heuristique résulte d'une observation recueillie, lors du développement de l'approche présentée au chapitre précédent 5. Ce qui suit s'inscrit dans la continuité du chapitre précédent, et il est important pour la bonne compréhension des techniques développées ici de se référer à l'introduction sur l'instanciation des quantificateurs dans les solveurs SMT, donnée dans le chapitre précédent. Le principe de l'instanciation dans les solveurs SMT s'appuie sur une sélection efficace de termes permettant de dériver une certain nombre d'instances à partir de formules quantifiées présentes dans un problème non satisfaisable. Les stratégies utilisées dans les solveurs SMT, pour dériver de telles instances sont, comme montrées dans le chapitre précédent, encore imparfaites. Plus particulièrement les stratégies par triggers et énumérations, produisent un nombre important d'instances pour chaque formule quantifiée. Trop souvent ces instances sont générées à l'aveugle, c'est-à-dire qu'il n'y a aucune garantie de leur utilité (au sens précisé dans le chapitre 5), ce qui a généralement pour conséquence d'augmenter le nombre de modèles, et donc forcément de ralentir la convergence du solveur vers une solution, c'est-à-dire de trouver la ou les instance rendant la partie ground (sans quantificateur) du problème non satisfaisable.

Pour résoudre ce problème nous avons proposé dans le chapitre précédent une méthode de sélection permettant de n'introduire que les instances jugées utiles, au moyen d'algorithmes d'apprentissage. Au cours du développement de cette approche nous avons réalisé qu'un paramètre, en particulier, permettait d'influer de façon importante sur les prédictions de l'algorithme d'apprentissage. Ce paramètre, évoqué en section 5.3.6, exprime le nombre de symboles de Skolem

présents dans une instance mais aussi dans le problème ground (puisque chaque observation emporte avec elle une partie du problème ground). Ainsi, à la suite d'une étude plus approfondie des conséquences liées à ce paramètre, nous avons réalisé, qu'un nombre important de symboles de Skolems, pouvait indiquer de mauvais choix de la part du module d'instanciation, et conduire le solveur dans une impasse lors de sa recherche de solution. Par conséquent nous avons mis au point une heuristique permettant de réduire le nombre de symboles Skolem, grâce à un critère simple qui est décrit dans la première partie de ce chapitre 6.1.

Une seconde heuristique est développée dans la deuxième partie de ce chapitre 6.2. Elle permet de restreindre de manière naïve mais efficace, le nombre d'instances par formule quantifiée. À l'origine, cette stratégie de restriction est motivée par une discussion initiée par Andrew Reynolds et Haniel Barbosa qui dans un premier temps suggéraient une restriction des paires triggers-instances, dans la stratégie d'instanciation par triggers. L'approche développée par Andrew Reynolds, et expérimentalement implémentée dans le solveur cvc4 a montré de nettes améliorations (en termes de nombre de problèmes résolus et de vitesse de résolution) sur des problèmes avec quantificateurs de la librairie SMT-LIB. Par conséquent nous avons dans un premier temps essayé cette approche puis tenté de généraliser l'idée pour les deux stratégies d'instanciation, à savoir par triggers et par énumération. Ce travail a abouti à la stratégie de sélection présentée dans ce chapitre et qui a aussi permis d'obtenir de très bons résultats sur les problèmes avec quantificateurs de la librairie SMT-LIB.

La combinaison des deux approches développées permet une amélioration assez remarquable du nombre de problèmes résolus avec de petites limites de temps, 30 secondes. Ces résultats sont présentés dans la troisième partie de ce chapitre 6.3.

6.1 Contrôler la skolemisation

La skolemisation des formules est une étape importante dans le processus de résolution de problèmes avec quantificateurs. Comme décrit dans la section 3.2 la skolemisation permet d'éliminer les occurrences de quantificateurs existentiels dans une formule. Le principe général consiste à introduire une nouvelle fonction dont l'arité dépend du nombre de quantificateurs universels qui précèdent le quantificateur existentiel à éliminer. Par exemple, pour skolemiser la formule $\forall(x_1 : \tau_1)\forall(x_2 : \tau_n)\ \exists(y : \sigma)\ p(x_1, x_2, y)$, y devra être remplacé par une fonction dite de *Skolem* f, dont l'arité sera deux. La formule obtenue après skolemisation est $\forall(x_1 : \tau_1)\forall(x_2 : \tau_n)\ p(x_1, x_2, f(x_1, x_2))$, où y, la variable existentiellement quantifiée, a été remplacée par la fonction f, appliquée aux deux variables universellement quantifiées x_1, x_2 qui précédaient le quantificateur existentiel liant y.

En pratique, et plus particulièrement pour le solveur SMT veriT, la skolemisation est effectuée, uniquement, quand la quantification existentielle apparaît en tête de la formule, c'est-à-dire que le premier quantificateur de la formule est un quantificateur existentiel. Dans cette configuration, la fonction de Skolem introduite est d'arité nulle, et on parle alors de *constante* de

Skolem. Par exemple dans veriT la formule $\forall(x_1 : \tau_1)\forall(x_2 : \tau_n)\ \exists(y : \sigma)\ p(x_1, x_2, y)$ n'est pas immédiatement skolemisée. Néanmoins, après instanciation des quantificateurs universels $\forall(x_1 : \tau_1)$ et $\forall(x_2 : \tau_2)$, l'instance $\exists(y : \sigma)\ p(a, b, y)$, où les constantes a et b ont été déterminées au moyen des stratégies d'instanciation décrites précédemment (en section 2.3), peut être skolemisée car $\exists(y : \sigma)$ apparaît en tête de la formule. Pour skolemiser l'instance $\exists(y : \sigma)\ p(a, b, y)$, le solveur veriT, introduit une nouvelle constante sk, qui permet d'éliminer le quantificateur existentiel placé en tête, obtenant ainsi la formule ground $p(a, b, sk)$, c'est-à-dire sans quantificateur.

Cette gestion paresseuse des quantificateurs existentiels permet de simplifier l'implémentation du processus de skolemisation, ainsi que la gestion des variables (renommage, portée...) dans les solveurs. En contrepartie, le solveur est obligé d'instancier tous les quantificateurs universels avant de pouvoir traiter les quantificateurs existentiels. Ce qui a pour conséquence de réduire le contrôle et la visibilité du module d'instanciation, mais aussi du solveur, sur les formules quantifiées à instancier, en particulier lorsque ce sont des formules complexes, c'est-à-dire qu'elles contiennent une alternance de quantificateurs existentiels et universels. Une seconde conséquence, dont les effets peuvent altérer le bon fonctionnement du solveur est le nombre important de nouveaux symboles introduits en comparaison avec l'approche classique. L'exemple ci-dessous illustre ce comportement.

Exemple 6.1.1. Soient $f, g, h : \tau \to \tau$, considérons l'ensemble de contraintes $E_0 = \{\varphi_1\}$, où la formule quantifiée φ_1 est la suivante :

$$\exists y_1.\forall x.f(x) \simeq y_1 \wedge \exists y_2.h(y_2) \simeq g(x) \wedge \forall z.x \not\simeq h(y_1) \wedge ((z \simeq f(y_2) \wedge y_2 \simeq f(z)) \Rightarrow y_2 \simeq z)$$

Cette formule est non satisfaisable. Par conséquent veriT, devrait être capable de déterminer une preuve de ce problème rapidement. Si l'on s'appuie sur le processus de skolemisation implémenté dans le solveur SMT veriT, décrit juste avant, pour éliminer les quantificateurs existentiels dans cette formule, on doit en premier lieu, après avoir éliminé le quantificateur de tête de la formule φ_1, obtenir la formule φ_2 suivante :

$$\forall x.f(x) \simeq sk_1 \wedge \exists y_2.h(y_2) \simeq g(x) \wedge \forall z.x \not\simeq h(sk_1) \wedge ((z \simeq f(y_2) \wedge y_2 \simeq f(z)) \Rightarrow y_2 \simeq z)$$

où sk_1 est le symbole de Skolem fraîchement introduit. À cette étape il n'existe qu'un unique symbole ground pour substituer la variable x, et donc instancier la formule φ_2. On suppose donc que via la stratégie d'instanciation par énumération le solveur dérive l'instance $f(sk_1) \simeq sk_1 \wedge \varphi_3$, où φ_3 est la formule suivante :

$$\exists y_2.h(y_2) \simeq g(sk_1) \wedge \forall z.sk_1 \not\simeq h(sk_1) \wedge ((z \simeq f(y_2) \wedge y_2 \simeq f(z)) \Rightarrow y_2 \simeq z)$$

Le problème obtenu après instanciation est donc le suivant $E_1 = \{f(sk_1) \simeq sk_1, \varphi_3, \varphi_2, \varphi_1\}$. Le littéral $f(sk_1) \simeq sk_1$ est trivialement satisfaisable, il faut donc que le solveur lance à nouveau la procédure d'instanciation. À cette étape le module d'instanciation a plusieurs choix, soit instancier de nouveau φ_2, ou skolemiser φ_3 et φ_1. L'instance produite à partir de φ_2 est $f(f(sk_1)) \simeq sk_1 \wedge \varphi_4$, où φ_4 est la formule :

$$\exists y_2.h(y_2) \simeq g(f(sk_1)) \wedge \forall z.f(sk_1) \not\simeq h(sk_1) \wedge ((z \simeq f(y_2) \wedge y_2 \simeq f(z)) \Rightarrow y_2 \simeq z)$$

Après skolemisation de φ_3 on obtient la formule $h(sk_2) \simeq g(f(sk_1)) \wedge \varphi_5$ où φ_5 est :

$$\forall z.sk_1 \not\simeq h(sk_1) \wedge ((z \simeq f(sk_2) \wedge sk_2 \simeq f(z)) \Rightarrow sk_2 \simeq z)$$

La formule obtenue après avoir skolemisé une nouvelle fois φ_1 est identique à φ_2, sauf que sk_1 est remplacé par sk_3, résultant en la formule φ_6. Le problème global est maintenant $E_2 = \{h(sk_2) \simeq g(f(sk_1)), etf(f(sk_1)) \simeq sk_1, \varphi_4, \varphi_5, \varphi_6\} \cup E_1$. La partie ground $\{h(sk_2) \simeq g(f(sk_1)), f(f(sk_1)) \simeq sk_1, f(sk_1) \simeq sk_1\}$ étant trivialement satisfaisable, le module d'instanciation est de nouveau sollicité. On peut maintenant remarquer que le solveur a encore plusieurs options. Sans aucune restriction, il n'est pas garanti que le module d'instanciation, et par conséquent le solveur puisse converger vers la bonne instance pour la formule φ_5 permettant de résoudre le problème. En effet étant donné qu'à chaque étape de skolemisation le solveur introduit un nouveau symbole il est possible d'introduire une séquence cyclique d'instances : par exemple en skolemisant à chaque tour φ_1 et choisissant le terme $f(sk_i)$ pour substituer la variable x dans φ_2. Cette séquence de choix aurait des conséquences dramatiques sur les performances du solveur. •

Dans l'exemple ci-dessus nous mettons en évidence une problématique liée à la procédure de skolemisation, telle qu'implémentée dans le solveur SMT veriT. Plus particulièrement nous mettons en lumière les effets catastrophiques que peuvent entraîner un usage trop permissif de la règle de skolemisation dans ce contexte. Une façon de remédier à cette problématique est d'introduire une restriction sur chaque formule existentiellement quantifiée. Cette restriction se présente sous la forme d'un compteur attaché à chaque formule existentiellement quantifiée dans notre heuristique implémentée dans veriT. Ce compteur est ensuite utilisé pour déterminer si la formule doit être skolemisée. Si l'on reprend notre exemple ci-dessus, il est suffisant pour chaque formule existentielle de n'être skolemisée qu'une seule fois. Par conséquent avant de skolemiser chaque formule existentiellement quantifiée le module d'instanciation doit vérifier que la valeur du compteur associée à cette formule doit être inférieur à un certain seuil définit par l'utilisateur. Dans notre exemple ce seuil pourrait être 1, cependant afin d'offrir plus de flexibilité il est utile d'augmenter la valeur de ce seuil. En pratique la valeur de seuil idéale pour les problèmes de la SMT-LIB, est 9. Dans l'exemple ci-dessous nous montrons comment ce seuil permet au solveur d'avancer, et de réduire le nombre d'instances inutiles.

Exemple 6.1.2. Reprenons, notre exemple précédent, après avoir skolemisé une première fois la formule φ_1 et instancié la formule φ_2. Le problème est donc le suivant $E_1 = \{f(sk_1) \simeq sk_1, \varphi_3, \varphi_2, \varphi_1\}$. Précédemment, à cette étape le module d'instanciation avait trois options : soit instancier de nouveau φ_2, ou skolemiser φ_3 ou φ_1. À présent si l'on considère un seuil de 1 pour chaque formule quantifiée existentiellement, le module d'instanciation ne peut plus skolemiser

φ_1. Les choix sont donc réduits à instancier φ_2, ou skolemiser φ_3 ; puisque cette formule n'a pas encore été skolemisée son compteur est donc à 0.

Le problème global est maintenant $E_2' = \{h(sk_2) \simeq g(f(sk_1)),$ et $f(f(sk_1)) \simeq sk_1, \varphi_4, \varphi_5\} \cup E_1$. La partie ground $\{h(sk_2) \simeq g(f(sk_1)), f(f(sk_1)) \simeq sk_1, f(sk_1) \simeq sk_1\}$ étant trivialement satisfaisable, le module d'instanciation est de nouveau sollicité. On peut de nouveau remarquer qu'a cette étape il n'est plus possible ni de skolemiser φ_3, ni φ_1, le module d'instanciation peut donc se concentrer sur les formules φ_5 et φ_2. En effet il suffit de dériver les instances $sk_1 \not\simeq h(sk_1) \wedge ((sk_1 \simeq f(sk_2) \wedge sk_2 \simeq f(sk_1)) \Rightarrow sk_2 \simeq sk_1)$, et $sk_1 \not\simeq h(sk_1) \wedge ((g(f(sk_1) \simeq f(sk_2) \wedge sk_2 \simeq f(g(f(sk_1))) \Rightarrow sk_2 \simeq g(f(sk_1))$, pour rendre le problème non satisfaisable au niveau ground. La première instance peut être dérivée à partir de la substitution de z par sk_1 alors que la seconde de z par $g(f(sk_1))$. •

Cette restriction permet de résoudre un certain nombre de cas supplémentaires, et permet une amélioration des performances de veriT. Des évaluations de cette restriction sont apportées dans la section 6.3.

6.2 Réduire les instances envahissantes

Au chapitre 5, nous avons présenté une approche permettant de réduire le nombre d'instances inutiles, au moyen d'une méthode de sélection des instances produites par les stratégies d'instanciation utilisant des triggers ou basées sur des méthodes d'énumération. Cette approche s'appuie sur un classifieur, placée en bout de chaîne (c'est-à-dire qui intervient au moment d'introduire les instances dans le solveur ground), et préalablement entraîné à reconnaître les bonnes instances. Nous avons montré que cette approche peut améliorer les résultats du solveur SMT, sur une certaine catégorie de problèmes. Dans cette section nous avons expérimenté une approche de sélection plus naïve, mais qui en pratique est efficace. La problématique soulevée dans cette section est donc identique à celle présentée dans la section 5.2.3. Il est donc ici question de déterminer une heuristique permettant de réduire le nombre d'instances introduites dans le solveur ground, à l'issue d'une des stratégies utilisée dans le solveur SMT veriT, à savoir l'instanciation par triggers ou par énumération.

L'approche est initialement motivée par une discussion entretenue avec Haniel Barbosa et Andrew Reynolds sur l'optimisation de l'approche par triggers. Plus précisément, dans cette discussion Andrew Reynolds propose une restriction pour chaque paire (triggers, terme ground) unifiable et déterminée au moyen de l'algorithme de E-matching. En pratique cette restriction a pour but de réduire les effets pervers de certains comportements exponentiels observés notamment lors de l'usage des techniques dites de multi-triggers. Cette approche généralise la notion d'instanciation par triggers présentée dans le chapitre précédent, et permet d'obtenir des substitutions, via le plus souvent l'algorithme de E-matching, qui satisfont plusieurs triggers à la fois. En effet cette technique peut être utile dans le cas où l'on se retrouve à devoir manipuler des

axiomes du type $\forall xyz.F[x,\ldots,y]$, qui nécessite l'usage d'un multi-trigger tel que $f(x), f(y), f(z)$, où il est nécessaire de déterminer une substitution via E-matching pour les 3 variables x, y, z. Le problème est que si l'on dispose de 20 termes ground candidats, alors ce seul axiome peut admettre $20^3 = 8000$ instances. Andrew Reynolds a d'ailleurs mentionné dans son mail que ce type de comportement se produisaient assez fréquemment sur des benchmarks de la librairie SMT-LIB et notamment sur ceux issus de l'outil Sledgehammer. Andrew Reynolds a donc proposé un compromis simple et élégant, qui fonctionne assez bien et qui consiste à n'utiliser qu'au plus qu'une seule instanciation pour chaque paire (trigger, terme ground). Cette restriction peut d'ailleurs être utilisée autant pour les approches multi-triggers que pour l'approche trigger unique tel que présentée dans les chapitres précédents. Cette technique est implémentée dans le solveur cvc4, et permet un gain intéressant de nouveaux problèmes résolus.

Dans veriT l'approche par multi-trigger est présente, mais en pratique ne permet pas d'obtenir de meilleurs résultats qu'une approche par triggers simple. Il est aussi important de comprendre que la force de ces approches est souvent obtenue en coopération, car l'approche par multi-triggers étant plus complète, elle résout une certaine catégorie de problème que l'approche par trigger unique ne peut pas résoudre et vice versa. En règle générale on définit une liste de paramètres qui combinés ensemble permettent de résoudre un maximum de problèmes. C'est l'approche par portefeuille de stratégies (ou portfolio en anglais). C'est donc la raison pour laquelle il est important dans ces scripts d'alterner plusieurs stratégies complètes et incomplètes pour couvrir le maximum de situations et donc maximiser le nombre de problèmes que l'on peut résoudre.

L'idée qui a donc émergé de notre côté, suite a cette discussion, est la suivante : plutôt que de restreindre les triggers, ce qui est finalement assez spécifique à une approche d'instanciation, nous avons voulu savoir s'il était possible d'appliquer une restriction plus générale à toutes les approches heuristiques d'instanciation impliquant des comportements exponentiels. En particulier l'approche par énumération peut être extrêmement coûteuse puisque pour chaque variable, elle essaye d'énumérer l'ensemble des termes possibles du même type. Par conséquent une solution assez simple consiste à appliquer une restriction sur les paires (formule quantifiée, instance), où les instances sont dérivées depuis une stratégie soit par triggers soit par énumérations. Malheureusement, restreindre à une instance chaque formule quantifiée est trop restrictif, et ne permet pas d'obtenir de résultats convainquants, néanmoins il est possible de trouver un bon compromis avec une valeur de nombre d'instances par formule quantifiée qui se situe aux alentours d'une centaine d'instances par formule quantifiée. Pour pouvoir utiliser cette approche efficacement, et donc pallier le manque de complétude d'une telle restriction, il faut la coupler à une procédure qui permet, lorsque toutes les formules quantifiées ont atteint le seuil limite d'instances, d'autoriser de façon incrémentale de nouvelles instances pour chacune des formules. Le seuil augmente donc incrémentalement par pas de trente instances jusqu'à que le module d'instanciation puisse produire un certain nombre d'instances. Généralement ces instances produites peuvent elles-mêmes être des formules quantifiées, ce qui a pour effet de relancer le processus puisque que leur seuil est à 100 et leur compteur est initialement à zéro.

6.3 Évaluation de l'approche

Dans cette section nous présentons les évaluations des deux approches présentées dans ce chapitre. Nous présentons également quelques résultats obtenus à la compétition SMT-COMP de cette année où cette stratégie a pu être utile. Les expériences ont été menées dans le solveur SMT veriT, sur des machines équipées de deux processeurs Intel Xeon Gold 6130 avec 16 cœurs par processeur et 192 GB RAM. Nous avons réalisé nos expériences en utilisant les benchmarks de la catégorie UF de l'édition 2019 de la SMT-LIB, comptant 7572 formules.

	30 s
veriT	2903
veriT + sk	2952
veriT + sk + I	3034
Vampire smtcomp mode	3154
cvc4	3114

TABLE 6.1 – Résultats sur les benchmarks dans la catégorie SMT-LIB UF.

La table donnée dans la figure 6.1 reporte les meilleurs résultats obtenus avec chacune des heuristiques présentées dans ce chapitre, sur les benchmarks de la catégorie UF de la dernière édition de la SMT-COMP, où chaque solveur est lancé avec une limite de temps de 30 secondes. À titre informatif, ces résultats sont comparés avec ceux des solveurs cvc4 et Vampire, qui sont les solveurs ayant obtenu les meilleurs résultats dans cette catégorie, aux dernières éditions de la compétition SMT-COMP. La version de cvc4 utilisée est celle de la compétition 2019 de la SMT-COMP, la version de Vampire est la 4.2.2, sans le solveur SMT z3 (étant donné que dans ces évaluations nous ne faisons pas appel aux théories). Vampire est exécuté avec le mode smtcomp, tandis que le solveur cvc4 est ici exécuté avec le mode par défaut, ce qui explique la différence entre les résultats obtenus dans la table 5.11 et ceux présentés dans cette section. Dans la table 6.1, la configuration veriT + sk exprime l'heuristique permettant de restreindre l'instanciation des formules existentielles décrite en section 6.1. La configuration veriT + sk + I exprime la combinaison de la restriction décrite en section 6.1, ainsi que l'heuristique limitant les instances dérivées depuis des stratégies de triggers ou d'énumération, décrite en section 6.2. On peut remarquer que la première restriction veriT + sk, permet d'obtenir un gain de 51 problèmes par rapport à la version par défaut de veriT, alors que la combinaison des deux restrictions décrites dans ce chapitre permet un gain de 131 problèmes supplémentaires par rapport à la version par défaut de veriT, avec une limite de temps de 30 secondes. À titre purement informatif on peut aussi remarquer que cette optimisation permet au solveur veriT de se rapprocher des solveurs cvc4 et Vampire, qui le devance dans la compétition.

Cette heuristique a été utilisée lors de la dernière compétition de la SMT-COMP. Cependant un travail important a été réalisé pour combiner cette heuristique avec les heuristiques déjà utilisée lors des dernières éditions de la compétition. Pour obtenir l'approche portefeuille optimale, c'est-à-dire la collection de configurations lancées en séquence dans un certain ordre et dans une durée de temps fixée par les règles de la compétition, Hans-Jörg Schurr a développé une stratégie

de sélection des configurations du solveur veriT, qui maximise le nombre de problèmes résolus dans un temps imposé. Plusieurs des configurations obtenues après sélection utilisaient les heuristiques présentées dans ce chapitre. Grâce à ces efforts, cette année, la version de veriT a pu obtenir de meilleurs résultats dans plusieurs catégories de la compétitions SMT-COMP, que l'an passé. En particulier dans la catégorie 24 s Performance, veriT a obtenu la seconde place derrière les deux versions de Vampire 2018 et 2019, et devant la version cvc4 de 2019. Ces résultats sont disponibles à l'adresse suivante : `https://smt-comp.github.io/2020/results/uf-single-query`. Pour comparaison veriT se positionnait derrière les deux versions de Vampire, Par4 et les deux versions de cvc4 dans cette même catégorie (voir `https://smt-comp.github.io/2019/results/uf-single-query`), l'année précédente. C'est donc une belle avancée, en sachant que la catégorie UF est difficile.

6.4 Conclusion

Dans ce chapitre nous avons présenté deux heuristiques, se présentant sous la forme de restrictions des instances dérivées soit par le processus de skolemisation, soit par des stratégies d'instanciation par triggers ou énumération. Il est à la fois triste mais aussi intéressant d'observer que des heuristiques aussi simples que celles présentées dans ce chapitre puissent permettre au solveur SMT veriT d'améliorer ses performances sur la catégorie de problème UF de la SMT-LIB. Triste, car l'approche présentée dans ce chapitre permet à elle seule d'obtenir des résultats comparables à l'approche présentée dans le chapitre 5, qui fait intervenir des techniques complexes, et qui demande un haut niveau d'affinage au niveau des paramètres d'apprentissage et de sélections des instances. Mais à la fois intéressant, car elle met en lumière des faiblesses finalement assez évidentes des approches actuellement en place dans les solveurs SMT, et présente une solution simple pour y remédier. Les résultats présentés dans la section 6.3 apporte une forme d'espoir, pour l'amélioration du comportement en général des solveurs SMT pour la résolution des formules avec quantificateurs, et ouvre la porte vers de nouvelles optimisation pour ce type de problèmes.

Lors de la réalisation de ce travail je me suis intéressé à l'élaboration de méthodes qui permettraient de déterminer dynamiquement les valeurs de seuils limitant respectivement les instances : de formules existentiellement quantifiées (décrit en section 6.1), et de formules universellement quantifiées dérivées à partir des approches par triggers ou énumération (décrites en section 6.2). En effet, manuellement, j'ai pu observer que certains problèmes pouvaient être résolus avec une certaine valeur de seuil, soit uniquement pour limiter les instances de formules existentiellement quantifiées ou universellement quantifiées, ou bien même pour une combinaison spécifique des deux seuils. Cependant je n'ai pas réussi à déterminer une approche efficace pour pouvoir trouver dynamiquement ces seuils. Une proposition intéressante serait d'essayer dans un premier temps d'appliquer une approche par apprentissage telle que décrite dans le chapitre 5, pour apprendre les valeurs de ces seuils sur les problèmes de UF de la SMTLIB, par exemple. Puis d'utiliser les résultats pour déterminer des paramètres permettant d'adapter ces valeurs de seuil de manière

dynamique pour chaque problème. En s'appuyant sur les résultats obtenus dans les deux derniers chapitres de cette thèse on pourrait facilement imaginer la valeur ajoutée d'une approche où l'usage de ces deux seuils serait totalement contrôlé. Le code source des heuristiques décrites dans ce chapitre est disponible sur le dépôt [6].

6. `https://github.com/delouraoui/these`

7

Conclusion

Ce travail porte sur l'amélioration, d'une manière générale, des interactions entre solveurs SMT et outils de preuve formelle. Nous focalisons notre attention sur une partie de cette coopération : l'amélioration des solveurs SMT pour les outils de preuve formelle. Cette thèse est organisée selon deux axes : proposer une architecture qui étend les capacités de raisonnement des solveurs SMT à la logique d'ordre supérieur, et améliorer la gestion des quantificateurs. Ce chapitre présente les conclusions générales de ce travail.

Tout au long de ce travail nous avons essayé, autant que possible, d'apporter une évaluation réaliste des approches présentées. Ces évaluations nous ont permis d'orienter nos recherches, et d'appuyer nos analyses, c'est pourquoi nous apportons, de manière systématique, une attention particulière à la collecte de résultats expérimentaux. De plus pour chacune des approches présentées nous mettons à disposition le code source qui sera disponible depuis le dépôt [7]. Notre étude débute donc par un état de l'art qui permet d'une manière générale de comprendre le sens de notre travail. Dans ce chapitre nous abordons l'ensemble des notions fondamentales liées au raisonnement automatique et plus particulièrement à SMT. Nous présentons l'architecture et les algorithmes utilisés dans les solveurs SMT, ainsi que les approches utilisées pour traiter les problèmes de la logique du premier ordre avec quantificateurs. En fin de chapitre nous faisons état des problématiques et challenges liées au processus d'instanciations et faisons un parallèle avec les contributions apportées dans cette thèse.

Dans le chapitre 3, nous développons le premier axe de cette thèse à savoir une extension du solveur veriT pour la logique d'ordre supérieur. Nous proposons tout d'abord une extension du langage SMTLIB 2.6, permettant d'exprimer des contraintes logiques plus expressives, notamment par l'ajout de constructions pour les applications partielles, les λ-expressions et les types fonctionnels (introduction du constructeur flèche). Une description du système de types pour ces constructions, suivie d'une discussion sur les difficultés relatives à son implémentation y sont également rapportées. L'approche générale est aussi discutée dans ce chapitre, nous y dé-

7. `https://github.com/delouraoui/these`

crivons les choix de conception. Une première approche développée par des collègues de Stanford et de l'Université d'Iowa, dans le solveur cvc4 est décrite. Cette première approche basée sur une utilisation intelligente de l'opérateur @ est utilisée comme base de comparaison pour notre approche. Également implémentée dans le solveur veriT, elle nous a permis de valider nos choix de conception, et servi de point de repère lors de nos évaluations. L'approche développée dans veriT est décrite en détail, et s'organise en trois parties : la description conceptuelle du calcul, l'implémentation et la structure de données implémentée dans veriT (décrite dans le chapitre 4), puis la description de l'extension du module d'instanciation. En fin de chapitre nous décrivons les expériences réalisées avec les deux solveurs cvc4 et veriT qui s'appuient sur des comparaisons avec d'autres pouveurs dits d'ordre supérieur, sur plusieurs catégories de problèmes. Ces travaux ont fait l'objet de trois publications : pour le volet preuve et syntaxe d'ordre supérieur [7], pour le volet raisonnement sans quantificateur et résultats préliminaires [11], et finalement la présentation des deux prototypes, étendu pour la logique d'ordre supérieur, avec un support pour quantificateurs [10]. Au terme de ce travail, il ressort que les approches développées permettent d'obtenir des performances acceptables et encourageantes pour le développement de solveurs SMT pour la logique d'ordre supérieur.

Dans le chapitre 5, nous abordons la problématique de l'instanciation des quantificateurs dans les solveurs SMT d'un point de vue expérimental. Ce travail est à l'origine motivé par deux questions : est-il possible de faire coopérer une approche d'apprentissage machine (ou machine learning), dans un solveur SMT. Si oui pourrait-elle permettre de résoudre des problèmes complexes tels que celui de l'instanciation. La réponse à ces deux questions se trouve en partie dans ce chapitre. Cependant il est important de nuancer et d'interpréter correctement les résultats obtenus lors de nos expérimentations. Dans ce chapitre nous présentons le problème de l'instanciation dans SMT. L'ensemble du chapitre s'articule autour d'une étude de cas concret impliquant l'usage de plusieurs heuristiques. Le problème est décortiqué dans une première partie, ce qui nous permet de mettre en lumière les faiblesses des approches actuellement employées dans les solveurs SMT pour résoudre des problèmes avec quantificateurs. Dans une deuxième partie nous introduisons les notions de base pour l'apprentissage, qui nous permettent de donner une caractérisation de notre problème pour les algorithmes d'apprentissage. Dans une troisième partie nous présentons une heuristique de sélection s'appuyant sur des suggestions proposées par un modèle prédictif préalablement entraîné à partir d'un certain nombre d'observations. Dans la dernière partie de ce chapitre nous proposons une évaluation de notre approche sur plusieurs jeux de données. En premier lieu, nous évaluons indépendamment chaque approche, sur des problèmes nouveaux c'est-à-dire qui n'ont pas été utilisés pour construire le modèle prédictif. Puis nous comparons nos approches face à des solveurs SMT très performants. En conclusion l'approche présentée dans cette thèse permet de répondre par la positive aux deux questions initialement posées, avec une réserve. En effet les approches présentées s'appuient sur une caractérisation du problème de l'instanciation qui dépend fortement de la syntaxe (symboles de fonctions, constantes, etc.). Cette dépendance implique une condition forte pour l'emploi de cette approche qui est d'avoir un jeu de problèmes avec des noms de symboles "cohérents". C'est-à-dire qu'il faut idéalement que les mêmes symboles soient utilisées dans tout le jeu de problèmes, sur lequel on souhaite appliquer l'approche. Ces expérimentations montrent néanmoins que sur

des problèmes dont les noms de symbole sont cohérents, il est possible pour un modèle prédictif d'apporter des prédictions pertinentes améliorant le comportement des solveurs SMT, et par conséquent leurs performances. Idéalement, s'affranchir de cette condition par une caractérisation indépendante de la syntaxe, basée sur des paramètres structurels des formules et du processus de résolution du solveur, permettrait de manière certaine d'obtenir des solveurs SMT très performants.

De manière générale la combinaison des deux approches développées dans cette thèse pourrait fortement contribuer à l'amélioration des solveurs SMT dans le cadre des assistants de preuves. Néanmoins il reste des défis à relever pour atteindre un niveau de performance permettant de simplifier davantage le travail des utilisateurs dans les assistants de preuve, lors de formalisation, en rendant la résolution des obligations de preuves générées par les assistants de preuve plus systématique.

Les axes, qui à mon sens (en lien avec mon expérience de thèse), doivent à l'avenir être développés sont donnés ci-dessous. Premièrement, l'étape suivante dans la transformation des solveurs SMT existants à la logique d'ordre supérieur concerne le développement du framework CCFV pour cette logique plus expressive. Plus précisément CCFV est un calcul qui permet de résoudre des problèmes de (dé)unification modulo théorie équationnelle. En particulier ce calcul a été pensé pour pouvoir "caster" la majorité des heuristiques utilisées pour l'instanciation des formules quantifiées dans les solveurs SMT au travers d'un unique calcul. Il semble donc tout à fait naturel de s'orienter vers l'extension de ce calcul pour des logiques plus expressives telles que la logique d'ordre supérieur. En fin de thèse ce travail a été entamé en collaboration avec Haniel Barbosa, Pascal Fontaine et Sophie. La première tentative d'extension de CCFV à l'ordre supérieur consistait en une simple curryfication des règles du calcul. Malheureusement après implémentation il s'est avéré que les résultats obtenus avec ce calcul étaient en deçà de ceux obtenu avec notre première version du solveur présentée dans cette thèse. L'origine de cette inefficacité étant fortement liée au problématique d'indexation évoquées à la section 3.7 de cette thèse, nous n'avons pas retenu cette approche. Une deuxième tentative est maintenant en cours de développement. Cette approche tente d'étendre le calcul pour le fragment logique λ-free, et se base sur le fait que le problème CCFV est NP-complet, pour ce fragment. Par conséquent l'approche utilise une réduction à SAT du problème CCFV. Un premier article a été publié [116] à SMT en juillet 2020.

Concernant le deuxième axe de cette thèse à savoir l'amélioration du processus d'instanciation dans les solveurs SMT, il y a plusieurs points qu'il serait intéressant de développer dans le futur. Le problème de l'instanciation est un problème difficile, et d'un point de vue purement personnel je pense que pour le moment nous avons atteint la limite des approches existantes, pour le problème de l'instanciation modulo théorie équationnelle (UF) (d'un point de vue purement stratégie de calcul). Je pense qu'il est toujours possible d'améliorer l'efficacité des solveurs SMT tels qu'ils existent, cependant je pense que la bonne direction à emprunter se situe plutôt autour de l'étude de stratégies de filtrage des instances. Les approches par apprentissage sont assez prometteuses dans ce domaine mais elles ont des lacunes, c'est pourquoi il serait intéressant

d'investiguer plus profondément la généralisation des features. En particulier réussir à déterminer des features, plus indépendantes de la syntaxe, qui caractérisent les processus de résolution du solveur au travers de différents paramètres, comme par exemple dans l'esprit du travail réalisé sur le solveur SATzilla [124] et le travail de Mate Soos et al [110]. Pour aller plus loin, je pense qu'il est possible de développer une approche par renforcement pour la sélection d'instances dans un solveur SMT. Cette approche pourrait s'inspirer de notre travail et généraliser l'idée développée dans cette thèse. Pour cela je pense à trois options, une première pourrait consister à travailler à la génération de nouveau modèle à la volée. C'est-à-dire que le solveur pourrait stocker l'ensemble des observations relatives à ses expériences passées et générer à partir d'un certain nombre de problèmes résolus de nouveaux modèles pouvant être combinés à ceux déjà existant. Cette combinaison pourrait par exemple s'inspirer de l'approche développée en section 5.4. Une seconde option serait de s'inspirer des approches développées dans [68] et [58]. Une dernière option, que je n'ai malheureusement pas eu le temps d'essayer serait tous simplement de retenir les preuves générées pour chaque problème résolu, et tenter à partir des preuves retenues de trouver des correspondances dans les problèmes nouvellement résolus. Cette approche pourrait être tout aussi prometteuse.

Bibliographie

[1] Jesse Alama, Tom Heskes, Daniel Kühlwein, Tsivtsivadze Evgeni, and Josef Urban. Premise selection for mathematics by corpus analysis and kernel methods. *J. Autom. Reasoning*, 52(2), 2014.

[2] Miltiadis Allamanis, Pankajan Chanthirasegaran, Pushmeet Kohli, and Charles Sutton. Learning continuous semantic representations of symbolic expressions. In Doina Precup and Yee Whye Teh, editors, *ICML 2017*, volume 70, pages 80–88. PMLR, 2017.

[3] Peter B. Andrews. Resolution in type theory. *J. Symb. Log*, 36(3) :414–432, 1971.

[4] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. Logic and Computation*, 4(3) :217–247, 1994.

[5] Kshitij Bansal, Andrew Reynolds, Tim King, Clark Barrett, and Thomas Wies. Deciding local theory extensions via E-matching. In Daniel Kroening and Corina S. Păsăreanu, editors, *CAV 2015*, volume 9207 of *LNCS*. Springer, 2015.

[6] Haniel Barbosa. *New techniques for instantiation and proof production in SMT solving*. PhD thesis, Université de Lorraine, Universidade Federal do Rio Grande do Norte, 2017.

[7] Haniel Barbosa, Jasmin Christian Blanchette, Simon Cruanes, Daniel El Ouraoui, and Pascal Fontaine. Language and proofs for higher-order SMT (work in progress). In Catherine Dubois and Bruno Woltzenlogel Paleo, editors, *PXTP 2017*, volume 262 of *EPTCS*, pages 15–22, 2017.

[8] Haniel Barbosa, Pascal Fontaine, and Andrew Reynolds. Congruence closure with free variables. Technical report, Inria, 2017. `https://hal.inria.fr/hal-01442691`.

[9] Haniel Barbosa, Pascal Fontaine, and Andrew Reynolds. Congruence closure with free variables. In Axel Legay and Tiziana Margaria, editors, *TACAS 2017*, volume 10206 of *LNCS*, pages 214–230, 2017.

[10] Haniel Barbosa, Andrew Reynolds, Daniel El Ouraoui, Cesare Tinelli, and Clark Barrett. Extending SMT solvers to higher-order logic. In Pascal Fontaine, editor, *CADE-27*, volume 11716 of *LNCS*, pages 35–54. Springer, 2019.

[11] Haniel Barbosa, Andrew Reynolds, Pascal Fontaine, Daniel El Ouraoui, and Cesare Tinelli. Higher-order SMT solving (work in progress). In *SMT 2018*. Dimitrova, 2018.

[12] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB standard : version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017.

[13] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *FAIA*, chapter 26, pages 825–885. IOS Press, 2009.

[14] Clark Barrett, Igor Shikanian, and Cesare Tinelli. An abstract decision procedure for a theory of inductive data types. *J. on Satisfiability, Boolean Modeling and Computation*, 3(1-2) :21–46, 2007.

[15] Peter Baumgartner, Ulrich Furbach, and Ilkka Niemelä. Hyper tableaux. In *JELIA 1996*, pages 1–17. Springer, 1996.

[16] Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, Petar Vukmirović, and Uwe Waldmann. Superposition with lambdas. In Pascal Fontaine, editor, *CADE–27*, volume 11716 of *LNCS*, pages 35–54. Springer, 2019.

[17] Alexander Bentkamp, Jasmin Christian Blanchette, Simon Cruanes, and Uwe Waldmann. Superposition for lambda-free higher-order logic. In *IJCAR*, volume 10900 of *LNCS*, pages 28–46. Springer, 2018.

[18] Christoph Benzmuller and Chad Brown. A structured set of higher-order problems. In *TPHOLs*, pages 66–81. Springer, 2005.

[19] Christoph Benzmüller, Dale Miller, Herman Geuvers, and Bruno Woltzenlogel Paleo. Automation of higher-order logic. In Jörg H. Siekmann, editor, *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 215–254. Elsevier, 2014.

[20] Christoph Benzmüller, Nik Sultana, Lawrence C Paulson, and Frank Theiß. The higher-order prover LEO-II. *J. Autom. Reason.*, 55 :389–404, 2015.

[21] Ahmed Bhayat and Giles Reger. Set of support for higher-order reasoning. In Boris Konev, Josef Urban, and Philipp Rümmer, editors, *PAAR 2018*, volume 2162 of *CEUR Workshop Proceedings*, pages 2–16. CEUR-WS.org, 2018.

[22] Wolfgang Bibel. On matrices with connections. *J. ACM*, 28(4) :633–645, 1981.

[23] Wolfgang Bibel. *Automated theorem proving*. Springer, 2013.

[24] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. *Handbook of satisfiability*, volume 185 of *FAIA*. IOS Press, 2009.

[25] Jasmin Christian Blanchette. *Automatic proofs and refutations for higher-order logic*. PhD thesis, Technical University Munich, 2012.

[26] Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein, and Josef Urban. A learning-based fact selector for Isabelle/HOL. *J. Autom. Reason.*, 57(3) :219–244, 2016.

[27] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C Paulson, and Josef Urban. Hammering towards QED. *J. Formalized Reason.*, 9(1) :101–148, 2016.

[28] Sascha Böhme and Tobias Nipkow. Sledgehammer : judgement day. In Jürgen Giesl and Reiner Hähnle, editors, *IJCAR 2010*, volume 6173 of *LNCS*, pages 107–121. Springer, 2010.

[29] Maria Paola Bonacina, Christopher Lynch, and Leonardo De Moura. On deciding satisfiability by DPLL($\gamma + \mathcal{T}$) and unsound theorem proving. In Renate A. Schmidt, editor, *CADE–22*, pages 35–50. Springer, 2009.

[30] Maria Paola Bonacina, Christopher A Lynch, and Leonardo De Moura. On deciding satisfiability by theorem proving with speculative inferences. *J. Autom. Reason.*, 47(2) :161–189, 2011.

[31] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. veriT : an open, trustable and efficient SMT-solver. In Renate A. Schmidt, editor, *CADE–22*, volume 5663 of *LNCS*, pages 151–156. Springer, 2009.

[32] Leo Breiman. Random forests. *Machine Learning*, 45 :5–32, 2001.

[33] Chad E. Brown. Satallax : an automatic higher-order prover. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR 2012*, volume 7364 of *LNCS*, pages 111–117. Springer, 2012.

[34] Tianqi Chen and Carlos Guestrin. XGBoost : a scalable tree boosting system. In Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi, editors, *KDD 2016*, pages 785–794. ACM, 2016.

[35] Stephen A Cook. The complexity of theorem-proving procedures. In *STOC 1971*, pages 151–158. ACM, 1971.

[36] Simon Cruanes. Superposition with structural induction. In Clare Dixon and Marcelo Finger, editors, *FroCoS 2017*, volume 10483 of *LNCS*, pages 172–188. Springer, 2017.

[37] Łukasz Czajka and Cezary Kaliszyk. Hammer for Coq : automation for dependent type theory. *J. Autom. Reason.*, 61(1-4) :423–453, 2018.

[38] Steinberg Dan and Colla Phillip. CART : classification and regression trees. *The top ten algorithms in data mining*, 9 :179, 2009.

[39] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *J. ACM*, 5(7) :394–397, 1962.

[40] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3) :201–215, 1960.

[41] Leonardo De Moura and Nikolaj Bjørner. Efficient E-matching for SMT solvers. In Frank Pfenning, editor, *CADE–21*, volume 4603 of *LNCS*, pages 183–198. Springer, 2007.

[42] Leonardo De Moura and Nikolaj Bjørner. Z3 : an efficient SMT solver. In *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[43] Leonardo De Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In Armin Biere, Carl Pixley, et al., editors, *FMCAD 2009*, pages 45–52. IEEE, 2009.

[44] Morgan Deters, Andrew Reynolds, Tim King, Clark Barrett, and Cesare Tinelli. A tour of CVC4 : how it works, and how to use it. In Koen Claessen, Viktor Kunčak, and Barbara Jobstmann, editors, *FMCAD 2014*. IEEE, 2014.

[45] David Detlefs, Greg Nelson, and James B. Saxe. Simplify : a theorem prover for program checking. *J. ACM*, 52 :365–473, 2005.

[46] Gilles Dowek. Higher-order unification and matching. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, pages 1009–1062. Elsevier and MIT Press, 2001.

[47] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27 :758–771, 1980.

[48] Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich. Adding decision procedures to SMT solvers using axioms with triggers. *J. Autom. Reason.*, 56(4) :387–457, 2016.

[49] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *CAV 2014*, pages 737–744. Springer, 2014.

[50] Bruno Dutertre and Leonardo De Moura. A fast linear-arithmetic solver for DPLL(T). In Thomas Ball and Robert B. Jones, editors, *CAV 2006*, pages 81–94. Springer, 2006.

[51] Herbert B Enderton. *A mathematical introduction to logic*. Elsevier, 2001.

[52] Michael Färber and Chad E. Brown. Internal guidance for Satallax. In Nicola Olivetti and Ashish Tiwari, editors, *IJCAR 2016*, volume 9706 of *LNCS*, pages 349–361. Springer, 2016.

[53] Andreas Fellner, Pascal Fontaine, and Bruno Woltzenlogel Paleo. NP-completeness of small conflict set generation for congruence closure. *Formal Methods in System Design*, 51 :533–544, 2017.

[54] Yoav Freund, Robert Schapire, and Naoki Abe. A short introduction to boosting. *J. JSAI*, 14(771-780) :1612, 1999.

[55] Yeting Ge and Leonardo de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In Ahmed Bouajjani and Oded Maler, editors, *CAV 2009*, volume 5643 of *LNCS*, pages 306–320. Springer, 2009.

[56] Paul C Gilmore. A proof method for quantification theory : its justification and realization. *IBM J. Res. Dev.*, 4 :28–35, 1960.

[57] Zarathustra Goertzel, Jan Jakubuv, and Josef Urban. ENIGMAWatch : ProofWatch meets ENIGMA. In Serenella Cerrito and Andrei Popescu, editors, *TABLEAUX 2019*, volume 11714 of *LNCS*, pages 374–388. Springer, 2019.

[58] Stéphane Graham-Lengrand and Michael Färber. Guiding SMT solvers with Monte Carlo tree search and neural networks. In Thomas Hales, Stephan Kaliszyk, Cezary Schulz, and Josef Urban, editors, *AITP 2018*, 2018.

[59] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and trends® in programming languages*, 4(1-2) :1–119, 2017.

[60] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Le Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean Mclaughlin, Tat Thang Nguyen, and et al. A formal proof of the kepler conjecture. *Forum of Mathematics, Pi*, 5, 2017.

[61] Leon Henkin. Completeness in the theory of types. *J. Symb. Log*, 15(2) :81–91, 1950.

[62] Jinbo Huang et al. The effect of restarts on the efficiency of clause learning. In Manuela M. Veloso, editor, *IJCAI 2007*, volume 7, pages 2318–2323, 2007.

[63] Gérard Huet. A mechanization of type theory. In Nils J. Nilsson, editor, *IJCAI 1973*, pages 139–146. William Kaufmann, 1973.

[64] Gérard P. Huet. A unification algorithm for typed λ-calculus. *Theor. Comput. Sci.*, 1, 1975.

[65] R. J. M. Hughes. Super combinators : a new implementation method for applicative languages. In *LFP'82*, pages 1–10. ACM, 1982.

[66] Geoffrey Irving, Christian Szegedy, Alexander A. Alemi, Niklas Een, François Chollet, and Josef Urban. DeepMath–deep sequence models for premise selection. In Daniel D. Lee, Masashi Sugiyama, Ulrike V. Luxburg, Isabelle Guyon, and Roman Garnett, editors, *NIPS 2016*, pages 2235–2243. Curran Associates, 2016.

[67] Jan Jakubův and Josef Urban. ENIGMA : efficient learning–based inference guiding machine. In Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe, and Olaf Teschke, editors, *CICM 2017*, volume 10383 of *LNCS*, pages 292–302. Springer, 2017.

[68] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Mirek Olšák. Reinforcement learning of theorem proving. In Samy Bengio, Hanna Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *NeurIPS 2018*, pages 8835–8846. Curran Associates, 2018.

[69] Cezary Kaliszyk, Josef Urban, and Jiri Vyskočil. Efficient semantic features for automated reasoning over large theories. In Qiang Yang and Michael Wooldridge, editors, *IJCAI 2015*, pages 3084–3090. AAAI Press, 2015.

[70] Hadi Katebi, Karem A. Sakallah, and Igor L. Markov. Symmetry and satisfiability : an update. In Ofer Strichman and Stefan Szeider, editors, *SAT 2010*, pages 113–127. Springer, 2010.

[71] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. SeL4 : formal verification of an os kernel. In *SOSP'09*, page 207–220. ACM, 2009.

[72] Michael Kohlhase. Higher-order tableaux. In Peter Baumgartner, Reiner Hähnle, and Joachim Posegga, editors, *TABLEAUX 1995*, volume 918 of *LNCS*, pages 294–309. Springer, 1995.

[73] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *CAV 2013*, volume 8044 of *LNCS*. Springer, 2013.

[74] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553) :436–444, 2015.

[75] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert - a formally verified optimizing compiler. In *ERTS 2016*. SEE, 2016.

[76] Sarah Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. In Thomas Eiter and David Sands, editors, *LPAR–21*, volume 46 of *EPiC Series in Computing*, pages 85–105. EasyChair, 2017.

[77] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4) :173–180, 1993.

[78] William McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *J. Autom. Reason.*, 9 :147–167, 1992.

[79] Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *J. Autom. Reason.*, 40(1) :35–60, 2008.

[80] Dale A. Miller. A compact representation of proofs. *Studia Logica*, 46 :347–370, 1987.

[81] Georg Moser and Richard Zach. The epsilon calculus. In Matthias Baaz and Johann A. Makowsky, editors, *CSL 2003*, pages 455–455. Springer, 2003.

[82] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff : engineering an efficient SAT solver. In *DAC 2001*, page 530–535. ACM, 2001.

[83] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1 :245–257, 1979.

[84] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2) :356–364, 1980.

[85] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In Jürgen Giesl, editor, *RTA 2005*, volume 3467 of *LNCS*, pages 453–468. Springer, 2005.

[86] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *IC*, 2005(4) :557–580, 2007.

[87] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories : from an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, pages 937–977, 2006.

[88] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, pages 371–443. Elsevier and MIT Press, 2001.

[89] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL : a proof assistant for higher-order logic*, volume 2283 of *LNCS*. Springer, 2002.

[90] Kohei Noshita. Translation of turner combinators in O(n log n) space. *IPL*, 20 :71 – 74, 1985.

[91] Jens Otten and Wolfgang Bibel. leanCoP : lean connection-based theorem proving. *J. Symb. Computation*, 36(1-2) :139–161, 2003.

[92] Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska, editors, *IWIL-2010*, volume 2 of *EPiC*, pages 1–11. EasyChair, 2012.

[93] Sylvain Perifel. *Complexité algorithmique*. Ellipses, 2014.

[94] Bartosz Piotrowski and Josef Urban. ATPboost : learning premise selection in binary setting with ATP feedback. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *IJCAR 2018*, volume 10900 of *LNCS*, pages 566–574. Springer, 2018.

[95] Andrew Reynolds, Haniel Barbosa, and Pascal Fontaine. Revisiting enumerative instantiation. In Dirk Beyer and Marieke Huisman, editors, *TACAS 2018*, volume 10806 of *LNCS*, pages 112–131. Springer, 2018.

[96] Andrew Reynolds and Jasmin Christian Blanchette. A decision procedure for (co)datatypes in SMT solvers. *J. Autom. Reason.*, 58 :341 – 362, 2017.

[97] Andrew Reynolds, Cesare Tinelli, and Leonardo De Moura. Finding conflicting instances of quantified formulas in SMT. In Koen Claessen, Viktor Kunčak, and Barbara Jobstmann, editors, *FMCAD 2014*, pages 195–202. IEEE, 2014.

[98] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters, and Clark Barrett. Quantifier instantiation techniques for finite model finding in SMT. In Maria Paola Bonacina, editor, *CADE–24*, volume 7898 of *LNCS*, pages 377–391. Springer, 2013.

[99] G Robinson and Lawrence Wos. Paramodulation and theorem-proving in first-order theories with equality. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning*, pages 298–313. Springer, 1983.

[100] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1) :23–41, 1965.

[101] John Alan Robinson. Mechanizing higher order logic. *Machine Intelligence*, 4 :151–170, 1969.

[102] John Alan Robinson. Automatic deduction with hyper-resolution. *J. Symb. Log.*, 39(1) :189–190, 1974.

[103] John Alan Robinson and Andrei Voronkov. *Handbook of automated reasoning*, volume 1. Gulf Professional Publishing, 2001.

[104] Dan Rosén and Nicholas Smallbone. TIP : tools for inductive provers. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *LPAR 20*, pages 219–232. Springer, 2015.

[105] Michaël Rusinowitch. Theorem-proving with resolution and superposition. *J. Symb. Computation*, 11(1-2) :21–49, 1991.

[106] Stephan Schulz. E - A brainiac theorem prover. *AI Commun.*, 15 :111–126, 2002.

[107] Taro Sekiyama and Kohei Suenaga. Automated proof synthesis for the minimal propositional logic with deep neural networks. In Sukyoung Ryu, editor, *APLAS 2018*, volume 11275 of *LNCS*, pages 309–328. Springer, 2018.

[108] Robert E. Shostak. Deciding combinations of theories. In D. W. Loveland, editor, *CADE–6*, pages 209–222. Springer, 1982.

[109] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676) :354–359, 2017.

[110] Mate Soos, Raghav Kulkarni, and Kuldeep S. Meel. Crystalball : gazing in the black box of SAT solving. In Mikoláš Janota and Inês Lynce, editors, *SAT 2019*, pages 371–387. Springer, 2019.

[111] Alexander Steen and Christoph Benzmüller. The higher-order prover Leo-III. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *IJCAR 2018*, volume 10900 of *LNCS*, pages 108–116. Springer, 2018.

[112] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for an extensional theory of arrays. In Joseph Halpern, editor, *LICS 2001*, pages 29–37. IEEE Computer Society, 2001.

[113] Nik Sultana, Jasmin Christian Blanchette, and Lawrence C. Paulson. LEO-II and Satallax on the Sledgehammer test bench. *J. Applied Logic*, 11(1) :91–102, 2013.

[114] Geoff Sutcliffe. The TPTP problem library and associated infrastructure. *J. Autom. Reason.*, 43(4) :337–362, 2009.

[115] Geoff Sutcliffe. The CADE ATP system competition - CASC. *AI Magazine*, 37(2) :99–101, 2016.

[116] Sophie Tourret, Pascal Fontaine, Daniel El Ouraoui, and Haniel Barbosa. Lifting congruence closure with free variables to λ-free higher-order logic via SAT encoding, 2018.

[117] G. S. Tseitin. *On the complexity of derivation in propositional calculus*, pages 466–483. Springer, 1983.

[118] Josef Urban, Jiří Vyskočil, and Petr Štěpánek. MaLeCoP : machine learning connection prover. In Kai Brünnler and George Metcalfe, editors, *TABLEAUX 2011*, pages 263–277. Springer, 2011.

[119] Vladimir Vapnik. *The nature of statistical learning theory*. Springer, 2013.

[120] Andrei Voronkov. AVATAR : The architecture for first-order theorem provers. In Armin Biere and Roderick Bloem, editors, *CAV 2014*, pages 696–710. Springer, 2014.

[121] Petar Vukmirović, Alexander Bentkamp, and Visa Nummelin. Efficient full higher-order unification. In Zena M. Ariola, editor, *FSCD 2020*, volume 167, pages 5 :1–5 :17. LIPICS, 2020.

[122] Petar Vukmirović, Jasmin Blanchette, Simon Cruanes, and Stephan Schulz. Extending a brainiac prover to lambda-free higher-order logic. In Tomáš Vojnar and Lijun Zhang, editors, *TACAS 2019*, volume 11427, pages 192–210. Springer, 2019.

[123] Daniel P.Z. Whalen. Holophrasm : a neural automated theorem prover for higher-order logic. *CoRR*, abs/1608.02644, 2016.

[124] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla : portfolio-based algorithm selection for SAT. *JAIR*, 32 :565–606, 2008.

[125] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In Rolf Ernst, editor, *ICCAD 2001*, page 279–285. IEEE Press, 2001.

Résumé

La vérification formelle de programmes informatiques ou de systèmes dits critiques tels que dans le transport, l'énergie, etc, est essentielle pour garantir le bon fonctionnement de ces systèmes. Les méthodes de vérification employées s'appuient très fortement sur des procédés mathématiques et logiques permettant de raisonner de manière formelle sur le comportement de ces systèmes. Ces procédés définissent généralement les comportements sous forme de grands ensembles de contraintes logiques. L'approche par satisfaisabilité est une méthode largement utilisée pour vérifier ces contraintes et est un exemple de cas, où les solveurs SMT (satisfaisabilité modulo théories) sont fortement sollicités.

En effet, les solveurs SMT ont la capacité de manipuler de grandes formules logiques dans des langages expressifs disposant de plusieurs opérateurs interprétés (par exemple, opérateurs sur l'arithmétique et structures de données). Ils permettent donc d'automatiser la résolution de ces contraintes et sont par conséquent, aujourd'hui, très largement utilisés dans ce but.

Ces outils sont construits au-dessus de solveurs SAT (satisfaisabilité propositionnelle) pour gérer la structure booléenne de la formule et des solveurs de théorie permettant de gérer les formules atomiques (par exemple $x > y + z$ pour la théorie de l'arithmétique).

Généralement, les solveurs SMT ne gèrent que la logique de premier ordre. Ils ne peuvent pas raisonner sur des expressions d'ordre supérieur, et ils ne peuvent généralement pas effectuer de preuves par induction. C'est regrettable, car la plupart des outils de vérification interactifs, qui utilisent les solveurs SMT, utilisent des langages d'ordre supérieur.

L'objectif de cette thèse dans sa globalité est d'offrir des solutions pour améliorer les interactions entre solveur automatique et assistant de preuves. En particulier nous répondons à deux problématiques importantes permettant d'améliorer les usages de solveurs SMT au sein des assistants de preuves. Notre première contribution permet de réduire l'écart entre solveur et assistant de preuve en proposant une architecture adaptée pour la logique d'ordre supérieur. La seconde contribution permet d'améliorer les capacités de raisonnement des solveurs SMT pour les quantificateurs. Pour les deux approches développées nous apportons un ensemble d'évaluation sur des problèmes extraits pour la grande majorité de problèmes de formalisation. Les résultats obtenus lors de ces évaluations sont encourageants et montrent que les techniques développées dans cette thèse peuvent apporter de bonnes améliorations pour les solveurs SMT.

Ce doctorat s'est effectué dans le cadre du projet ERC de J. Blanchette (Matryoshka), un ambitieux projet quinquennal qui vise à construire des prouveurs automatiques utiles pour la vérification interactive, et réduire l'écart entre les prouveurs interactifs et solveurs automatiques.

L'un des objectifs concrets du projet est d'étendre les capacités de raisonnement des solveurs SMT vers l'ordre supérieur.

Abstract

Many applications, notably in the context of verification (for critical systems in transportation, energy, etc.), rely on checking the satisfiability of logic formulas.

Satisfiability-modulo-theories (SMT) solvers handle large formulas in expressive languages with built-in and custom operators (e.g. arithmetic and data structure operators). These tools are built using a cooperation of a SAT (propositional satisfiability) solver to handle the Boolean structure of the formula and theory reasoners to tackle the atomic formulas (e.g. $x > y + z$ for the theory of arithmetic).

Currently, SMT solvers only handle first-order logic. They cannot reason about higher-order expressions, and they generally cannot perform proofs by induction. This is unfortunate, because most interactive verification tools, which use SMT solvers as back-end reasoning engines, offer higher-order languages.

This thesis offers solutions to improve interactions between automatic solvers and proof assistants. In particular, we answer two important issues allowing us to improve the use of SMT solvers within proof assistants. Our first contribution consists in providing a suitable architecture to SMT solvers for higher-order logic. The second contribution aims to improve quantifier reasoning inside SMT solvers. For both approaches, we developed a practical implementation and provide a concrete evaluation on a large collection of problems mostly coming from formalization problems. The results obtained during these evaluations are encouraging and show that the developed techniques can provide good improvements for SMT solvers.

This PhD thesis is being carried out within J. Blanchette's ERC Starting Grant Matryoshka, an ambitious five-year project that aims at making automatic provers more useful for interactive verification, by reducing the gap between the automatic and interactive worlds. One of the concrete goals of the project is to lift up the reasoning capabilities of SMT solvers towards higher order.