New techniques for instantiation and proof production in SMT solving

Haniel Barbosa

Haniel Barbosa

New techniques for instantiation and proof production in SMT solving

PhD Thesis
September 2017

Thesis Supervisors: Pascal Fontaine, David Déharbe, and Stephan Merz

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Nouvelles techniques pour l'instanciation et la production des preuves dans SMT

THÈSE

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Haniel Barbosa

Septembre 2017

Membres du jury

Rapporteurs:		
Erika Ábrahám	PR	RWTH Aachen University
Philipp Rümmer	Chercheur	Uppsala University
Examinateurs:		
David Déharbe	MCF	Federal University of Rio Grande do Norte (co-directeur)
Catherine Dubois	PR	ENSIIE
Pascal Fontaine	MCF	University of Lorraine, CNRS, Inria, LORIA (co-directeur)
João Marcos	MCF	Federal University of Rio Grande do Norte
Stephan Merz	DR Inria	University of Lorraine, CNRS, Inria, LORIA (directeur)
Andrew Reynolds	Chercheur	University of Iowa

Loria
Laboratoire lorrain de recherche en informatique et ses applications

UFRN
UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

Pós-graduação em Sistemas e Computação

Novas técnicas de instanciação e produção de demonstrações para a resolução SMT

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada do Centro de Ciências Exatas e da Terra da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção do grau de Doutor em Ciência e Computação.

Haniel Barbosa

Setembro de 2017

Membros da Banca

Relatores:	
Erika Ábrahám	RWTH Aachen University
Philipp Rümmer	Uppsala University
Examinadores:	
David Déharbe	UFRN (co-orientador)
Catherine Dubois	ENSIIE (presidente da banca)
Pascal Fontaine	University of Lorraine, CNRS, Inria, LORIA (co-orientador)
João Marcos	UFRN
Stephan Merz	University of Lorraine, CNRS, Inria, LORIA (orientador)
Andrew Reynolds	University of Iowa

DIMAp UFRN

Abstract

In many formal methods applications it is common to rely on SMT solvers to automatically discharge conditions that need to be checked and provide certificates of their results. In this thesis we aim both to improve their efficiency of and to increase their reliability.

Our first contribution is a uniform framework for reasoning with quantified formulas in SMT solvers, in which generally various instantiation techniques are employed. We show that the major instantiation techniques can be all cast in this unifying framework. Its basis is the problem of E-ground (dis)unification, a variation of the classic rigid E-unification problem. We introduce a decision procedure to solve this problem in practice: Congruence Closure with Free Variables (CCFV). We measure the impact of optimizations and instantiation techniques based on CCFV in the SMT solvers veriT and CVC4, showing that our implementations exhibit improvements over state-of-the-art approaches in several benchmark libraries stemming from real world applications.

Our second contribution is a framework for processing formulas while producing detailed proofs. The main components of our proof producing framework are a generic contextual recursion algorithm and an extensible set of inference rules. With suitable data structures, proof generation creates only a linear-time overhead, and proofs can be checked in linear time. We also implemented the approach in veriT. This allowed us to dramatically simplify the code base while increasing the number of problems for which detailed proofs can be produced.

Keywords: quantifier instantiation, proof production, proof automation, smt solving, formal verification.

Résumé

Des nombreuses applications de méthodes formelles se fonde sur les solveurs SMT pour valider automatiquement les conditions à vérifier et fournissent des certificats de leurs résultats. Dans cette thèse, nous visons à la fois à améliorer l'efficacité des solveurs SMT et à accroître leur fiabilité.

Notre première contribution est un cadre uniforme pour le raisonnement avec des formules quantifiées dans les solveurs SMT, dans lequel généralement diverses techniques d'instanciation sont utilisées. Nous montrons que les principales techniques d'instanciation peuvent être jetées dans ce cadre. Le cadre repose sur le problème de l'E-ground (dis)unification. Nous présentons un procédure de décision pour résoudre ce problème en pratique: Fermeture de congruence avec variables libres (CCFV). Nous mesurons l'impact des CCFV dans les solveurs SMT veriT et CVC4. Nous montrons que nos implémentations présentent des améliorations par rapport aux approches à la fine pointe de la technologie dans plusieurs bibliothèques de référence issues d'applications du monde réel.

Notre deuxième contribution est un cadre pour le traitement des formules tout en produisant des preuves détaillées. Les principaux composants de notre cadre de production de preuve sont un algorithme de récurrence contextuelle générique et un ensemble extensible de règles d'inférence. Avec des structures de données appropriées, la génération des preuves ne crée que des frais généraux linéaires et les vérifications peuvent être vérifiées en temps linéaire. Nous avons également mis en œuvre l'approche en veriT. Cela nous a permis de simplifier considérablement la base du code tout en augmentant le nombre de problèmes pour lesquels des preuves détaillées peuvent être produites.

Mots-clés: instanciation des quantificateurs, production des démonstrations, automatisation des démonstrations, résolution smt, vérification formelle.

Resumo

Em muitas aplicações de métodos formais, como verificação formal, síntese de programas, testes automáticos e análise de programas, é comum depender de solucionadores de satisfatibilidade módulo teorias (SMT) como backends para resolver automaticamente condições que precisam ser verificadas e fornecer certificados de seus resultados. Nesta tese, objetivamos melhorar a eficiência dos solucionadores SMT e aumentar sua confiabilidade.

Nossa primeira contribuição é fornecer um arcabouço uniforme e eficiente para raciocinar com fórmulas quantificadas em solucionadores SMT, em que, geralmente, várias técnicas de instanciação são empregadas para lidar com quantificadores. Mostramos que as principais técnicas de instanciação podem ser lançadas neste arcabouço unificador para lidar com fórmulas quantificadas com igualdade e funções não interpretadas. O arcabouço baseia-se no problema de E-ground (dis)unificação, uma variação do problema clássico de E-unificação rígida. Apresentamos um cálculo correto e completo para resolver esse problema na prática: Fechamento de Congruência com Variáveis Livres (CCFV). Uma avaliação experimental é apresentada, na qual medimos o impacto das otimizações e técnicas de instanciação baseadas no CCFV nos solucionadores SMT veriT e CVC4. Mostramos que nossas implementações exibem melhorias em relação às abordagens de última geração em várias bibliotecas de referência, decorrentes de aplicações do mundo real.

Nossa segunda contribuição é uma estrutura para o processamento de fórmulas ao mesmo tempo que produz demonstrações detalhadas. Nosso objetivo é aumentar a confiabilidade nos resultados de solucionadores SMT e sistemas de raciocínio automatizado similares, fornecendo justificativas que podem ser verificadas com eficiência de forma independente e para melhorar sua usabilidade por aplicativos externos. Os assistentes de demonstração, por exemplo, geralmente requerem a reconstrução da justificação fornecida pelo solucionador em uma determinada obrigação de prova.

Os principais componentes da nossa estrutura de produção de demonstrações são um algoritmo genérico de recursão contextual e um conjunto extensível de regras de inferência. Clausificação, Skolemização, simplificações específicas de teorias e expansão das expressões "let" são exemplos dessa estrutura. Com estruturas de dados adequadas, a geração de demonstrações cria apenas uma sobrecarga de tempo linear, e as demonstrações podem ser verificadas em tempo linear. Também implementamos a abordagem em veriT. Isso nos permitiu simplificar drasticamente a base do código, aumentando o número de problemas para os quais demonstrações detalhadas podem ser produzidas.

Palavras-chave: instanciação de quantificadores, produção de demonstrações, automatizaçao de demonstrações, resulução smt, verificação formal.

Acknowledgments

First and foremost, I thank my main supervisor Pascal Fontaine for his almost endless patience and certainly endless support. This thesis was only possible because he believed in me far more than I did myself. I also thank my other supervisors David Déharbe and Stephan Merz, who helped me so much during these years.

I am grateful to the other members of my jury, Erika Ábrahám, Philipp Rümmer, Catherine Dubois, João Marcos, and Andrew Reynolds, who accepted to dedicate their time to read and evaluate this thesis. I also thank Laurent Vigneron, who provided helpful suggestions for my work as it progressed.

During these years of conferences and work travels, I had many interesting technical discussions with colleagues. I thank you all, specially Andrew Reynolds and Jasmin Blanchette, who became my co-authors and helped me so much into putting my results out there in the world.

The years in Nancy were made pleasant by the friends I made here. I say thank you, specially to Jordi Martori, Jonàs Martínez, Aybüke Özgün, Maike Massierer, Martin Riener, and Simon Cruanes.

I thank Inria and the Université de Lorraine for financially supporting me throughout my PhD.

E eu não posso deixar de agradecer a meus pais, que tanto sacrificaram para que eu pudesse ter uma boa educação e aprendesse a perseguir meus objetivos. Da mesma forma, minha família em geral e amigos do Brasil foram fundamentais para que eu mantivesse minha sanidade durante esses anos, me acolhendo tão alegremente durante minhas visitas. E claro, eu agradeço especialmente a Samantha, que ilumina minha vida com seu carinho e que me inspira com sua tenacidade.

To Samantha, for putting up with me.

"I guess I'm not in the mood for it today," Paul said.
"Mood?" Halleck's voice betrayed his outrage even through the shield's filtering. "What has mood to do with it? You fight when the necessity arises — no matter the mood! Mood is a thing for cattle or making love or playing the baliset. It's not for fighting."

Dune, by Frank Herbert

Contents

List of Figures

List of Tables

Extended Abstract in French

Introduction

Dans de nombreuses applications des méthodes formelles, telles que la vérification formelle, la synthèse de programmes, les tests automatiques et l'analyse de programme, il est fréquent de s'appuyer sur la logique pour représenter des conditions de vérification qui doivent être validées. Ces conditions peuvent souvent être réduites au problème de la *satisfiabilité modulo des théories* (SMT) (Chapitre 2): étant donné une formule logique de premier ordre, y a-t-il un modèle compatible avec une combinaison de théories d'arrière-plan? Les solveurs SMT sont ainsi devenus des backends populaires pour les outils d'automatisation de la vérification formelle, la synthèse de programmes, etc... Ils valident automatiquement les obligations de preuve — conditions à vérifier — et fournissent des certificats de leurs résultats, c'est-à-dire des modèles ou des preuves. Dans cette thèse, nous visons à la fois à améliorer l'efficacité des solveurs SMT et à accroître leur fiabilité.

Les solveurs SMT (Chapitre 3) ont été initialement conçus pour résoudre des problèmes sans quantification, sur lesquels ils sont très efficaces et capables de traiter de grandes formules avec des symboles interprétés. La logique du premier ordre pure avec quantificateurs est traitée plus avantageusement avec la *résolution* et la *superposition* [BG94, NR01]. Bien qu'il existe des premières tentatives [dMB08c] pour unifier ces techniques avec CDCL($\mathcal{T}$) [NOT06] (le calcul que les solveurs SMT implémentent habituellement), l'approche la plus commune est encore l'*instanciation*: les formules quantifiées sont réduites à des formules sans quantificateurs elles-même réfutées avec l'aide de procédures de décision pour les formules sans quantificateurs. Les principales techniques d'instanciation sont le E-matching basé sur les tiggers [DNS05, dMB07, Rüm12], la recherche d'instances conflictuelles [RTdM14] et l'instanciation de quantificateurs basée sur les modèles (MBQI) [GdM09, RTG+13]. Chacune de ces techniques contribue à l'efficacité des solveurs modernes, mais chacune est généralement implémentée indépendamment.

Nous présentons le problème de E-ground (dis)unification comme la pierre angulaire d'un cadre unique dans lequel toutes ces techniques peuvent être exprimées (Chapitre 4). Ce problème est lié au problème classique de E-unification [DV98] rigide et est également NP-complet. Résoudre le problème de E-ground (dis)unification se ramène à la recherche de substitutions telles que les littéraux contenant des variables libres soient substitués en littéraux présents dans le problème sans quantificateurs. Étant donné que le domaine d'instanciation de ces variables

est borné, un moyen possible de résoudre le problème est d'abord de déterminer de façon non déterministe une substitution et de vérifier s'il s'agit d'une solution. L'algorithme de *Fermeture de congruence avec variables libres* (congruence closure with free variables, CCFV) présenté ici est une procédure de décision pratique pour ce problème qui s'appuie sur l'algorithme classique de fermeture de congruence [NO80, NO07]. Il est axé sur les objectifs: les solutions sont construites de manière incrémentielle, compte tenu de la fermeture de congruence des termes définis par les égalités dans le contexte et des attributions possibles aux variables. Nous montrons ensuite comment se baser sur CCFV pour implémenter une instanciation basée sur les triggers, sur les conflits ou sur les modèles (Chapitre 5). Une évaluation expérimentale est présentée, dans laquelle nous mesurons l'impact des optimisations et des techniques d'instanciation basées sur CCFV. Nous montrons que nos implémentations présentent des améliorations par rapport à l'état de l'art. Il s'agit d'un travail en commun avec Pascal Fontaine et Andrew Reynolds [BFR17].

Les techniques d'instanciation pour SMT ont été largement étudiées. L'instanciation heuristique basée sur E-matching des déclencheurs sélectionnés a d'abord été introduite par Nelson [Nel80] et a été mise en œuvre avec succès dans plusieurs solveurs. Une implémentation très efficace de E-matching dans le contexte des solveurs SMT a été présentée par de Moura et Bjørner [dMB07]; Elle s'appuie sur des algorithmes d'unification incrémentaux, une génération de code machine pour optimiser les performances et des techniques d'indexation élaborées ayant des caractéristiques communes avec les indexages pour les testeurs de théorèmes basés sur la saturation. Rümmer utilise des trigger avec d'une méthode classique de tableaux [Rüm12]. L'instanciation basée sur les triggers produit malheureusement de nombreuses instances non pertinentes. Pour aborder ce problème, une technique d'instanciation axée sur les objectifs produisant uniquement des instances utiles a été introduite par Reynolds et al. [RTdM14]. CCFV partage des similarités avec cet algorithme, la recherche étant basée sur la structure des termes et un modèle actuel provenant du solveur pour la partie sans quantificateurs. L'approche ici est cependant plus générale, la technique précédente étant une spécialisation. L'instanciation de quantificateurs basée sur les modèles de Ge and de Moura (MBQI) [GdM09] fournit une méthode complète pour la logique du premier ordre grâce à la dérivation successive d'instances conflictuelles afin d'affiner un modèle candidat pour l'ensemble de la formule, quantificateurs compris. Des méthodes alternatives pour la construction et la vérification des modèles ont été présentées par Reynolds et al. [RTG+13]. Ces deux approches basées sur le modèle [GdM09, RTG+13] permettent l'intégration des théories au-delà de l'égalité, tandis que CCFV ne gère pour l'instant que l'égalité et les fonctions non interprétées. Backeman et Rümmer résolvent le problème connexe de l'E-unification rigide à travers le codage dans SAT, en utilisant un solveur SAT *off-the-shelf* pour calculer des solutions [BR15b]. Notre travail est plus dans la lignée des techniques telles que celles de Goubault [Gou93] et Tiwari et al. [TBR00], les algorithmes de fermeture de congruence étant très efficaces pour vérifier les solutions, nous pensons qu'ils peuvent aussi constituer le noyau d'algorithmes efficaces pour les découvrir. CCFV diffère de ces techniques antérieures notamment, car elle gère les inégalités et que la recherche de solutions

est guidée par la structure d'un modèle pour la formule sans quantificateurs et est donc la plus appropriée pour un contexte SMT.

Un aspect très important dans le raisonnement automatisé est de fournir des certificats vérifiables par machine pour les résultats produits. Les applications externes utilisant des systèmes automatiques ne nécessitent généralement pas seulement la réponse au problème de satisfaction, mais aussi une justification. Par exemple, en vérifiant si un théorème est valide en essayant de montrer que sa négation est insatisfaisante, produire un modèle dans le cas d'une formule satisfaisable fournit un contre-exemple pour la conjecture. En outre, étant donné que ces systèmes peuvent être des logiciels complexes, souvent avec des centaines de milliers de lignes de code, il est difficile de s'assurer que les implémentations sont exemptes d'erreurs. Par conséquent, être en mesure de vérifier les résultats obtenus est d'une importance primordiale pour améliorer la fiabilité d'un tel outil.

Un nombre croissant de prouveurs de théorèmes automatiques peuvent générer des certificats, ou des preuves, qui justifient leurs résultats. Ces preuves peuvent être vérifiées par d'autres programmes et partagées entre les systèmes de raisonnement. La production de preuve est généralement bien comprise pour les méthodes de preuve de base (par exemple, la superposition, les tableaux, l'apprentissage de clauses axé sur les conflits) et pour de nombreuses théories couramment utilisées en SMT. Cependant, la plupart des prouveurs automatiques effectuent également un traitement ou un prétraitement de formule — comme la clausification et la réécriture avec des lemmes spécifiques à la théorie — et la production de preuve pour cet aspect est moins bien comprise. Pour la plupart des prouveurs, le code pour préprocesser les formules n'est pas particulièrement compliqué, mais il est long et traite une multitude de cas, dont certains sont rarement exécutés. Bien qu'il soit crucial pour l'efficacité, ce code tend à recevoir beaucoup moins d'attention que d'autres aspects des prouveurs. Les développeurs sont réticents à investir de l'effort en produisant des preuves détaillées pour un tel traitement, car cela nécessite d'adapter beaucoup de code. En conséquence, la granularité des inférences pour le traitement de la formule est souvent grossière. Parfois, les fonctionnalités de traitement sont même désactivées pour éviter les lacunes dans les preuves, au détriment de l'efficacité.

La deuxième contribution principale de cette thèse est un cadre pour résoudre ces problèmes. C'est un travail en commun avec Jasmin Blanchette et Pascal Fontaine [BBF17]. Les preuves sont exprimées à l'aide d'un ensemble extensible de règles d'inférence (Chapitre 6). Les règles ont une granularité fine, permettant de séparer les théories ou même des lemmes différents de la même théorie. La clausification, les simplifications théoriques spécifiques et l'expansion des expressions "let" sont des exemples de ce cadre. La skolemisation peut sembler problématique, mais avec l'aide de l'opérateur de choix de Hilbert, elle peut également être intégrée au cadre. Au cœur du cadre se trouve un algorithme de récurrence contextuelle générique qui traverse les termes à traduire (Chapitre 7). Le contexte fixe certaines variables, maintient une substitution et permet de suivre les polarités ou d'autres données. Le travail spécifique à la transformation, y compris la génération des preuves, est effectué par des fonctions de plugin qui sont données en

tant que paramètres du cadre. L'algorithme de récursivité, qui est essentiel pour la performance et l'exactitude des preuves générées, doit être implémenté une seule fois. Un autre avantage de l'architecture modulaire est que nous pouvons combiner facilement plusieurs transformations en une seule passe, sans compliquer le code ou compromettre le niveau de détail de la sortie de preuve.

Les règles d'inférence et l'algorithme de récurrence contextuelle bénéficient de nombreuses propriétés souhaitables. Nous montrons que les règles sont correctes et que le traitement des quantificateurs est correct même en présence de conflit dans les noms. De plus, en supposant des structures de données appropriées, nous montrons que la génération de preuves ajoute un coût proportionnel au temps passé à traiter les termes. Les preuves sont des graphes acycliques (DAG) et peuvent être produites avec une complexité de temps linéaire dans leur taille.

Nous avons mis en œuvre l'approche également dans veriT, un solveur SMT. Le solveur est connu pour ses preuves détaillées [BdODF09], qui sont reconstruites dans les assistants de preuve Coq [AFG+11] et Isabelle/HOL [BBF+16]. En tant que preuve de concept, nous avons mis en place un prototype de checker dans Isabelle/HOL. En adoptant le nouveau cadre, nous avons pu supprimer de grandes quantités de code compliqué dans le solveur, tout en permettant des preuves détaillées pour plus de transformations qu'auparavant. L'algorithme de récurrence contextuelle devait être implémenté une seule fois et plus complètement testé que l'une quelconque des transformations monolithiques qu'il remplace. Notre évaluation empirique révèle que veriT est aussi rapide qu'avant, même s'il génère maintenant des preuves plus fines.

Résolution de satisfiabilité modulo des théories

La pertinence du problème SMT provient du gain considérable d'expressivité obtenu en combinant la logique du premier ordre avec les théories. Le raisonnement efficace dans un tel cadre est d'une importance capitale pour une variété d'applications formelles qui peuvent s'appuyer sur la logique pour coder des problèmes. Cela permet d'appliquer le raisonnement assisté par ordinateur, par exemple, la vérification formelle, la synthèse des programmes, les tests automatiques, l'analyse de programme, etc.

L'accent mis sur les problèmes du monde réel impose une restriction aux solutions pertinentes à un problème SMT donné. On s'intéresse généralement uniquement à des solutions qui sont des extensions des modèles standard des théories d'arrière-plan. Donc dans la pratique, lors de la résolution, par exemple, d'un problème de satisfaction modulo l'arithmétique linéaire entière, les seules solutions souhaitées sont celles qui interprètent les symboles $+$, $-$, , etc., de la façon habituelle, et notamment sur le domaine des entiers. Un avantage de fixer des interprétations pour des théories, plutôt que de s'appuyer par exemple, sur des axiomatizations, est de pouvoir utiliser des procédures de décision ad hoc pour ces structures spécifiques, au lieu de s'appuyer sur des méthodes générales. Pour des combinaisons de théories sans quantification, cela conduit souvent à des procédures de décision hautement efficaces.

L'importance du problème SMT a conduit au domaine de la recherche communément connu comme "solution SMT" (voir [Seb07] ou [BSST09] pour des aperçus complets). Il est originaire du problème plus simple de satisfaction booléenne, c'est-à-dire déterminer si une formule propositionnelle a un modèle. Ce problème est communément connu sous le nom SAT, et est l'un des premiers à être montré NP-complet. En tant que tel, il s'agissait d'un problème insoluble pendant de nombreuses années, jusqu'à la révolution SAT au tournant du siècle dernier: les solveurs SAT modernes, mettant en œuvre le calcul de l'apprentissage par clause (CDCL) axé sur les conflits, peuvent résoudre efficacement des problèmes contenant des centaines de milliers de variables propositionnelles et des millions de clauses. La satisfaisabilité modulo théorie est apparue initialement dans les années 1970 et 1980 par des pionniers tels que Nelson et Oppen [NO79, NO80] et Shostak [Sho84]. L'accent a été mis sur les logiques sans quantificateur, qui sont souvent décidables. Les approches modernes ont débuté à la fin des années 1990, exploitant la puissance des solveurs SAT pour construire des systèmes plus évolutifs. Ces approches peuvent être classées en deux grandes classes. Dans le premier cas, un problème SMT est entièrement codé dans une instance SAT et fourni à un solveur SAT, sans autre intervention. Le raisonnement au niveau théorie n'est effectué que pour fournir un meilleur encodage dans SAT. Dans le second cas, le raisonnement au niveau théorie est effectué par une combinaison de procédures de décision qui sont entrelacées avec un solveur SAT. Le problème SMT est généré par le SAT solveur, qui fournit une solution qui n'est qu'un candidat pour le problème d'origine. Le cadre CDCL($\mathcal{T}$) [NOT06] est une telle approche pour la résolution de SMT. C'est l'approche la plus commune adoptée dans les solveurs SMT actuels. Nous nous plaçons dans ce cadre avec comme objectif de raisonner ausi sur des formules quantifiées.

Dans les logiques classiques, le double problème de satisfaction est celui de la démonstration du théorème: montrer qu'une formule est un théorème équivaut à montrer que sa négation est insatisfaisante, tout en trouvant un modèle pour sa négation fournit un contre-exemple à la conjecture. Le champ de la démonstration automatique de théorème (ATP) historiquement était axé sur la logique du premier ordre pure avec égalité, où le plus grand défi consiste à gérer efficacement les formules quantifiées. Le raisonnement avec des théories et en présence de quantificateurs pose des défis majeurs, car il est souvent incomplet et difficile à réaliser efficacement, même de manière heuristique. Néanmoins, de nombreux travaux ont tenté d'étendre les systèmes ATP avec des théories, au point que les différences entre les solveurs SMT et les systèmes ATP deviennent de plus en plus petites.

Dans la Section 3.1, nous détaillons le calcul CDCL($\mathcal{T}$). CDCL($\mathcal{T}$) est une extension du calcul de l'apprentissage par clause de conflit (CDCL) qui intègre le raisonnement avec des théories. CDCL est un raffinement de la procédure classique DPLL [DLL62] qui combine une recherche en arrière pour un modèle propositionnel avec un puissant moteur d'analyse de conflit. La recherche s'effectue en entrelacant les "décisions", c'est-à-dire l'attribution heuristiques de valeur de vérité à des propositions, avec l'application de la propagation unitaire jusqu'à un point fixe. Le retour arrière explicite est effectué au moyen d'une *clause apprise* dérivée lorsqu'un

conflit est atteint dans la recherche. La Section **??** détaille CDCL($\mathcal{T}$), qui étend ceci dans le cadre SMT. En présence de quantificateurs, le problème satisfaisabilité modulo théorie est souvent coûteux, voir indécidable (en fonction de la théorie). Historiquement, dans le cadre CDCL($\mathcal{T}$), des techniques d'instanciation heuristiques ont été utilisées pour résoudre ce problème. Elles permettent de dériver rapidement des instances de base des quantificateurs et de ramener le raisonnement au niveau du solveur sans quantificateur. Nous décrivons ici les principales techniques d'instanciation appliquées dans CDCL($\mathcal{T}$).

Instantiation

Un problème central lors de l'application de techniques d'instanciation pour gérer des formules quantifiées en CDCL($\mathcal{T}$) est comment déterminer les instances à dériver. Chacune des principales techniques, qu'il s'agisse des triggers, de conflits ou d'instanciation basée sur les modèles, contribuent grandement à l'efficacité des solveurs, mais chacune est généralement implémentée de manière indépendante, et avec ses propres structures de données.

Dans le Chapitre 4, nous présentons un cadre uniforme pour le raisonnement avec des formules quantifiées dans la théorie de l'égalité et des fonctions non interprétées dans CDCL($\mathcal{T}$). Nous présentons le problème d'E-ground (dis)unification comme algorithme central (Section 4.1), avec lequel les diverses techniques d'instanciation peuvent être implémentées. Ce problème concerne le problème classique de l'E-unification [DV98] rigide, un problème omniprésent dans le contexte des tableaux pour la logique équationnelle du premier ordre. En exploitant les similitudes entre CDCL($\mathcal{T}$) et ces cadres, nous construisons sur des techniques classiques pour résoudre l'E-unification et présentons une procédure de décision pour l'E-ground (dis)unification: l'algorithme de *Fermeture de congruence avec variables libres* (congruence closure with free variables, CCFV) étend l'algorithme classique de fermeture de congruence [NO80, NO07], pour tenir compte des variables libres (Section 4.2). Nous montrons ensuite comment se baser sur CCFV pour implémenter une instanciation basée sur les triggers, sur les conflits ou sur les modèles (Chapitre 5). Ces chapitres décrivent le travail commun avec Pascal Fontaine et Andrew Reynolds [BFR17].

Production de preuves pour le traitement des formules

Un nombre croissant de prouveurs de théorèmes automatiques peuvent générer des certificats, ou des preuves, qui justifient les formules qu'ils dérivent. Ces preuves peuvent être vérifiées par d'autres programmes et partagées entre les systèmes de raisonnement. Certains utilisateurs voudront également inspecter ces sorties pour comprendre pourquoi une formule est valide. La production de preuve est généralement bien comprise pour les méthodes de preuve de base et pour de nombreuses théories couramment utilisées en SMT. Mais la plupart des prouveurs automatiques effectuent également un traitement ou un prétraitement de formule — comme la clausification et la réécriture avec des lemmes spécifiques à la théorie — et la production de

preuve pour cet aspect est moins mature.

Pour la plupart des prouveurs, le code pour les formules de traitement est long et traite d'une multitude de cas, dont certains sont rarement exécutés. Bien qu'il soit crucial pour l'efficacité, ce code tend à recevoir beaucoup moins d'attention que d'autres aspects des prouveurs. Les développeurs sont réticents à investir de l'énergie pour produire des preuves détaillées pour un tel traitement, car cela nécessite d'adapter beaucoup de code. En conséquence, la granularité des inférences pour le traitement de la formule est souvent grossière. Parfois, les fonctionnalités de traitement sont même désactivées pour éviter les lacunes dans les preuves, à un coût élevé dans la performance de la recherche de preuve.

Les preuves à grain fin sont importantes pour diverses applications. Nous proposons un cadre pour générer de telles preuves sans ralentir la recherche de preuve. Le Chapitre 6 décrit un calcul de preuve dans lequel générer des preuves fines pour traiter des formules. Le chapitre 7 présente un cadre évolutif pour générer de telles preuves. Les deux chapitres décrivent le travail commun avec Jasmin Blanchette et Pascal Fontaine [BBF17].

Nous proposons un algorithme générique pour les transformations de termes, basé sur la récurrence structurale. L'algorithme est paramétré par quelques fonctions de plugin simples incorporant l'essence de la transformation. En combinant des fonctions de plugin compatibles, nous pouvons effectuer plusieurs transformations dans une traversée. Les transformations peuvent dépendre d'un contexte qui encapsule les informations pertinentes, telles que les variables liées, les substitutions variables et la polarité. Chaque transformation peut définir sa propre notion de contexte qui est traversée par la récurrence.

La sortie est générée par un module de preuve qui maintient une pile d'arbres de dérivation. La procédure *apply*(R, n, Γ, t, u) retire n arbres de dérivation $\bar{\mathcal{D}}_n$ de la pile et ajoute l'arbre

$$\frac{\mathcal{D}_1 \quad \cdots \quad \mathcal{D}_n}{\Gamma \rhd t \simeq u}\,\mathrm{R}$$

sur la pile. Les fonctions du plugin sont chargées d'invoquer *apply* selon le cas.

L'algorithme effectue une récurrence contextuelle de post-profondeur en profondeur sur le terme à traiter. Les sous-termes sont traités en premier; Alors un terme intermédiaire est construit à partir des sous-termes résultants et est traité lui-même. Le contexte est mis à jour de manière spécifique à la transformation avec chaque appel récursif. Les fonctions du plugin sont divisées en deux groupes: celles qui mettent à jour le contexte lorsque l'algorithme entre dans le corps d'un quantificateur ou lorsqu'il passe d'un symbole de fonction à l'un de ses arguments; et celles qui renvoient le terme traité et produisent la preuve correspondante en effet de bord.

Conclusion

Notre première contribution dans cette thèse a été l'introduction de CCFV, une procédure de décision pour l'E-ground (dis)unification. Nous avons montré comment les principales tech-

niques d'instanciation en SMT peuvent être basées sur elle. Nous avons également discuté de la manière d'implémenter efficacement CCFV dans un solver CDCL($\mathcal{T}$), un aspect crucial dans notre domaine de travail. Notre évaluation expérimentale montre que CCFV conduit à des améliorations significatives dans les solveurs CVC4 et veriT, ce qui fait que l'ancien dépasse l'état de l'art en SMT avec instanciation. Les calculs présentés sont très généraux, permettant différentes stratégies et optimisations, comme nous l'avons déjà mentionné.

Il existe trois directions dans lesquelles le travail présenté ici peut être amélioré: la résolution CCFV, la portée de CCFV et les techniques d'instanciation construites au delà de CCFV. Une direction pour améliorer la résolution est d'utiliser l'apprentissage de *lemmas* dans CCFV en profondeur d'abord, de la même manière que les solveurs SAT. Lorsqu'une branche ne parvient pas à produire une solution et qu'elle est rejetée, l'analyse des littéraux qui ont mené au conflit peut permettre le *backjump* plutôt qu'un simple backtracking, réduisant ainsi l'espace de recherche de la solution. L'algorithme *Complementary Congruence Closure* présenté par Backeman et Rümmer [BR15a] pourrait être étendu pour effectuer une telle analyse. Plus généralement, nous croyons qu'un CCFV incrémental permettrait des gains de performance significatifs, comme c'est le cas pour les solveurs de théorie. Un point de départ serait de construire sur des techniques telles que celles décrites par Moura et Bjørner [dMB07] pour effectuer un E-matching incremental dans Z3. La portée de CCFV peut être étendue en allant au-delà de la logique équationnelle du premier ordre. Pour supporter d'autres théories, les calculs doivent être augmentés avec des règles pour les nouveaux symboles interprétés. Pour gérer l'arithmétique linéaire, par exemple, CCFV devrait être capable d'effectuer l'unification modulo associativité, commutativité et identité additive). Les implémentations pratiques nécessiteront également du solveur qu'il fournisse des contrôles d'implantation efficaces en relation avec les nouveaux symboles interprétés, comme c'est le cas pour l'égalité. Si la décision est facilement perdue lors de la combinaison de théories dans une logique quantifiée, il faut trouver un bon équilibre entre les procédures de décision pour des fragments spécifiques, tels que le fragment essentiellement non interprété décrit par Ge et de Moura [GdM09], et des heuristiques efficaces pour des paramètres incomplets . Outre le raisonnement théorique, l'unification de l'ordre supérieur pourrait aussi être abordée par CCFV. Les calculs devraient tenir compte des applications partielles, des variables fonctionnelles et des abstractions lambda. Cela permettrait aux solveurs SMT de gérer de manière native les problèmes d'ordre supérieur, évitant ainsi l'exhaustivité et les problèmes de performance avec des encodages maladroits dans la logique du premier ordre. Une autre extension possible de CCFV est de gérer l'E-unification rigide, de façon à l'intégrer dans des tableaux et des techniques basées sur des calculs séquentiels tels que BREU [BR15b] ou la résolution de conflit [SP16]. Cela équivaut à avoir des égalités avec variables dans E, ce qui signifie un changement significatif de l'algorithme CCFV. Cependant, cela permettrait d'intégrer une procédure efficace axée sur les objectifs dans des calculs qui nécessitent la résolution de problèmes d'E-unification.

CCFV peut être considéré comme une première étape vers un "solveur théorique" pour que les formules quantifiées soient intégrées dans un solveur CDCL($\mathcal{T}$). Les techniques d'instanciation

basées sur CCFV évalueraient le modèle candidat avec des formules quantifiées et:

i) fournit un modèle fini lorsqu'il existe,

ii) dérive des cas qui réfutent le candidat lorsqu'il existe,

iii) propage les instanciations pertinentes et

iv) est incrémental par rapport aux affectations propositionnelles.

L'incrémentalité dépend de la mise en œuvre de CCFV. L'instanciation basée sur les conflits répond déjà partiellement à (ii) et (iii), tandis que l'instanciation basée sur les triggers s'insère dans (iii) et l'instanciation basée sur le modèle quelque part dans (i) et (ii). La façon d'étendre les techniques d'instanciation afin d'atteindre pleinement ces objectifs reste un problème ouvert. Une technique complète d'instanciation basée sur le conflit, au moins par rapport à la logique équationnelle de premier ordre, pourrait être obtenue en considérant des formules quantifiées simultanément au lieu d'indépendamment, c'est-à-dire en trouvant des substitutions σ telles que

$$E \models \neg\psi_1\sigma \vee \cdots \vee \neg\psi_n\sigma, \text{ avec } \mathcal{Q} = \{\forall\bar{x}.\, \psi_1, \ldots, \forall\bar{x}.\, \psi_n\}$$

Les techniques des tableaux et des calculs de superposition, qui considèrent nativement des formules quantifiées simultanément, pourraient être combinées avec CCFV pour résoudre l'implication ci-dessus.

Notre deuxième contribution a été d'introduire un cadre pour représenter et générer des preuves du traitement de la formule et sa mise en œuvre dans veriT et Isabelle/HOL. Le cadre centralise la question subtile de la manipulation des variables et des substitutions liées de manière efficace. Ce cadre est suffisamment souple pour tenir compte de nombreuses transformations intéressantes. Bien qu'il ait été implémenté dans un solveur SMT, il semble qu'il n'y ait pas de limitation intrinsèque qui empêcherait son utilisation dans d'autres types de prouveurs automatiques du premier ordre ou même d'ordre supérieur. Le cadre couvre de nombreuses techniques de prétraitement et peut faire partie d'une plus grande boîte à outils. Il permet à veriT, dont les preuves détaillées ont été l'une de ses caractéristiques définissantes, de produire des justifications plus détaillées que jamais. Cependant, il existe encore des transformations globales pour lesquelles les preuves sont inexistantes ou laissent beaucoup à désirer. Pour gérer ces transformations, notre notion de contexte devrait être étendue, ainsi que les opérations autorisées à être exécutées sur des termes, de manière à pouvoir aller au-delà du remplacement de termes par leur équivalent. Le cadre peut également être étendu pour gérer différentes constructions de langage, telles que les abstractions lambda. Compte tenu de la généralité du calcul du traitement et de l'algorithme producteur de preuves, des opérations telles que la β-réduction peuvent être facilement intégrées.

Nous avons mis l'accent ici sur la production de preuves détaillées, visant à garantir une vérification d'une complexité raisonnable et facilitant ainsi l'implémentation effective des correcteurs. Une direction prometteuse est également de reconstruire les preuves détaillées dans les assistants

de preuve, qui fournissent des preuves officielles fiables et vérifiées par machine des théorèmes. Un défi général consiste à effectuer dans les assistants de preuve le raisonnement nécessaire pour vérifier les preuves. Pour certaines théories, telles que l'arithmétique non linéaire, cela peut être très difficile. Cependant, une fois possible, la reconstitution complète des preuves dans des assistants de preuve augmenterait considérablement la confiance globale sur les résultats des solveurs automatiques, évitant ainsi la nécessité de faire confiance à un logiciel dont la correction repose principalement sur des tests.

En conclusion, nous croyons que nos contributions sont bénéfiques pour l'état de l'art de la résolution SMT (et par conséquent aussi pour la démonstration automatique au premier ordre). En outre, étant donné que les deux cadres présentés sont assez généraux et extensibles, nous nous attendons à ce qu'ils servent de point de départ pour plusieurs autres techniques d'instanciation et de production de preuve en SMT, aidant ainsi les solveurs à résoudre des problèmes toujours plus expressifs et à produire des résultats plus fiables.

Extended Abstract in Portuguese

Introdução

Em muitas aplicações de métodos formais, como verificação formal, síntese de programas, testes automáticos e análise de programas, é comum depender de lógicas para representar condições que precisam ser verificadas. Essas condições geralmente podem ser reduzidas para o problema *Satisfatibilidade Módulo Teorias* (SMT) (Capítulo 2): dada uma fórmula em lógica de primeira ordem, ela possui um modelo consistente com uma combinação de teorias? Os solucionadores SMT tornaram-se, assim, recursos populares para ferramentas que automatizam a verificação formal, a síntese de programas e assim por diante. Eles eliminam automaticamente as obrigações de prova — condições a severificar — e fornecem certificados de seus resultados, ou seja, modelos ou demonstrações. Nesta tese, objetivamos melhorar a eficiência dos solucionadores SMT e aumentar sua confiabilidade.

Os solucionadores SMT (Capítulo 3) foram projetados principalmente para resolver problemas sem quantificadores, nos quais eles são altamente eficientes e capazes de lidar com grandes fórmulas contendo símbolos interpretados. A lógica de primeira ordem pura quantificada é melhor tratada com técnicas baseadas em *resolução* e *superposição* [BG94, NR01]. Embora existam tentativas [dMB08c] em unificar tais técnicas com CDCL($\mathcal{T}$) [NOT06], o cálculo que os solucionadores SMT geralmente implementam, a abordagem mais comum ainda é *instanciação*: as fórmulas quantificadas são reduzidas a fórmulas sem quantificadores e refutadas com a ajuda de procedimentos de decisão para tais fórmulas. As principais técnicas de instanciação são E-matching, com base em gatilhos [DNS05, dMB07, Rüm12], procura de instâncias conflitantes [RTdM14] e instanciação de quantificadores baseada em modelos (MBQI) [GdM09, RTG+13]. Cada uma dessas técnicas contribui para a eficiência de solucionadores de última geração, mas cada uma delas é tipicamente implementada de forma independente.

Apresentamos o problema de E-ground (dis)unification como a base de um arcabouço único no qual todas essas técnicas podem ser acomodadas (Capítulo 4). Este problema relaciona-se com o problema clássico de E-unification [DV98] e é também NP-completo. Resolver o problema de E-ground (dis)unification equivale a encontrar substituições de modo que os literais que contenham variáveis livres sejam válidos de acordo com os literais sem variáveis atualmente no contexto. Uma vez que o domínio de instanciação dessas variáveis pode ser delimitado, uma possível maneira de resolver o problema é adivinhar, de forma não determinista, uma substituição

e verificar se ela é uma solução. O algoritmo *Fechamento de Congruência com Variáveis Livres* (CCFV, abreviado) apresentado aqui é um procedimento de decisão para este problema na prática, baseado no algoritmo clássico de fechamento de congruência [NO80, NO07]. É um procedimento orientado por objetivos: as soluções são construídas de forma incremental, levando em consideração o fechamento de congruência dos termos definidos pelas igualdades no contexto e as possíveis atribuições às variáveis. Nós então mostramos como se basear no CCFV para implementar as técnicas de instanciação baseadas em gatilhos, conflitos e modelos (Capítulo 5). Uma avaliação experimental é apresentada, na qual medimos o impacto das otimizações e técnicas de instanciação baseadas no CCFV. Mostramos que nossas implementações exibem melhorias em relação às abordagens de última geração. Este trabalho levou a uma publicação conjunta com Pascal Fontaine e Andrew Reynolds [BFR17].

Técnicas de instanciação para SMT vem sendo amplamente estudadas. A instanciação heurística baseada em E-matching dos gatilhos selecionados foi introduzida por Nelson [Nel80] e foi implementada com sucesso em vários solucionadores. Uma implementação altamente eficiente de E-matching no contexto de solucionadores SMT foi apresentada por Moura e Bjørner [dMB07]; baseia-se em algoritmos incrementais de matching, geração de código de máquina para otimizar o desempenho e elaboradas técnicas de indexação combinando recursos comuns em índices para demonstradores de teoremas baseados em saturação. Rümmer usa gatilhos junto com um método clássico de tableaux [Rüm12]. A instanciação baseada em gatilhos, infelizmente, produz muitas instâncias irrelevantes. Para enfrentar este problema, uma técnica de instanciação orientada por objetivos que produz apenas instâncias relevantes foi apresentada por Reynolds et al. [RTdM14]. CCFV possuis semelhanças com este algoritmo, com sua busca sendo baseada na estrutura de termos e em um modelo atual proveniente do solucionador sem quantificaderos. A abordagem aqui é, no entanto, mais geral, da qual esta técnica anterior é uma especialização. A instanciação de quantificadores baseada em modelos (MBQI) de Ge e de Moura (MBQI) [GdM09] fornece um método completo para a lógica de primeira ordem através da derivação sucessiva de instâncias conflitantes para refinar um candidato a modelo para toda a fórmula, incluindo quantificadores. Ele permite ao solucionador encontrar modelos finitos quando estes existem. A verificação do modelo é realizada com uma cópia separada do solucionador SMT em busca de uma instância conflitante. Métodos alternativos para a construção e verificação do modelo foram apresentados por Reynolds et al. [RTG+13]. Essas abordagens baseadas em modelos [GdM09, RTG+13] permitem a integração de teorias além da igualdade, enquanto que CCFV por enquanto apenas lida com igualdade e funções não interpretadas. Backeman e Rümmer resolvem o problema relacionado de E-unification através da sua codificação em SAT, usando um solucionador SAT off-the-shelf para calcular soluções [BR15b]. Nosso trabalho está mais em linha com técnicas como as de Goubault [Gou93] e Tiwari et al. [TBR00], orientadas por objetivos; uma vez que os algoritmos de fechamento de congruência são muito eficientes na verificação de soluções, acreditamos que eles também podem ser o núcleo de algoritmos eficientes para descobri-las. CCFV difere das técnicas anteriores de forma notável, já que lida com desigualdades e que restringe

a busca de soluções com base na estrutura de um modelo para a fórmula sem quantificadores, sendo portanto mais adequada para um contexto SMT.

Um aspecto muito importante no raciocínio automatizado é fornecer certificados para os resultados produzidos que possam ser verificados por máquinas. As aplicações externas que utilizam sistemas automáticos geralmente não precisam apenas das respostas ao problema de satisfação, mas também de suas justificativas. Por exemplo, ao se tentar verificar se um teorema é válido através da demonstração de que sua negação é insatisfatível, produzir um modelo caso a fórmula seja satisfatível fornece um contra-exemplo para a conjectura. Além disso, uma vez que esses solucionadores podem ser sistemas com uma engenharia bastante complexa, muitas vezes com centenas de milhares de linhas de código, é difícil garantir que as implementações sejam livres de erros. Portanto, ser capaz de verificar os resultados produzidos é de suma importância para melhorar a confiabilidade de qualquer dessas ferramentas.

Um número crescente de demonstradores automáticos de teoremas pode gerar certificados ou demonstrações que justifiquem seus resultados. Essas demonstrações podem ser verificadas por outros programas e compartilhadas entre sistemas de raciocínio. A produção de demonstrações geralmente é bem compreendida para os métodos principais de demonstração (por exemplo, superposição, tableaux, aprendizagem de cláusulas orientada por conflitos) e para muitas teorias comumente usadas em SMT. No entanto, a maioria dos demonstradores automáticos também realizam algum processamento ou pré-processamento de fórmulas — por exemplo clausificação e reescrita com lemas específicos de teorias — e a produção de demonstrações para estas técnicas é menos madura. Para a maioria dos demonstradores, o código para processamento de fórmulas não é particularmente complicado, mas é longo e lida com uma infinidade de casos, alguns dos quais raramente são executados. Embora seja crucial para a eficiência, este código tende a receber muito menos atenção do que outros aspectos dos demonstradores. Os desenvolvedores são relutantes em investir esforço na produção de demonstrações detalhadas para esse processamento, pois isso requer a adaptação de uma grande quantidade de código. Como resultado, a granularidade das inferências para o processamento da fórmula é muitas vezes grosseira. Às vezes, os recursos de processamento são até desativados para evitar lacunas nas demonstrações, com alto custo no desempenho da procura de demonstrações.

A segunda contribuição principal desta tese é um arcabouço para abordar lidar com esses problemas. Este trabalho levou a uma publicação conjunta com Jasmin Blanchette e Pascal Fontaine [BBF17]. As demonstrações são expressadas usando um conjunto extensível de regras de inferência (Capítulo 6). As regras têm uma granularidade fina, possibilitando uma clara separação entre teorias ou até mesmo lemmas diferentes da mesma teoria. Clausificação, simplificações específicas de teorias e expansão das expressões de "let" são exemplos de técnicas acomodadas neste arcabouço. A Skolemização pode parecer problemática, mas com a ajuda do operador de escolha de Hilbert ela também pode ser integrada no arcabouço. No coração do arcabouço encontra-se um algoritmo genérico de recursão contextual que atravessa os termos a serem processados (Capítulo 7). O contexto fixa algumas variáveis, mantém uma substituição e

acompanha as polaridades ou outros dados. O trabalho específico de transformação, incluindo a geração de demonstrações, é executado por funções plugáveis que são fornecidas como parâmetros para o arcabouço. O algoritmo de recursão, que é crítico para o desempenho e a correção das demonstrações produzidas, precisa ser implementado apenas uma vez. Outro benefício da arquitetura modular é que podemos combinar facilmente várias transformações em uma única passagem, sem complicar indevidamente o código ou comprometer o nível de detalhe da demonstração produzida.

As regras de inferência e o algoritmo de recursão contextual possuem muitas propriedades desejáveis. Mostramos que as regras são corretas e que o tratamento de binders é correto mesmo na presença de conflitos de nomes. Além disso, assumindo estruturas de dados adequadas, mostramos que a geração de demonstrações cria apenas uma sobrecarga que é proporcional ao tempo gasto no processamento dos termos. A verificação de demonstrações representadas como grafos direcionadas acíclicos (DAGs) pode ser realizada com uma complexidade de tempo que é linear em seu tamanho.

Implementamos a abordagem também no solucionador SMT veriT. O solucionador é conhecido por suas demonstrações detalhadas [BdODF09], que são reconstruídas nos assistentes de demonstração Coq [AFG+11] e Isabelle/HOL [BBF+16]. Para validar o conceito, implementamos um protótipo para checagem em Isabelle/HOL. Ao adotar o novo arcabouço, conseguimos remover grandes quantidades de código complicado no solucionador, ao mesmo tempo em que permitimos a produção de demonstrações detalhadas para mais transformações do que antes. O algoritmo de recursão contextual teve que ser implementado apenas uma vez e é testado de forma mais completa do que qualquer uma das transformações monolíticas que ele substitui. Nossa avaliação empírica revela que veriT é tão rápido quanto antes, embora ele agora gere demonstrações de melhor qualidade.

Resolução de satisfatibilidade módulo teorias

A relevância do problema SMT vem do ganho considerável de expressividade alcançado pela combinação de lógica de primeira ordem com teorias. Raciocinar de forma eficiente em tal cenário é de extrema importância para uma variedade de métodos formais que podem depender de lógica para codificar problemas. Isso permite que o raciocínio assistido por computador seja aplicado em, por exemplo, verificação formal, síntese de programas, testes automáticos, análise de programas e assim por diante.

O foco em problemas do mundo real impõe uma restrição às soluções relevantes para um determinado problema SMT. Geralmente busca-se apenas soluções que são extensões dos modelos padrão das teorias. Então, na prática, ao resolver, por exemplo, um problema de satisfação módulo aritmética linear com inteiros, as únicas soluções desejadas são aquelas que interpretam os símbolos $+$, $-$, $<$, etc., da maneira usual, no domínio dos inteiros. Um benefício de fixar interpretações para teorias, ao invés de depender por exemplo de axiomatizações, é poder usar

procedimentos (de decisão) ad-hoc para essas estruturas específicas, em vez de depender de métodos de propósito geral. Para combinações de teorias livres de quantificadores, isto geralmente leva a procedimentos de decisão altamente eficientes.

A importância do problema SMT levou ao campo de pesquisa comumente conhecido como "Resolução SMT" (veja [Seb07] ou [BSST09] para obter mais detalhes). Ele se originou com o problema mais simples de satisfatibilidade Booleana, isto é, determinar se uma fórmula proposicional possui um modelo. Este problema é comumente conhecido como SAT, e é um dos primeiros demonstrados ser NP-completo. Como tal, foi um problema intratável por muitos anos, até a chamada revolução SAT na virada do século passado: os modernos solucionadores SAT, implementando o cálculo de aprendizagem de cláusulas orientada por conflitos (CDCL), podem resolver de forma eficiente problemas muito grandes provenientes de problemas do mundo real contendo centenas de milhares de variáveis proposicionais e milhões de cláusulas. A resolução SMT foi desenvolvida inicialmente nos anos 1970 e 1980 por pioneiros como Nelson e Oppen [NO79, NO80] e Shostak [Sho84]. O foco desde então foi geralmente em lógicas livres de quantificadores, que são muitas vezes decidíveis. As abordagens modernas começaram no final da década de 1990, aproveitando o poder dos solucionadores SAT para construir sistemas com maior escalabilidade. Essas abordagens podem ser classificadas em duas grandes classes. Na primeiro, um problema SMT é totalmente codificado em uma instância SAT e resolvido com um solucionador SAT, sem intervenção adicional. O raciocínio para teorias é realizado apenas para fornecer uma melhor codificação para SAT. Na segunda, o raciocínio para teorias é realizado por uma combinação de procedimentos de decisão que são intercalados com um solucionador SAT. O problema SMT é abstraído como uma instância SAT, cuja solução é apenas uma candidata para o problema original: uma conjunção proposicionalmente consistente de literais, que é então verificada de acordo com a consistência módulo teorias. O arcabouço CDCL($\mathcal{T}$) [NOT06] é uma abordagem para a resolução SMT. É o mais comum adotado para sistemas que executam a resolução SMT automática. Nosso foco é neste arcabouço e, mais importante ainda, sobre como raciocinar de forma eficiente com fórmulas quantificadas nele.

Na lógica clássica, o duplo problema da satisfação é o da demonstração de teoremas: mostrar que uma fórmula é um teorema é equivalente a mostrar que sua negação é insatisfatível, ao encontrar um modelo para a sua negação fornece-se um contra-exemplo para a conjectura. O campo de demonstração automática de teoremas (ATP) historicamente foi focado em lógica de primeira ordem pura e equacional, onde o maior desafio é como gerenciar eficientemente fórmulas quantificadas. O raciocínio de teorias na presença de quantificadores traz grandes desafios, pois muitas vezes é incompleto e difícil de ser realizado de forma eficiente, mesmo de forma heurística. No entanto, inúmeros trabalhos tentaram estender os sistemas ATP com raciocínio de teorias, ao ponto de as diferenças entre solucionadores SMT e sistemas ATP serem cada vez menores.

Na Seção 3.1 detalhamos o cálculo CDCL($\mathcal{T}$), que é uma extensão do cálculo de aprendizagem de cláusulas de conflito (CDCL) que acomoda o raciocínio de teorias. CDCL é um refinamento do clássico procedimento DPLL [DLL62] que combina um poderoso mecanismo de análise de

conflitos com a busca, baseada em backtracking de um modelo proposicional. A busca é realizada através da intercalação de "decisões", ou seja, atribuições heurísticas de literais, com a aplicação de propagação de unidade até um ponto fixo. Backtracking é realizada de forma explícita por meio de uma *cláusula aprendida* derivada quando um conflito é atingido na busca. A Seção 3.2 detalha o raciocínio com quantificadores em CDCL($\mathcal{T}$). Raciocinar com quantificadores na presença de teorias é muitas vezes dispendioso e em geral indecidível. Historicamente, no arcabouço CDCL($\mathcal{T}$), geralmente técnicas de instanciação heurística foram usadas para enfrentar esse problema. Elas permitem derivar rapidamente as instâncias dos quantificadores e transferir o raciocínio de volta para o eficiente solucionador livre de quantificadores. Aqui descrevemos as principais técnicas de instanciação aplicadas em CDCL($\mathcal{T}$).

Instanciação

Um problema central ao aplicar técnicas de instanciação em CDCL($\mathcal{T}$) é como determinar quais instâncias derivar. Cada uma das principais técnicas, seja a baseada em gatilhos, conflitos ou modelos, contribui muito para a eficiência de solucionadores de última geração, mas cada uma delas é tipicamente implementada de forma independente e ad hoc em seus próprios arcabouços.

No Capítulo 4 apresentamos um arcabouço uniforme para o raciocínio com fórmulas quantificadas em CDCL($\mathcal{T}$) na teoria da igualdade e funções não interpretadas. Apresentamos o problema E-ground (dis)unification como a base deste arcabouço (Seção 4.1), em que as técnicas de instanciação baseadas em gatilhos, conflitos e modelos podem ser acomodadas. Este problema relaciona-se com o problema clássico de E-unification [DV98], pervasivo em cálculos para lógica equacional de primeira ordem baseados em sequentes e tableaux. Ao explorar as semelhanças entre CDCL($\mathcal{T}$) e esses arcabouços, nos baseamos em técnicas convencionais para solucionar E-unification e apresentamos um procedimento de decisão para E-ground (dis)unification: *Fechamento de Congruência com Variáveis Livres* (CCFV, abreviado), que estende o clássico algoritmo de fechamento de congruência [NO80, NO07], para acomodar variáveis livres (Seção 4.2). Em seguida, mostramos como se basear em CCFV para executar instanciações baseadas em gatilhos, conflitos e modelos. Um relatório detalhado da implementação e avaliação experimental de CCFV e as técnicas de instanciação em torno dele é apresentada no Capítulo 5.

Produção de demonstrações para o processamento de fórmulas

Um número crescente de demonstradores automáticos de teorema pode gerar certificados ou demonstrações que justifiquem seu funcionamento. Essas demonstrações podem ser verificadas por outros programas e compartilhadas entre sistemas de raciocínio. Alguns usuários também querem inspecionar estes resultados para entender por que uma fórmula é válida. A produção de demonstrações geralmente é bem compreendida para os principais métodos de demonstração e para muitas teorias comumente usadas em SMT. Mas a maioria dos demonstradores automáticos

também executam algum processamento ou pré-processamento de fórmulas — como clausificação e reescrita com lemas específicos de teorias — e a produção de demonstração para este aspecto é menos madura.

Para a maioria dos demonstradores, o código para processamento de fórmulas é longo e lida com uma infinidade de casos, alguns dos quais raramente são executados. Embora seja crucial para a eficiência, este código tende a receber muito menos atenção do que outros aspectos dos provers. Os desenvolvedores são relutantes em investir tempo na produção de demonstrações detalhadas para esse processamento, pois isso requer a adaptação de muito código. Como resultado, a granularidade das inferências para o processamento da fórmula é muitas vezes grosseira. Às vezes, os recursos de processamento são até desativados para evitar lacunas nas provas, com alto custo no desempenho da procura de prova.

As provas de granularidade fina são importantes para uma variedade de aplicações. Propomos um arcabouço para gerar tais demonstrações sem desacelerar a busca por demonstrações. O Capítulo 6 descreve um cálculo de demonstrações com o qual gerar demonstrações detalhadas para o processamento de fórmulas. O Capítulo 7 apresenta um arcabouço escalável para gerar tais demonstrações.

Propomos um algoritmo genérico para transformações de termos, com base em recursão estrutural. O algoritmo é parametrizado por algumas funções plugáveis simples que incorporam a essência da transformação. Ao combinar funções plugáveis compatíveis, podemos realizar várias transformações em uma única passagem. As transformações podem depender de algum contexto que encapsula informações relevantes, como variáveis fixadas, substituições de variáveis e polaridade. Cada transformação pode definir sua própria noção de contexto que é transmitida através da recursão.

O resultado é gerado por um módulo de demonstração que mantém uma pilha de árvores de derivações. O procedimento *apply*(R, n, Γ, t, u) retira n árvores de derivação $\bar{\mathcal{D}}_n$ da pilha e insere a árvore

$$\frac{\mathcal{D}_1 \quad \cdots \quad \mathcal{D}_n}{\Gamma \rhd t \simeq u}\,\mathrm{R}$$

na pilha. As funções plugáveis são responsáveis por invocar *apply*, como apropriado.

O algoritmo executa uma recursão contextual em pós-ordem e em profundidade no termo a ser processado. Subtermos são processados primeiro; então um termo intermediário é construído a partir dos subtermos resultantes e é ele mesmo processado. O contexto é atualizado de uma maneira específica para a transformação com cada chamada recursiva. Ele é abstrato do ponto de vista do algoritmo. As funções plugáveis são divididas em dois grupos: aquelas que atualizam o contexto ao inserir o corpo de um binder ou quando se deslocam de um símbolo de função para um dos seus argumentos; e aquelas que retornam o termo processado e produzem a demonstração correspondente como um efeito colateral.

Conclusão

Nossa primeira contribuição nesta tese foi a introdução de CCFV, um procedimento de decisão para E-ground (dis)unification. Nós mostramos como as principais técnicas de instanciação de resolução SMT podem ser baseadas nele. Nós também discutimos como implementar eficientemente CCFV em um solucionador CDCL($\mathcal{T}$), um aspecto crucial em nosso campo de trabalho. Nossa avaliação experimental mostra que CCFV leva a melhorias significativas nos solucionadores CVC4 e veriT, fazendo com que o anterior superasse o estado da arte na resolução SMT baseada em instanciação e o último se torne competitivo em vários benchmarks. Os cálculos apresentados são muito gerais, permitindo diferentes estratégias e otimizações, conforme discutido anteriormente.

Existem três direções nas quais o trabalho aqui apresentado pode ser melhorado: a resolução CCFV, o alcance de CCFV e as técnicas de instanciação baseadas nele. Uma direção para melhorar a resolução é usar *aprendizagem de lemas* no CCFV com busca em profundidade, de forma semelhante à que os solucionadores SAT realizam. Quando um ramo não consegue produzir uma solução e é descartado, analisar os literais que levaram ao conflito pode permitir *backjumping* ao invés de simples backtracking, reduzindo ainda mais o espaço de busca da solução. O *Fechamento de Congruência Complementar* apresentado por Backeman e Rümmer [BR15a] poderia ser estendido para realizar essa análise. De forma mais geral, acreditamos que implementar CCFV de forma incremental permitiria ganhos de desempenho significativos, como acontece com os solucionadores para raciocínio de teorias. Um ponto de partida seria construir técnicas como as descritas por Moura e Bjørner [dMB07] para realizar E-matching incremental em Z3. O alcance de CCFV pode ser estendido indo além da lógica equacional de primeira ordem. Para lidar com outras teorias, os cálculos devem ser estendidos com regras para os novos símbolos interpretados. Para lidar com a aritmética linear, por exemplo, CCFV deve ser capaz de realizar unificação AC1 (unificação módulo associatividade, comutatividade e identidade aditiva). Implementações práticas também exigirão que o solucionador livre de quantificadores permita verificações eficientes de consequência com relação aos novos símbolos interpretados, como é o caso para igualdade. Além disso, já que decidibilidade é facilmente perdida ao combinar teorias em lógica quantificada, deve-se obter um bom equilíbrio entre procedimentos de decisão para fragmentos específicos, como o fragmento essencialmente não interpretado descrito por Ge e de Moura [GdM09], e heurísticas efetivas para configurações incompletas. Além do raciocínio de teorias, a unificação de ordem superior também pode ser tentada com extensões de CCFV. Os cálculos precisariam lidar com aplicações parciais, variáveis funcionais e abstrações com lambdas. Isso permitiria que os solucionadores SMT raciocinassem com problemas de ordem superior, evitando a incompletude e problemas de desempenho com codificações desajeitadas em lógica de primeira ordem. Outra extensão possível de CCFV é lidar com E-unification, permitindo a sua integração em técnicas baseadas em cálculos de sequentes, como BREU [BR15b] ou resolução de conflitos [SP16]. Isso equivale a ter igualdades envolvendo variáveis em E, o que significa mudar significativamente como CCFV funciona atualmente. No entanto, permitiria integrar um

eficiente procedimento orientado por objetivos em cálculos que exigem a resolução de problemas de E-unification.

CCFV pode ser visto como um primeiro passo para um "solucionador de teorias" para fórmulas quantificadas a ser integrado em um solucionador CDCL($\mathcal{T}$). As técnicas de instanciação baseadas em CCFV avaliariam o modelo candidato com fórmulas quantificadas e iriam:

i) fornecer um modelo finito quando existisse,

ii) derivar instâncias refutando o candidato quando existissem,

iii) propagar instâncias relevantes e

iv) ser incremental em relação às atribuições proposicionais.

A incrementalidade depende da implementação de CCFV. A instanciação baseada em conflitos contempla parcialmente (ii) e (iii), enquanto a instanciação baseada em gatilhos se encaixa em (iii) e instanciação baseada em modelos um pouco em (i) e (ii). Como estender as técnicas de instanciação para atingir esses objetivos permanece um problema em aberto. Uma técnica completa de instanciação baseada em conflito, pelo menos em relação à lógica equacional de primeira ordem, poderia ser alcançada considerando fórmulas quantificadas simultaneamente em vez de independentemente, isto é, encontrando substituições *sigma* tal que

$$E \models \neg\psi_1\sigma \vee \cdots \vee \neg\psi_n\sigma, \text{ with } \mathcal{Q} = \{\forall\bar{x}.\, \psi_1, \ldots, \forall\bar{x}.\, \psi_n\}$$

Técnicas de tableaux e cálculos de superposição, que nativamente consideram fórmulas quantificadas simultaneamente, poderiam ser combinadas com CCFV para resolver o problema acima.

Nossa segunda contribuição foi apresentar um arcabouço para representar e gerar demonstrações do processamento de fórmulas e sua implementação em veriT e Isabelle/HOL. A estrutura centraliza a sutil questão de manipular variáveis vinculadas e substituições de forma correta e eficiente, e é flexível o suficiente para acomodar muitas transformações interessantes. Embora tenha sido implementado em um solucionador SMT, parece não haver uma limitação intrínseca que evite seu uso em outros tipos de demonstradores automáticos de primeira ordem, ou mesmo de ordem superior. O arcabouço cobre muitas técnicas de pré-processamento e pode ser parte de uma caixa de ferramentas maior. Ele permite que veriT, cujas demonstrações detalhadas tem sido uma de suas características definidoras, possa produzir justificativas mais detalhadas do que nunca. No entanto, ainda existem algumas transformações globais para as quais as demonstrações são inexistentes ou deixam muito a desejar. Para lidar com essas transformações nossa noção de contexto teria que ser estendida, bem como as operações que podem ser realizadas em termos, de tal forma que possamos ir além da substituição de iguais por iguais. A estrutura também pode ser estendida para lidar com diferentes construções de linguagem, como abstrações com lambdas. Dada a generalidade do cálculo de processamento e o algoritmo de produção de demonstrações, as operações como redução beta podem ser facilmente integradas. O último, por exemplo, é apenas uma variação da expansão de "let".

Nosso foco aqui tem sido a produção de demonstrações detalhadas, com o objetivo de garantir que a verificação de demonstrações tenha uma complexidade razoável e assim facilitar a implementação efetiva de verificadores. Uma direção promissora é também reconstruir as demonstrações detalhadas em assistentes de demonstração, que fornecem demonstrações formais de teoremas que são confiáveis e verificáveis por máquinas. Um desafio geral é executar dentro dos assistentes de demonstração o raciocínio necessário para verificar as demonstrações. Para algumas teorias, como a aritmética não linear, isso pode ser bastante desafiador. Uma vez possível, a reconstrução total de demonstrações em assistentes de demonstração aumentaria consideravelmente a confiança geral sobre os resultados de solucionadores automáticos, evitando assim a necessidade de se confiar em programas que dependem principalmente de testes para executar raciocínio lógica de forma correta.

Para concluir, acreditamos que nossas contribuições são benéficas para o estado-da-arte da resolução SMT (e, conseqüentemente, também da demonstração de teorema em lógica de primeira ordem). Além disso, uma vez que ambos os arcabouços apresentados são bastante gerais e extensíveis, esperamos que eles sirvam como ponto de partida para várias outras técnicas de instanciação e produção de demonstrações na resolução SMT, tornando solucionadores capazes de enfrentar problemas cada vez mais expressivos e produzindo resultados mais confiáveis.

Chapter 1

Introduction

In many formal methods applications, such as formal verification, program synthesis, automatic testing, and program analysis, it is common to rely on logics to represent conditions that need to be asserted. These conditions can often be reduced to the *Satisfiability Modulo Theories* (SMT) problem (Chapter 2): given a first-order logic formula, does it have a model consistent with a combination of background theories? SMT solvers have thus become popular backends for tools automating formal verification, program synthesis and so on. They automatically discharge proof obligations — conditions to be checked — and provide certificates of their results, i.e. models or proofs. In this thesis we aim both to improve the efficiency of SMT solvers and to increase their reliability.

SMT solvers (Chapter 3) have been primarily designed to solve quantifier-free problems, on which they are highly efficient and capable of handling large formulas with interpreted symbols. Pure quantified first-order logic is best handled with *resolution* and *superposition* based theorem proving [BG94, NR01]. Although there are first attempts [dMB08c] to unify such techniques with CDCL($\mathcal{T}$) [NOT06], the calculus that SMT solvers usually implement, the most common approach is still *instantiation*: quantified formulas are reduced to ground ones and refuted with the help of decision procedures for ground formulas. The main instantiation techniques are E-matching based on triggers [DNS05, dMB07, Rüm12], finding conflicting instances [RTdM14] and model based quantifier instantiation (MBQI) [GdM09, RTG+13]. Each of these techniques contributes to the efficiency of state-of-the-art solvers, yet each one is typically implemented independently.

We introduce the E-ground (dis)unification problem as the cornerstone of a unique framework in which all these techniques can be cast (Chapter 4). This problem relates to the classic problem of rigid E-unification [DV98] and is also NP-complete. Solving E-ground (dis)unification amounts to finding substitutions such that literals containing free variables hold in the context of currently asserted ground literals. Since the instantiation domain of those variables can be bound, a possible way of solving the problem is by first non-deterministically guessing a substitution and checking if it is a solution. The *Congruence Closure with Free Variables* algorithm (CCFV, for short) presented here is a practical decision procedure for this problem based on the

classic congruence closure algorithm [NO80, NO07]. It is goal-oriented: solutions are constructed incrementally, taking into account the congruence closure of the terms defined by the equalities in the context and the possible assignments to the variables. We then show how to build on CCFV to implement trigger based, conflict based and model based instantiation (Chapter 5). An experimental evaluation is presented, in which we measure the impact of optimizations and instantiation techniques based on CCFV. We show that our implementations exhibit improvements over state-of-the-art approaches. This work led to a joint publication with Pascal Fontaine and Andrew Reynolds [BFR17].

Instantiation techniques for SMT have been extensively studied. Heuristic instantiation based on E-matching of selected triggers was first introduced by Nelson [Nel80] and has been successfully implemented in several solvers. A highly efficient implementation of E-matching in the context of SMT solvers was presented by de Moura and Bjørner [dMB07]; it relies on incremental matching algorithms, generation of machine code for optimizing performance, and elaborated indexing techniques combining features common in indexes for saturation based theorem provers. Rümmer uses triggers alongside a classic tableaux method [Rüm12]. Trigger based instantiation unfortunately produces many irrelevant instances. To tackle this issue, a goal-oriented instantiation technique producing only useful instances was introduced by Reynolds et al. [RTdM14]. CCFV shares resemblance with this algorithm, since its search is also based on the structure of terms and the use of the model coming from the ground solver. The approach here is however more general, of which this previous technique is a specialisation. Ge and de Moura's model based quantifier instantiation (MBQI) [GdM09] provides a complete method for equational first-order logic and other fragments through successive derivation of conflicting instances to refine a candidate model for the whole formula, including quantifiers. It allows the solver to find finite models when they exist. Model checking is performed with a separate copy of the ground SMT solver searching for a conflicting instance. Alternative methods for model construction and checking were presented by Reynolds et al. [RTG+13]. Both these model based approaches [GdM09, RTG+13] allow integration of theories beyond equality, while CCFV for now only handles equality and uninterpreted functions. Backeman and Rümmer solve the related problem of rigid E-unification through encoding into SAT, using an off-the-shelf SAT solver to compute solutions [BR15b]. Our work is more in line with goal-oriented techniques as those by Goubault [Gou93] and Tiwari et al. [TBR00]; congruence closure algorithms being very efficient at checking solutions, we believe they can also be the core of efficient algorithms to discover them. CCFV differs from those previous techniques notably, since it handles disequalities and since the search for solutions is pruned based on the structure of a ground model and is thus most suitable for an SMT context.

A very important aspect in automated reasoning is to provide machine checkable certificates for produced results. External applications using automatic systems generally need not only the answer to the satisfiability problem but also a justification. For example, when checking if a theorem is valid by trying to show that its negation is unsatisfiable, producing a model in the

case a satisfiable formula provides a counter-example for the conjecture. Moreover, since these systems can be really complex engineering softwares, often with hundreds of thousands of lines of code, it is hard to ensure that implementations are free of errors. Therefore being able to verify the produced results is of paramount importance for improving the reliability of any such tool.

An increasing number of automatic theorem provers can generate certificates, or proofs, that justify their results. These proofs can be checked by other programs and shared across reasoning systems. Proof production is generally well understood for the core proving methods (e.g., superposition, tableaux, conflict-driven clause learning) and for many theories commonly used in satisfiability modulo theories (SMT). However, most automatic provers also perform some formula processing or preprocessing—such as clausification and rewriting with theory-specific lemmas—and proof production for this aspect is less mature. For most provers, the code for processing formulas is not particularly complicated, but it is lengthy and deals with a multitude of cases, some of which are rarely executed. Although it is crucial for efficiency, this code tends to be given much less attention than other aspects of provers. Developers are reluctant to invest effort in producing detailed proofs for such processing, since this requires adapting a lot of code. As a result, the granularity of inferences for formula processing is often coarse. Sometimes, processing features are even disabled to avoid gaps in proofs, at a high cost in proof search performance.

The second main contribution of this thesis is a framework to address these issues. This work led to a joint publication with Jasmin Blanchette and Pascal Fontaine [BBF17]. Proofs are expressed using an extensible set of inference rules (Chapter 6). The succedent of a rule is an equality between the original term and the translated term. (It is convenient to consider formulas as a special case of terms.) The rules have a fine granularity, making it possible to cleanly separate theories or even different lemmas from the same theory. Clausification, theory-specific simplifications, and expansion of 'let' expressions are instances of this framework. Skolemization may seem problematic, but with the help of Hilbert's choice operator, it can also be integrated into the framework. Some provers provide very detailed proofs for parts of the solving, but we are not aware of any publications about practical attempts to provide easily reconstructible proofs for processing formulas containing quantifiers and 'let' expressions. At the heart of the framework lies a generic contextual recursion algorithm that traverses the terms to translate (Chapter 7). The context fixes some variables, maintains a substitution, and keeps track of polarities or other data. The transformation-specific work, including the generation of proofs, is performed by plugin functions that are given as parameters to the framework. The recursion algorithm, which is critical for the performance and correctness of the generated proofs, needs to be implemented only once. Another benefit of the modular architecture is that we can easily combine several transformations in a single pass, without unduly complicating the code or compromising the level of detail of the proof output.

The inference rules and the contextual recursion algorithm enjoy many desirable properties.

We show that the rules are sound and that the treatment of binders is correct even in the presence of name clashes. Moreover, assuming suitable data structures, we show that proof generation adds an overhead that is proportional to the time spent processing the terms. Checking proofs represented as directed acyclic graphs (DAGs) can be performed with a time complexity that is linear in their size.

We implemented the approach also in the SMT solver veriT. The solver is known for its detailed proofs [BdODF09, BFT11, DFP11], which are reconstructed in the proof assistants Coq [AFG+11] and Isabelle/HOL [BBF+16]. As a proof of concept, we implemented a prototype checker in Isabelle/HOL. By adopting the new framework, we were able to remove large amounts of complicated code in the solver, while enabling detailed proofs for more transformations than before. The contextual recursion algorithm had to be implemented only once and is more thoroughly tested than any of the monolithic transformations it subsumes. Our empirical evaluation reveals that veriT is as fast as before even though it now generates finer-grained proofs.

Chapter 2

Conventions and definitions

We introduce here the logical framework and conventions used throughout this thesis. Our setting is a many-sorted classical first-order logic with equality (see e.g. Fitting [Fit96] or Baader and Nipkow [BN98] for general accounts), as defined by the SMT-LIB standard [BFT15]. We introduce below the syntax and semantics we rely on in our work on satisfiability checking modulo theories.

2.1 Syntax

Definition 2.1 (Many-sorted first-order language)**.** *Given disjoint countably infinite sets $\mathcal{S}$ of* sorts *(or* types*), $\mathcal{X}$ of* variables*, $\mathcal{F}$ of* function symbols*, and a function* assign $: \mathcal{X} \cup \mathcal{F} \to \mathcal{S}^+$*, a* many-sorted first-order language *$\mathscr{L}$ is a collection of these objects represented as a tuple*

$$\mathscr{L} = \langle \mathcal{S}, \mathcal{X}, \mathcal{F}, \text{assign} \rangle$$

The function assign *assigns sorts to symbols. The* sort of a variable *x in $\mathcal{X}$ is a tuple of one element from $\mathcal{S}$. The* sort of a function symbol *f in $\mathcal{F}$ is a tuple of $n+1$ elements from $\mathcal{S}$, with $n \in \mathbb{N}$ being called the* arity *of f, which is referred to as an n-ary symbol. Nullary, i.e. 0-ary, functions are called* constants*.* •

Definition 2.2 (Terms)**.** *Given a language $\mathscr{L}$ and a sort $\tau \in \mathcal{S}$, the sets of* τ-terms *are the least sets, according to the inclusion order, defined inductively as follows:*

(i) *a variable x of sort $\langle \tau \rangle$ is a τ-term;*

(ii) *if f is an n-ary function symbol of sort $\langle \tau_1, \ldots, \tau_n, \tau \rangle$ and $t_1, \ldots, t_n$ are τ_1-, …, τ_n-terms, respectively, then $f(t_1, \ldots, t_n)$ is a τ-term.*

The set of all τ-terms *is denoted as $\mathbf{T}_\tau$. Every τ-term is a* term*. The* set of all terms *is* $\mathbf{T} = \bigcup_{\tau \in \mathcal{S}} \mathbf{T}_\tau$. •

Variables of sort τ are also referred to as τ-variables. A τ-term $f(t_1, \ldots, t_n)$ is also referred to as an *application* or f-application over the *parameters* $t_1, \ldots, t_n$, with f being the *top symbol* of the application. Whenever convenient and unambiguous, we drop the τ from our denomination. A constant is a *ground term*. A term $f(t_1, \ldots, t_n)$ is ground if and only if $t_1, \ldots, t_n$ are all ground — ground terms are those without variables. The subterm relation is defined recursively as follows: a term is subterm of itself; if a term is an application, all subterms of its parameters are also its subterms.

In order do build formulas, we assume that the language contains *logical symbols*: a $\mathsf{Bool} \in \mathcal{S}$ sort; a family $(\simeq : \langle \tau, \tau, \mathsf{Bool} \rangle)_{\tau \in \mathcal{S}}$ of equality symbols, the connectives $\neg$ (negation) and $\vee$ (disjunction), and the binder $\forall$ (universal quantification). We refer to equality over Bool as *equivalence*. Nullary function symbols of sort Bool are called *propositions*.

Definition 2.3 (Formulas)**.** *Given a language $\mathscr{L}$, the* set of $\mathscr{L}$-formulas *is the least set defined inductively as follows:*

(i) *if t_1 and t_2 are τ-terms, for some sort τ, then $t_1 \simeq t_2$ is a formula;*

(ii) *if p is an n-ary function symbol of sort $\langle \tau_1, \ldots, \tau_n, \mathsf{Bool} \rangle$ and $t_1, \ldots, t_n$ are τ_1-, $\ldots$, τ_n-terms, respectively, then $p(t_1, \ldots, t_n)$ is a formula;*

(iii) *if φ is a formula, $\neg\varphi$ is a formula;*

(iv) *if φ_1 and φ_2 are formulas, then $\varphi_1 \vee \varphi_2$ is a formula;*

(v) *if x is a variable and φ is a formula, then $\forall x.\, \varphi$ is a formula.* •

Formulas are therefore terms of type Bool. A formula $\forall x.\, \varphi$ is called a *quantified formula*, with φ being its *body* and x a *bound variable*. Formulas $t_1 \simeq t_2$ and $p(t_1, \ldots, t_n)$ are *atomic formulas*, or *atoms*. A *literal* is either an atom or its negation. Given a sort τ, the *set of all τ-terms in a formula φ* is $\mathbf{T}_\tau(\varphi)$, and the *set of all terms in a formula φ* is $\mathbf{T}(\varphi) = \bigcup_{\tau \in \mathcal{S}} \mathbf{T}_\tau(\varphi)$.

The following conventions are adopted in the rest of this thesis:

$$t_1 \not\simeq t_2 \stackrel{\text{def}}{=} \neg(t_1 \simeq t_2) \qquad [\textit{Disequality}]$$

$$\varphi_1 \wedge \varphi_2 \stackrel{\text{def}}{=} \neg(\neg\varphi_1 \vee \neg\varphi_2) \qquad [\textit{Conjunction}]$$

$$\varphi_1 \to \varphi_2 \stackrel{\text{def}}{=} \neg\varphi_1 \vee \varphi_2 \qquad [\textit{Implication}]$$

$$\exists x.\, \varphi \stackrel{\text{def}}{=} \neg\forall x.\, \neg\varphi \qquad [\textit{Existential Quantification}]$$

$$\forall x_1 \ldots x_n.\, \varphi \stackrel{\text{def}}{=} \forall x_1.\, (\forall x_2.\, \ldots (\forall x_n.\, \varphi) \ldots) \qquad [\textit{Multiple universal quantification}]$$

Definition 2.4 (Free variables)**.** *Given an $\mathscr{L}$-formula φ, its* set of free variables $\mathrm{FV}(\varphi)$ *is defined*

recursively as:

$$\begin{aligned}
\mathrm{FV}(x) &= \{x\} \\
\mathrm{FV}(f(t_1, \ldots, t_n)) &= \bigcup_{i=1}^{n} \mathrm{FV}(t_i) \\
\mathrm{FV}(t_1 \simeq t_2) &= \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2) \\
\mathrm{FV}(\neg\varphi) &= \mathrm{FV}(\varphi) \\
\mathrm{FV}(\varphi_1 \vee \varphi_2) &= \mathrm{FV}(\varphi_1) \cup \mathrm{FV}(\varphi_2) \\
\mathrm{FV}(\forall x.\, \varphi) &= \mathrm{FV}(\varphi) \setminus \{x\}
\end{aligned}$$ •

A formula φ is a *sentence*, *closed formula*, if and only if $\mathrm{FV}(\varphi) = \varnothing$. A formula is *quantifier-free* if and only if it does not contain quantifiers, *ground* if and only if it is a quantifier-free sentence. The set of bound variables of a formula φ is denoted $\mathrm{BV}(\varphi)$, consisting of the least set defined inductively as the union of the sets of bound variables occurring in subformulas of φ.

Definition 2.5 (Substitution)**.** *A* substitution *is a function* $\sigma : \mathcal{X} \to \mathbf{T}$*, such that*

$$\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}, \text{ with } x_i \neq x_j, \text{ for } 1 \leq i < j \leq n,\ n \in \mathbb{N}^+,$$

maps each τ*-variable* x_i *into a* τ*-term* t_i *and every other variable not in* $x_1, \ldots, x_n$ *to itself. The* domain of σ*, written* $\mathrm{dom}(\sigma)$*, is defined as the set of variables* $\{x_1, \ldots, x_n\}$ *that* σ *is replacing by terms that are not themselves, i.e.* $\mathrm{dom}(\sigma) = \{x \mid x \in \mathcal{X} \text{ and } \sigma(x) \neq x\}$*, while the* range of σ*, written* $\mathrm{ran}(\sigma)$*, is the set of substituted terms, i.e.* $\mathrm{ran}(\sigma) = \{\sigma(x) \mid x \in \mathrm{dom}(\sigma)\}$*. A substitution* σ *is* ground *if and only if every term in* $\mathrm{ran}(\sigma)$ *is ground.*

The application of a substitution is recursively defined over terms as follows:

$$\begin{aligned}
x\sigma &= \sigma(x) \\
f(t_1, \ldots, t_n)\sigma &= f(t_1\sigma, \ldots, t_n\sigma) \\
(t_1 \simeq t_2)\sigma &= t_1\sigma \simeq t_2\sigma \\
(\neg\varphi)\sigma &= \neg(\varphi\sigma) \\
(\varphi_1 \vee \varphi_2)\sigma &= \varphi_1\sigma \vee \varphi_2\sigma \\
(\forall x.\, \varphi)\sigma &= (\forall z.\, (\varphi\{x \mapsto z\}))\sigma, \text{ with } z \text{ being a fresh variable of the appropriate sort,} \\
&\qquad \text{i.e. } z \notin \mathrm{dom}(\sigma) \cup \mathrm{ran}(\sigma) \cup \mathrm{FV}(\sigma)
\end{aligned}$$

Composition of substitutions σ *and* ρ*, denoted* $\sigma \circ \rho$*, is defined as for functions (i.e.,* ρ *is applied first). When written in postfix notation, the composition is* $\rho\sigma$*. The* fixed-point of a substitution σ *is a sequence of applications* $\sigma^\star = \sigma \ldots \sigma$ *such that* $\sigma^\star\sigma = \sigma^\star$*. A substitution* σ *is* acyclic *if and only if, for any variable* x*,* x *does not occur in* $x\sigma^\star$*. A formula* ψ_1 *is an* instance *of a formula* ψ_2 *if and only if there is a substitution* σ *such that* $\psi_1 = \psi_2\sigma$*. The substitution* σ *is referred to as an* instantiation *of* ψ_2*.* •

The application of a substitution is both respectful of *shadowing*, i.e. it does not substitute bound variables, and *capture-avoiding*: it avoids, by introducing renamings when necessary, to replace variables by terms whose free variables would be "captured" by a quantifier, i.e. become bound by it.

Definition 2.6 (Subformulas and polarity)**.** *Given an $\mathscr{L}$-formula φ, the* sets of subformulas with positive, negative, and both polarities*, denote respectively by* $\mathrm{SF}^+(\varphi)$*,* $\mathrm{SF}^-(\varphi)$*, and* $\mathrm{SF}^\pm(\varphi)$*, are the least sets such that*

(i) $\varphi \in \mathrm{SF}^+(\varphi)$*;*

(ii) *if* $\neg\psi \in \mathrm{SF}^+(\varphi)$ *(resp.* $\mathrm{SF}^-(\varphi)$*); then* $\psi \in \mathrm{SF}^-(\varphi)$ *(resp.* $\mathrm{SF}^+(\varphi)$*);*

(iii) *if* $\psi_1 \vee \psi_2 \in \mathrm{SF}^\pm(\varphi)$*, then* $\{\psi_1, \psi_2\} \subseteq \mathrm{SF}^\pm(\varphi)$*;*

(iv) *if* $\forall x.\psi \in \mathrm{SF}^\pm(\varphi)$*, then* $\psi \in \mathrm{SF}^\pm(\varphi)$*.*

The set of subformulas $\mathrm{SF}(\varphi)$ *is defined as* $\mathrm{SF}(\varphi) = \mathrm{SF}^+(\varphi) \cup \mathrm{SF}^-(\varphi)$*.* •

A subformula *occurs positively, resp. negatively, in* φ if and only if it is in $\mathrm{SF}^+(\varphi)$, resp. in $\mathrm{SF}^-(\varphi)$.

Definition 2.7 (Skolem Normal Form)**.** *An $\mathscr{L}$-formula φ is in* Skolem form *if and only if no quantified subformula $\exists x.\ \psi_1$ occurs positively and no quantified subformula $\forall x.\ \psi_2$ occurs negatively. In this case, φ is referred to as a* Skolem formula*.* •

Definition 2.8 (Conjunctive Normal Form)**.** *An $\mathscr{L}$-formula φ is in* conjunctive normal form *(CNF) if it is a conjunction of disjunctions of literals. Therefore a formula in CNF has the form $\bigwedge_{i=1}^{n} C_i$, in which each C_i has the form $\bigvee_{j=1}^{m} l_{ij}$, for literals l_{ij} and $n, m \in \mathbb{N}^+$. Each C_i is denoted a* clause*.* •

Formulas in CNF will conventionally be written as sets of clauses $\{C_1, \ldots, C_n\}$, and clauses as sets of literals $\{l_{i1}, \ldots, l_{im}\}$, such that whether the set stands for a conjunction or a disjunction of its elements is to be inferred from the context. Sets that stand for a conjunction, respectively for a disjunction, of their elements are said to be *conjunctive*, respectively *disjunctive*. Conjunctive or disjunctive sets may be written as conjunctions or disjunctions, respectively, whenever convenient and unambiguous.

We use the notations $\bar{a}_n$ and $(a_i)_{i=1}^n$ to denote the tuple, or vector, $(a_1, \ldots, a_n)$, in which $n \in \mathbb{N}^+$. We write $[n]$ for $\{1, \ldots, n\}$.

2.2 Semantics

The machinery for the assessment of truth and falsehood for terms in a language $\mathscr{L}$, and therefore meaning to the language, is presented below.

Definition 2.9 (Structure). *An $\mathscr{L}$-*structure *$\mathcal{A}$ is defined as a pair $\mathcal{A} = \langle \mathcal{D}, \mathcal{I} \rangle$, in which $\mathcal{D}$ is a* frame*, defined as a collection of non-empty* domain sets *$\mathcal{D}_\tau$, for each sort $\tau \in \mathcal{S}$, and $\mathcal{I}$ is an interpretation function that maps n-ary function symbols f of sort $\langle \tau_1, \ldots, \tau_n, \tau \rangle$ to total functions*

$$f^{\mathcal{I}} : \mathcal{D}_{\tau_1} \times \cdots \times \mathcal{D}_{\tau_n} \to \mathcal{D}_\tau$$

The elements of each domain $\mathcal{D}_\tau$ are referred to as domain elements of sort τ*. The domain of the sort* Bool *is the* set of truth values $\{\top, \bot\}$*. We say that a structure is* finite *if and only if all domain sets in its frame are finite. Respectively, a structure is* infinite *if any of its domain sets are infinite.* •

Definition 2.10 (Valuation and Interpretation). *Given an $\mathscr{L}$-structure $\mathcal{A} = \langle \mathcal{D}, \mathcal{I} \rangle$, a* valuation for $\mathcal{A}$ *is a function $\mathcal{V} : \mathcal{X} \to \mathcal{D}$, such that each τ-variable $x \in \mathcal{X}$ is mapped to an element in $\mathcal{D}_\tau$.*

*An $\mathscr{L}$-*interpretation *$\mathcal{M}$ comprises a structure $\mathcal{A}$ and a valuation $\mathcal{V}$, such that $\mathcal{M} = \langle \mathcal{A}, \mathcal{V} \rangle$. We say that $\mathcal{M}$ is a* finite interpretation *if and only if $\mathcal{A}$ is a finite structure. Respectively, M is an* infinite interpretation *if and only if $\mathcal{A}$ is an infinite structure.* •

Consider valuations $\mathcal{V}$ and $\mathcal{V}'$ and a set of variables Z. $\mathcal{V}'$ *agrees with* $\mathcal{V}$ *except for* Z if and only if, for every variable $x \notin Z$, $\mathcal{V}'(x) = \mathcal{V}(x)$. $\mathcal{V}'$ and $\mathcal{V}$ *agree on* Z if and only if $\mathcal{V}'(y) = \mathcal{V}(y)$, for every variable $y \in Z$.

Definition 2.11 (Extending interpretations). *Given two languages $\mathscr{L}$ and $\mathscr{L}'$, and an $\mathscr{L}$-interpretation $\mathcal{M} = \langle \mathcal{A}, \mathcal{V} \rangle$, an $\mathscr{L}'$-interpretation $\mathcal{M}' = \langle \mathcal{A}', \mathcal{V}' \rangle$* extends *$\mathcal{M}$ if and only if*

(i) *$\mathscr{L}' \supseteq \mathscr{L}$;*

(ii) *every domain set in $\mathcal{A}$ is a subset of the respective domain set in $\mathcal{A}'$;*

(iii) *for every function symbol f interpreted in $\mathcal{M}$, $f^{\mathcal{I}'}$ in $\mathcal{M}'$ extends $f^{\mathcal{I}}$, i.e. $f^{\mathcal{I}'}$ coincides with $f^{\mathcal{I}}$ on the domain sets of $\mathcal{A}$*

An $\mathscr{L}'$-interpretation $\mathcal{M}'$ extends *a class of $\mathscr{L}$-interpretations Ω if and only if $\mathcal{M}'$ extends each interpretation $\mathcal{M} \in \Omega$.* •

Let $\mathcal{M} = \langle \mathcal{A}, \mathcal{V} \rangle$ be an $\mathscr{L}$-interpretation. The notation $\mathcal{M}_{\bar{x}_n \mapsto \bar{d}_n}$ denotes an $\mathscr{L}$-interpretation with the same structure as $\mathcal{M}$ and whose valuation $\mathcal{V}_{\bar{x}_n \mapsto \bar{d}_n}$ agrees with $\mathcal{V}$ except for $x_1, \ldots, x_n$, besides mapping each τ-variable x_i into the element $d_i \in \mathcal{D}_\tau$.

Definition 2.12 (Evaluation in an interpretation). *Given an $\mathscr{L}$-interpretation $\mathcal{M} = \langle \mathcal{A}, \mathcal{V} \rangle$ and*

an $\mathscr{L}$-formula ψ, the evaluation of ψ in $\mathcal{M}$*, noted $[\![\psi]\!]^{\mathcal{M}}$, is defined recursively as follows:*

$$
\begin{aligned}
&[\![x]\!]^{\mathcal{M}} && = \mathcal{V}(x)\\
&[\![f(t_1, \ldots, t_n)]\!]^{\mathcal{M}} && = f^{\mathcal{I}}([\![t_1]\!]^{\mathcal{M}}, \ldots, [\![t_n]\!]^{\mathcal{M}})\\
&[\![t_1 \simeq t_2]\!]^{\mathcal{M}} && = \top \text{ if and only if } [\![t_1]\!]^{\mathcal{M}} = [\![t_2]\!]^{\mathcal{M}};\\
& && \quad \bot \text{ otherwise}\\
&[\![\neg\varphi]\!]^{\mathcal{M}} && = \top \text{ if and only if } [\![\varphi]\!]^{\mathcal{M}} = \bot;\\
& && \quad \bot \text{ otherwise}\\
&[\![\varphi_1 \vee \varphi_2]\!]^{\mathcal{M}} && = \top \text{ if and only if } [\![\varphi_1]\!]^{\mathcal{M}} = \top \text{ or } [\![\varphi_2]\!]^{\mathcal{M}} = \top;\\
& && \quad \bot \text{ otherwise}\\
&[\![\forall x.\, \varphi]\!]^{\mathcal{M}} && = \top \text{ if and only if } [\![\varphi]\!]^{\mathcal{M}_{x\mapsto d}} = \top, \text{ for every } d \in \mathcal{D}_\tau, \ x \text{ being a } \tau\text{-term};\\
& && \quad \bot \text{ otherwise.}
\end{aligned}
$$

A formula ψ is true in $\mathcal{M}$ *if and only if $[\![\psi]\!]^{\mathcal{M}} = \top$ and* false in $\mathcal{M}$ *if and only if $[\![\psi]\!]^{\mathcal{M}} = \bot$.* •

2.3 Satisfiability

Definition 2.13 (Satisfiability)**.** *Given an $\mathscr{L}$-interpretation $\mathcal{M}$ and an $\mathscr{L}$-formula ψ, $\mathcal{M}$* satisfies *ψ, written $\mathcal{M} \models \psi$, if and only if it makes it* true*, i.e. $[\![\psi]\!]^{\mathcal{M}} = \top$. A formula φ is* satisfiable *if and only if there is an interpretation $\mathcal{M}$ such that $\mathcal{M} \models \varphi$, in which case we say that $\mathcal{M}$ is a* model *of φ; otherwise φ is* unsatisfiable*, written $\models \neg\varphi$. Given two $\mathscr{L}$-formulas φ and ψ, φ* entails *ψ, written $\varphi \models \psi$, if and only if every model of φ is also a model of ψ. An $\mathscr{L}$-formula φ is* valid*, written $\models \varphi$, if and only if it is true in every $\mathscr{L}$-interpretation $\mathcal{M}$.* •

In this thesis we are mainly concerned with a specific kind of interpretation: as defined in Baumgartner and Waldmann [BW13], an $\mathscr{L}$-interpretation $\mathcal{M}$ is *term-generated* if and only if every element in a domain set $\mathcal{D}_\tau$ is the interpretation of some ground τ-term. These are the type of interpretations we are going to build when automatically assessing satisfiability of formulas. Due to the Herbrand theorem, we know that checking satisfiability of closed Skolem formulas in many-sorted first-order logic with equality can be reduced to checking satisfiability of ground instances. Therefore it suffices to look for term-generated models when trying to assess satisfiability.

Next we define the more general problem that our work addresses: that of determining satisfiability when considering theories.

Definition 2.14 (Theory)**.** *A* theory *$\mathcal{T}$ is a set of $\mathscr{L}$-sentences* closed under entailment*: for every $\mathscr{L}$-formula φ, if $\mathcal{T} \models \varphi$ then $\varphi \in \mathcal{T}$.* •

To work with the above definition in terms of interpretations, we present the following definitions. Given an $\mathscr{L}$-interpretation $\mathcal{M}$, the *theory of* $\mathcal{M}$, denoted $\mathrm{Th}(\mathcal{M})$, is the set of all $\mathscr{L}$-sentences satisfied by $\mathcal{M}$. This notion is generalized to a class of $\mathscr{L}$-interpretations Ω, such

that the *theory of* Ω, denoted $\mathrm{Th}(\Omega)$, is the set of all $\mathscr{L}$-sentences simultaneously satisfied by every $\mathcal{M} \in \Omega$.

Definition 2.15 (Satisfiability modulo theory)**.** *Consider a theory* $\mathcal{T} = \mathrm{Th}(\Omega)$*, in which* Ω *is a class of* $\mathscr{L}$*-interpretations, and a language* $\mathscr{L}'$ *such that* $\mathscr{L}' \supseteq \mathscr{L}$*. We say that an* $\mathscr{L}'$*-formula* φ *is* $\mathcal{T}$-satisfiable *if and only if there is an interpretation* $\mathcal{M}'$ *which extends* Ω *and is a model for* φ*. Similarly, given* $\mathscr{L}'$*-formulas* φ *and* ψ*, we say that* φ $\mathcal{T}$-entails ψ*, written* $\varphi \models_{\mathcal{T}} \psi$*, if and only if every* $\mathscr{L}'$*-model of* φ *extending* Ω *is also a model of* ψ*.* •

In general not only one but a combination, i.e. a union, of theories is considered when checking satisfiability. We follow Barrett et al. [BSST09] for establishing satisfiability modulo a combination of theories defined through classes of interpretations. We assume that if the theories were defined axiomatically, as in Definition 2.14, their combination could be made simply by the union of their axioms (if the languages were disjoint, otherwise through a renaming of common symbols that takes into account whether shared function symbols are meant to stand for the same function).

Definition 2.16 (Satisfiability modulo a combination of theories)**.** *An* $\mathscr{L}$*-interpretation* $\mathcal{M}$ *is a* $\mathscr{L}$-reduct *of an* $\mathscr{L}'$*-interpretation* $\mathcal{M}'$*, with* $\mathscr{L}' \supseteq \mathscr{L}$*, if and only if* $\mathcal{M}$ *and* $\mathcal{M}'$ *have the same frame and the interpretation function of* $\mathcal{M}$ *coincides with that of* $\mathcal{M}'$ *on the symbols of* $\mathscr{L}$*.*

Consider theories $\mathcal{T}_1 = \mathrm{Th}(\Omega_1)$ *and* $\mathcal{T}_2 = \mathrm{Th}(\Omega_2)$*, in which* Ω_1 *is a class of* $\mathscr{L}_1$*-interpretations and* Ω_2 *a class of* $\mathscr{L}_2$*-interpretations, with* $\mathscr{L}_1$ *and* $\mathscr{L}_2$ *being disjoint. Assume that both* Ω_1 *and* Ω_2 *are closed under isomorphisms. The combination* $\mathcal{T}_1 \oplus \mathcal{T}_2$ *is defined by the class* Ω_{1+2} *of* $(\mathscr{L}_1 \cup \mathscr{L}_2)$*-interpretations such that, for every interpretation in* Ω_{1+2}*, its* $\mathscr{L}_1$*-reduct is an interpretation in* Ω_1 *and its* $\mathscr{L}_2$*-reduct is an interpretation in* Ω_2*. This way, determining satisfiability modulo* $\mathcal{T} = \mathcal{T}_1 \oplus \mathcal{T}_2$ *to an* $\mathscr{L}'$*-formula* φ*, with* $\mathscr{L}' \supseteq (\mathscr{L}_1 \cup \mathscr{L}_2)$*, amounts, as before, to finding an* $\mathscr{L}'$*-model* $\mathcal{M}$ *for* φ *which extends* Ω_{1+2}*.* •

Chapter 3

Satisfiability modulo theories solving

We have formally defined the problem of satisfiability modulo background theories in many-sorted first-order logic with equality. We now proceed to the main topic of this thesis: how to automatically determine the satisfiability of a first-order formula while taking into account a set of theories. This is commonly known as the SMT problem. Its relevance comes from the considerable gain in expressivity achieved by combining first-order logic with theories. Reasoning efficiently in such a setting is of utmost importance to a variety of formal methods applications that can rely on logic to encode problems. This allows computer-aided reasoning to be applied on e.g. formal verification, program synthesis, automatic testing, program analysis, and so on.

The focus on real-world problems imposes a restriction on the relevant solutions to a given SMT problem. One generally is interested only in solutions which are extensions of the standard models of the background theories. So in practice, when solving e.g. a satisfiability problem modulo linear integer arithmetic, the only desired solutions are those that interpret the symbols $+$, $-$, $<$, and so on, in the usual way over the integer domain. One benefit of fixing interpretations for theories, rather than relying e.g. on axiomatizations, is to be able to use ad-hoc (decision) procedures for these specific structures, instead of relying on general purpose methods. For combinations of quantifier-free theories this often leads to highly efficient decision procedures.

The importance of the SMT problem led to the field of research commonly know as "SMT solving" (see [Seb07] or [BSST09] for comprehensive overviews). It originated with the simpler problem of Boolean satisfiability, i.e. determining if a propositional formula has a model. This problem is commonly known as SAT, and is one of the first to be shown NP-complete. As such it was an intractable problem for many years, until the so called SAT revolution in the turn of the last century: modern SAT solvers, implementing the conflict-driven clause-learning (CDCL) calculus, can efficiently solve very large problems coming from real world problems containing hundreds of thousands of propositional variables and millions of clauses. SMT solving developed initially in the 1970s and 1980s by pioneers such as Nelson and Oppen [NO79, NO80] and Shostak [Sho84]. The focus has been generally on quantifier-free logics, which are often decidable. Modern approaches started in the late 1990s, harnessing the power of SAT solvers to build more scalable systems. These approaches can be classified in two broad classes. In the first,

an SMT problem is fully encoded into a SAT instance and fed into a SAT solver, without further intervention. Theory reasoning is performed only in order to provide a better encoding into SAT. In the second, the theory reasoning is performed by a combination of decision procedures that are interleaved with a SAT solver. The SMT problem is abstracted as a SAT instance, whose solution is only a candidate one for the original problem: a propositionally consistent conjunction of literals, which is then checked for consistency modulo the respective theories. The CDCL($\mathcal{T}$) framework [NOT06] is such an approach for SMT solving. It is the most common one adopted for systems performing automatic SMT solving. Our focus is in this framework and, more importantly, on how to reason efficiently with quantified formulas on it.

In classical logics, the dual problem of satisfiability is that of theorem proving: showing that a formula is a theorem is equivalent to showing that its negation is unsatisfiable, while finding a model for its negation provides a counter-example to the conjecture. The field of automatic theorem proving (ATP) historically was focused on pure and equational first-order logic, where the biggest challenge is how to efficiently handle quantified formulas. Theory reasoning in the presence of quantifiers poses steep challenges, as it is often incomplete and hard to be performed efficiently even in an heuristic manner. Nevertheless, numerous works have tried to extend ATP systems with theory reasoning, to the point that the differences between SMT solvers and ATP systems are becoming increasingly smaller.

3.1 CDCL($\mathcal{T}$) framework

As previously described, what is commonly known as an SMT solver is a system implementing the CDCL($\mathcal{T}$) calculus, by Nieuwenhuis et al. [NOT06]. Examples of such systems are the widely used CVC4 [BCD+11] and Z3 [dMB08b] solvers, as well as Yices [Dut14] and veriT [BdODF09].

CDCL($\mathcal{T}$) is an extension to the conflict-driven clause learning (CDCL) calculus that accommodates theory reasoning. CDCL is a refinement of the classic DPLL [DLL62] procedure that combines a backtracking search for a propositional model with a powerful conflict analysis engine. The search is performed by interleaving "decisions", i.e. heuristic assignments of literals, with the application of unit propagation until a fixpoint. The explicit backtracking is performed by means of a *learned clause* derived when a conflict is reached in the search.

Considering a theory $\mathcal{T}$, which is generally a combination of theories $\mathcal{T}_1, \ldots, \mathcal{T}_n$, such as the theory of equality and uninterpreted functions[1] and linear arithmetic, CDCL($\mathcal{T}$) determines the satisfiability of a first-order formula φ, in conjunctive normal form (CNF), modulo the theory $\mathcal{T}$. Here we assume that φ is quantifier-free. How to handle quantified formulas in a CDCL($\mathcal{T}$) architecture is discussed in the next section. The reasoning modulo each theory $\mathcal{T}_i$ is performed by a specific *theory solver*, which is generally a decision procedure for the $\mathcal{T}_i$-

[1] This theory coincides with many-sorted first-order logic with equality. In an SMT context the term "uninterpreted" refers to function symbols not interpreted by the background theories. Since in many-sorted equational first-order logic only the connectives and the equality symbol have fixed interpretations, all other function symbols are considered "uninterpreted".

satisfiability of a conjunction of literals in the language of $\mathcal{T}_i$. Even though theory solvers cannot be combined modularly in general — there are theories whose satisfiability problem is decidable in isolation but not when combined with other theories — some restricted scenarios allow successful combinations. The Nelson-Oppen framework [NO79] is a common choice for combining theory solvers into a decision procedure for the overall combination. It does not impose too many formal restrictions on the considered theories and can lead to efficient implementations. With such a framework, from models obtained by each theory solver a model for the combination of the theories can be built, if it exists. The procedure in Figure 3.1 summarises how the calculus is applied.

```
function CheckSat(φ, T) is
  φ ← Process(φ)                          // Simplifications, CNF transformation
  do
    E ← CheckBoolean(abs(φ))                                        // SAT solver
    if E = ∅ then
      return Unsat
    C ← CheckGround(E, T)                                          // Theory solvers
    φ ← φ ∪ C
  while C ≠ ∅
  return Sat
```

Figure 3.1: An abstract procedure for checking the satisfiability of quantifier-free formulas based on the CDCL($\mathcal{T}$) framework.

The Process routine normalises the input formula into CNF and performs transformations aiming to optimise the solving. Examples of such transformations are arithmetic simplifications and applying symmetry breaking [DFMP11]. The result of Process must be equisatisfiable to the input formula, i.e. if the original formula had a model extending the intended model of $\mathcal{T}$, so will the processed formula, and vice-versa. Generally, a SAT solver implementing CDCL embodies CheckBoolean. Its input is the propositional abstraction of the formula, obtained with the function *abs*. It maps every distinct atom into a distinct proposition. If the formula is propositionally satisfiable, the SAT solver outputs a conjunction of propositions, commonly known as an *assignment*, representing a valuation that makes $abs(\varphi)$ true: the found assignment entails the propositional abstraction of the input formula. The set of literals whose abstractions appear in the assignment is represented by E. If E is empty, the formula is propositionally unsatisfiable, therefore it is also $\mathcal{T}$-unsatisfiable. Since by construction the satisfiability of E entails the satisfiability of φ, every model of E is also a model for φ. From now on we will ambiguously refer to E as a *candidate model* for the original formula, due to the entailment relationship between them.

The CheckGround module is generally implemented as a combination of specialised solvers for each theory in a Nelson-Oppen framework. Thus it amounts to a ground decision procedure for checking the $\mathcal{T}$-satisfiability of E. If E is $\mathcal{T}$-unsatisfiable, a *conflict clause* is produced, embodying the explanation for the unsatisfiability of E. This clause is then learned, i.e. it is

added to the original problem in order to prevent the SAT solver from deriving again this faulty assignment. If no conflict clause is derived, then E is $\mathcal{T}$-satisfiable, and so is the original formula.

Example 3.1. Let φ be the formula

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

We evaluate the satisfiability of φ modulo the combination $\mathcal{T}$ of the quantifier-free theories of equality and uninterpreted functions and linear real arithmetic. This combination is decidable. The processing of φ yields the CNF

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

whose propositional abstraction is

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

A satisfying assignment for $abs(\varphi)$ produced by CheckBoolean may be

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

which then triggers CheckGround to determine the $\mathcal{T}$-satisfiability of the conjunctive set of literals $E = \{a \leq b,\, b \leq a + x,\, x \simeq 0,\, f(a) \not\simeq f(b)\}$. Generally, this would be done jointly by a *congruence closure* procedure [NO80, BTV03] for handling equality and uninterpreted symbols together with a simplex algorithm [DdM06] for handling arithmetic, both combined in a Nelson-Oppen framework. The arithmetic solver will determine that a and b are equal, and provide the equality $a \simeq b$ to the congruence closure algorithm. By congruence, $f(a) \simeq f(b)$ is derived, which is in conflict with the literal $f(a) \not\simeq f(b)$ from E. Therefore E is $\mathcal{T}$-unsatisfiable and the conflict clause

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

is derived and added to φ. After another iteration of the loop, a candidate model for the new formula is

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

which is also unsatisfiable in this combination of theories. By adding the conflict clause

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

to the new formula the SAT solver will be unable to produce a satisfying assignment. Therefore the procedure halts and answers Unsat, showing that the original formula φ is $\mathcal{T}$-unsatisfiable.

The key element in the quantifier-free CDCL($\mathcal{T}$) architecture are the theory solvers. They are generally *ad hoc*, with specialised (decision) procedures tailored to the theory in question.

An important element in the use of these solvers is whether they are invoked *offline* or *online*[2]. Invocations are done online when they occur at every decision point in the SAT solver search: after a decision is made and the SAT solver propagates until saturation, the theory solver is invoked to analyze the current assignment. An offline invocation occurs only when the SAT solver has provided a model for the propositional abstraction of the formula, i.e. a full assignment. For simplicity, in Figure 3.1 it is assumed that theory solvers are called offline. In practice, however, they are generally employed online. This allows for propositional assignments to be discarded as soon as they become $\mathcal{T}$-unsatisfiable, thus avoiding the consideration of all full assignments extending the refuted one.

Another point for theory solvers is that they generally enjoy some important features (as described in [Seb07] and [BSST09]) to make the most of the CDCL($\mathcal{T}$) framework:

- *Model generation*: when the procedure witnesses the consistency of the set of literals in its theory, it is capable of producing a model and communicating it to the other components of the framework.

- *Conflict set generation*: when the procedure attests the inconsistency of the set of literals in its theory, it produces a (possibly minimal) subset from the given set of literals which explains its unsatisfiability.

- *Incrementality and backtrackability*: the theory solver is stateful between different calls from the outer loop. When checking an assignment, only literals not previously asserted are evaluated. This can lead to significant performance gains, considering that theory solvers may be called numerous times while the SAT solver derives candidate models. This is specially the case when an online approach is implemented, with calls occurring at each partial propositional assignment.

- *Propagation*: the solver is capable of propagating useful information (such as asserting literals previously undefined in the SAT solver assignment) from the current candidate model. This often leads to significant reductions of the potential search space by deriving useful constraints.

3.2 Quantified formulas in CDCL($\mathcal{T}$)

Reasoning with quantifiers modulo background theories is often costly and not decidable in general. Historically, in the CDCL($\mathcal{T}$) framework mainly heuristic instantiation techniques have been used to tackle this problem. They allow to quickly derive ground instances from quantifiers and turn the reasoning back to the efficient ground solver. Here we describe the main instantiation techniques applied in CDCL($\mathcal{T}$).

[2]These are sometimes also referred to as *eager* and *lazy* approaches.

```
function CheckSatQ(φ, T) is
  φ ← Process(φ)                                  // Simplifications, CNF transformation
  do
    ⟨E, Q⟩ ← CheckBoolean(absQ(φ))                // SAT solver
    if E ∪ Q = ∅ then
      return Unsat
    C ← CheckGround(E, T)                         // Theory solvers
    if C ≠ ∅ then
      φ ← φ ∪ C
      continue
    I ← Inst(E, Q, T)                             // Instantiation module
    φ ← φ ∪ I
  while I ≠ ∅
  if models can be built for T then
    return Sat
  else
    return Unknown
```

Figure 3.2: An abstract procedure for checking satisfiability based on the CDCL($\mathcal{T}$) framework.

The abstract procedure CheckSat in Figure 3.1 is extended in Figure 3.2 to accommodate quantifiers. Process works as before, except that the formula is also put in Skolem form, with all quantified subformulas being quantified clauses[3]. The propositional abstraction produced by the function *absQ* coincides with *abs* for literals outside the scope of quantifiers but furthermore abstracts quantified formulas as propositions. The candidate model produced by CheckBoolean is separated into a set E of ground literals and $\mathcal{Q}$ of quantified formulas. E is from now on referred to as the *ground model*. When E is $\mathcal{T}$-satisfiable, the reasoning proceeds for evaluating $E \cup \mathcal{Q}$ as a whole, taking the quantifiers into account. Instances are derived by the function Inst, which embodies a quantifier module. The satisfiability check is subsequently performed in the processed formula augmented with the generated instances.

The set $\mathcal{I}$ is composed of *instantiation lemmas*

$$\forall \bar{x}_n.\, \psi \rightarrow \psi\sigma$$

from the quantified formulas $\forall \bar{x}_n.\, \psi$ in $\mathcal{Q}$, in which $\psi\sigma$ is a ground instance. When Inst does not lead to an overall refutational complete procedure, there is no guarantee that if the original formula is unsatisfiable then ground $\mathcal{T}$-unsatisfiability will be derived through instantiation. Similarly, when no instances are generated, the overall procedure can only answer Sat if the instantiation module is such that models can be produced when no more instances can be derived. If these conditions are not met, CheckSatQ may loop indefinitely or answer Unknown. Below we overview in detail the main general purpose techniques for implementing an instantiation module.

[3]This is not a requirement for handling quantified formulas in CDCL($\mathcal{T}$), but we adopt it for simplicity, without loss of generality.

3.2.1 Trigger based instantiation

The most common instantiation technique in SMT solving is based on the following observation [Nel80]: while a quantified formula $\forall \bar{x}_n.\, \psi$ is semantically equivalent to the infinite conjunction $\bigwedge_\sigma \psi\sigma$, in which σ ranges over all substitutions, only part of these instances are "relevant" to prove that the overall formula is unsatisfiable. Moreover, considering as relevant instances those containing terms appearing in the ground model is an effective trade off between completeness and efficiency.

This set of relevant instances is approximated through a selection of terms occurring in the quantified formula. Instantiations are derived by solving the E-matching of these selected terms with terms occurring in the ground model. Solutions are substitutions that make two terms equal modulo the equalities in E. The resulting instantiations yield ground formulas. These sets of selected terms are called *triggers*.

Definition 3.1 (E-matching)**.** *Given a conjunctive set of equality literals E and terms u and t, with t ground, the* E-matching problem *is that of finding substitutions σ such that $E \models u\sigma \simeq t$ holds.*

Definition 3.2 (Triggers)**.** *A* trigger *T for a quantified formula $\forall \bar{x}_n.\, \psi$ is a set of non-ground terms $u_1, \ldots, u_m \in \mathbf{T}(\psi)$ such that $\{\bar{x}_n\} \subseteq \mathrm{FV}(u_1) \cup \cdots \cup \mathrm{FV}(u_m)$.*

Given a model $E \cup \mathcal{Q}$, trigger based instantiation considers each quantified formula $\forall \bar{x}_n.\, \psi$ in $\mathcal{Q}$ independently. It proceeds by

1. selecting triggers $T_1, \ldots, T_m$ from the terms in ψ
2. deriving the set of instantiation lemmas

$$\mathcal{I} = \bigcup_{i=1}^{m} \left\{ \forall \bar{x}_n.\, \psi \to \psi\sigma \;\middle|\; \begin{array}{l} T_i = \{u_1, \ldots, u_{m_i}\},\ E \models u_1\sigma \simeq t_1, \ldots, E \models u_{m_i}\sigma \simeq t_{m_i}, \\ \text{in which } t_1, \ldots, t_{m_i} \text{ are ground terms from } E \end{array} \right\}$$

3. adding the lemmas in $\mathcal{I}$ to the original problem

Instantiations are determined by computing substitutions for the variables in the selected terms. These substitutions are the solutions for the simultaneous E-matching problem of the selected terms and chosen ground terms occurring in E. Solutions are substitutions for which E groundly entails that each selected term is equal, modulo the theory of equality, to the respective ground term.

An abstract version of the algorithm to compute such substitutions, as presented by de Moura and Bjørner [dMB07], is

$$\begin{array}{lcl} \mathit{ematch}(x,\, t,\, S) & = & \{\sigma \cup \{x \mapsto t\} \mid \sigma \in S,\, x \notin \mathit{dom}(\sigma)\} \cup \{\sigma \mid \sigma \in S,\, E \models x\sigma \simeq t\} \\ \mathit{ematch}(t',\, t,\, S) & = & \begin{cases} S & \text{if } E \models t' \simeq t \\ \varnothing & \text{otherwise} \end{cases} \\ \mathit{ematch}(f(\bar{s}_n),\, t,\, S) & = & \displaystyle\bigcup_{f(\bar{t}_n) \in \mathbf{T}(E),\, E \models f(\bar{t}_n) \simeq t} \mathit{ematch}(s_n,\, t_n,\, \ldots,\, \mathit{ematch}(s_1,\, t_1,\, S)\ldots) \end{array}$$

in which x is a variable, t is a ground term, s is a term, and S is a set of substitutions. All instantiations for a quantified formula with a trigger $T = \{u_1, \ldots, u_n\}$ can be obtained with the set of substitutions

$$S = \bigcup_{t_1, \ldots, t_n \in \mathbf{T}(E)} \mathit{ematch}(u_n, t_n, \ldots, \mathit{ematch}(u_1, t_1, \{\sigma\}) \ldots)$$

in which σ is a substitution mapping the variables in T to themselves.

Even though E-matching is an NP-complete problem, it can be performed efficiently in the context of a CDCL($\mathcal{T}$) solver (see e.g. [dMB07] describing an implementation in the Z3 solver). Implementations exploit the E-graph that is generally maintained by the ground solver. This data structure is a concretization of the separation of the set of terms $\mathbf{T}(E)$ into congruence classes according to the congruence relation $\simeq$.

Example 3.2. Let E and $\mathcal{Q}$ be conjunctive sets, with $E = \{f(a) \simeq g(b),\ h(a) \simeq b,\ f(a) \simeq f(c)\}$ and $\mathcal{Q} = \{\forall x.\ f(x) \not\simeq g(h(x))\}$. Possible triggers for $\forall x.\ f(x) \not\simeq g(h(x))$ are $T_1 = \{f(x)\}$, $T_2 = \{h(x)\}$, $T_3 = \{f(x), g(h(x))\}$ and so on. If only T_1 is chosen, the possible instantiations for the quantified formula are derived by E-matching $f(x)$ with terms containing f-applications in their congruence classes. The only such congruence class over $\mathbf{T}(E)$ is $\{f(a), f(c), g(b)\}$. Therefore the resulting substitutions are obtained from $\mathit{ematch}(f(x), g(b), \{\{x \mapsto x\}\})$. Since $g(b)$ is congruent to the terms $f(a)$ and $f(c)$, this function is reduced to

$$\mathit{ematch}(x, a, \{\{x \mapsto x\}\}) \cup \mathit{ematch}(x, c, \{\{x \mapsto x\}\})$$

which yields the set of substitutions with $\sigma_1 = \{x \mapsto a\}$ and $\sigma_2 = \{x \mapsto c\}$. These substitutions lead to the instantiation lemmas

$$\forall x.\ f(x) \not\simeq g(h(x)) \rightarrow f(a) \not\simeq g(h(a))$$

and

$$\forall x.\ f(x) \not\simeq g(h(x)) \rightarrow f(c) \not\simeq g(h(c))$$

which are then added to the original problem.

Trigger selection The effectiveness of trigger based instantiation is highly dependent on which terms are selected to compose triggers. Therefore good selection strategies are paramount for efficient CDCL($\mathcal{T}$) solvers. Alternatively, users may annotate quantified formulas with triggers, thus capturing their intuition of which instantiations are more important to solve a problem. The selection strategy also influences the completeness of the solver. Dross et al. [DCKP13] and Bansal et al. [BRK+15] have presented dedicated trigger selection strategies to obtain completeness for specific fragments of first-order logic, while Rümmer [Rüm12] presents a calculus, not in CDCL($\mathcal{T}$) framework, whose completeness results are independent of the trigger selection strategy.

There is no standard trigger selection strategy, but recent work by Leino and Pit-Claudel [LP16] combines typical approaches and introduces some sophisticated criteria. A possible strategy to be extracted from their work is as follows:

1. Traverse a quantified formula and annotate terms as trigger heads or trigger killers. *Trigger heads* are terms with at least one of the quantified variables as subterm, and that do not include trigger killers. *Trigger killers* are terms not permitted to appear in triggers, typically applications of interpreted symbols, e.g. arithmetic operations or logical connectives.

2. Candidate triggers are built by combining all possible trigger heads while ensuring two properties: adequacy and parsimony. A candidate is *adequate* if it contains all the quantified variables and *parsimonious* if removing any term from the candidate makes it inadequate. E.g. a quantified formula $\forall x.\, p_1(x) \vee \cdots \vee p_n(x)$ has the trigger heads $\{p_1(x), \ldots, p_n(x)\}$ and 2^n adequate candidates, but only n which are parsimonious: the singletons $\{p_1(x)\}$, $\ldots$, $\{p_n(x)\}$.

3. The remaining candidates are ordered by specificity. Let T_1, T_2 be candidate triggers. T_1 is less *specific* than T_2 if and only if all matchings of T_2 are also matchings for T_1. This is the case when for each trigger head t in T_1 there is a trigger head t' in T_2 such that t matches t', i.e. there is a substitution σ such that $t\sigma = t'$.

4. Finally, the possible triggers for the quantified formula are the minimal candidates.

Example 3.3. Consider again the quantified formula $\forall x.\, f(x) \not\simeq g(h(x))$. The possible triggers derived with the above strategy would be only $T_1 = \{f(x)\}$ and $T_2 = \{h(x)\}$, since any trigger with $g(h(x))$ will be less specific than T_2 and any non-singleton trigger will not be parsimonious.

Avoiding matching loops A *matching loop* occurs when quantifiers generate instances that only serve to re-instantiate them, thus locking the solver in an infinite chain of instantiations.

Example 3.4 ([Rey16])**.** Consider the formula $\forall x.\, f(f(x)) \simeq f(x)$ with the trigger $T = \{f(x)\}$. An instantiation $\{x \mapsto t\}$ will add the term $f(f(t))$ to the problem, which will subsequently lead to a possible new instantiation $\{x \mapsto f(t)\}$ for this trigger, which would generate a term $f(f(f(t)))$ and so on.

Up to now solutions have been proposed only to the cases in which loops depend on a single quantifier, i.e. when the instances of a quantifier directly trigger new instantiations of the same formula. Leino and Pit-Claudel [LP16] propose selection strategies to avoid matching loops. In the above example the trigger $\{f(x)\}$ would be discarded because it is a subterm of a term in the same formula, namely $f(f(x))$. The only possible trigger for this formula would be $\{f(f(x))\}$, which would not lead to looping instantiations on this quantifier. Other works rely not on trigger selection strategies, but on the handling of produced instances. Ge et al. [GBT07]

consider instantiations in a breadth-first manner by associating levels to instantiated terms, while de Moura and Bjørner [dMB07] and Barbosa [Bar16] discard instances based on their activity, measured according to the participation of instances in conflicts in the SAT solver. All these strategies have shortcomings, given the trade-off between avoiding matching loops and missing relevant instances.

Avoiding explosion The main issue of trigger based instantiation is the large number of produced instances, which often allow only a few rounds of instantiation before the ground solver is unable to continue the solving.

Example 3.5. Consider the quantified formula $\forall xyz.\, f(x) \simeq f(z) \to g(y) \simeq h(z)$ and the trigger $T = \{f(x),\, g(y),\, h(z)\}$. If E contains 100 applications of each function symbol f, g, and h, a million instances will be generated from this trigger for this formula.

Reynolds [Rey16] applies a strategy in the CVC4 solver that limits the number of possible instantiations for each trigger, with no more than one term per congruence class being used to generate instances. While their trigger based instantiation is very effective, the problem of generating too many instances remains. Leino et al. [LMO05] and later Moskal and Łopuszański [MŁ06] have used a *two-tier approach*: the instances, together with the current ground model E, are fed into a secondary SMT solver; after unsatisfiability is (hopefully) attained for this combination, only the instances that were relevant for deriving the conflict are added to the original problem. While this technique is promising, it has not been implemented in state-of-the-art CDCL($\mathcal{T}$) solvers. The approaches for discarding instances in [dMB07] and [Bar16] also aim to minimise the number of instances to consider, but their effectiveness is not yet clear.

3.2.2 Conflict based instantiation

The lack of a *goal* in trigger based instantiation (such as e.g. refuting the candidate model) leads to the production of many instances irrelevant for the solving. Furthermore, unlike other non-goal-oriented techniques, such as the techniques based on the superposition calculus, there are no straightforward redundancy criteria for the elimination of derived instances in the CDCL($\mathcal{T}$) framework. Therefore useless instances are generally kept, potentially hindering the solver's performance. Relying on heuristic instantiation also leads to the so called *butterfly effect*: small changes on the structure of the input problem have unpredictable effects on the solver's outcome. This is particularly harmful since CDCL($\mathcal{T}$) solvers are commonly used as backends in a myriad of tools, e.g. in formal verification, which cannot afford to suffer from such instability. These issues started to be addressed by Reynolds et al. [RTdM14] when introducing conflict based instantiation in the context of a CDCL($\mathcal{T}$) solver. It is a goal-oriented instantiation technique: only instances which are conflicting with the current ground model are derived. For efficiency reasons, quantified formulas are evaluated independently in the search for conflicting instances.

Given a candidate model $E \cup \mathcal{Q}$, an instance ψ is said to be *conflicting with* E if $E \wedge \psi \models_{\mathcal{T}} \bot$, or, alternatively, if $E \models_{\mathcal{T}} \neg\psi$. Therefore, given some quantified formula $\forall \bar{x}_n.\ \psi$ in $\mathcal{Q}$, finding conflicting instances amounts to searching for a ground substitution σ such that $E \models_{\mathcal{T}} \neg\psi\sigma$. Adding $\forall \bar{x}_n.\ \psi \rightarrow \psi\sigma$ to the original problem will prevent the derivation of that same candidate model. This way all instances produced by conflict based instantiation will a priori be relevant for the solving, witnessing a faulty assignment produced by the SAT solver and preventing its repetition, the same way that learned conflict clauses at the ground and propositional level do.

Example 3.6. Consider again the conjunctive sets $E = \{f(a) \simeq g(b),\ h(a) \simeq b,\ f(a) \simeq f(c)\}$ and $\mathcal{Q} = \{\forall x.\ f(x) \not\simeq g(h(x))\}$ from Example 4.2. While the trigger $\{f(x)\}$ led to the two instantiation lemmas

$$\forall x.\ f(x) \not\simeq g(h(x)) \rightarrow f(a) \not\simeq g(h(a))$$

and

$$\forall x.\ f(x) \not\simeq g(h(x)) \rightarrow f(c) \not\simeq g(h(c))$$

conflict based instantiation would derive only the first lemma, since $E \models f(a) \simeq g(h(a))$ makes the first instance conflicting with E. However, $E \not\models f(c) \simeq g(h(c))$, therefore the second instance is not conflicting. Note that the second lemma introduces a literal with the new terms $h(c)$ and $g(h(c))$, which did not previously appear in E, and whose relevance for the solving is a priori unknown.

Finding conflicting instances Intuitively, the problem of finding conflicting instances for a given quantified formula $\forall \bar{x}_n.\ \psi$ is much harder than instantiation based on E-matching and triggers. One must check that there is a substitution which makes the negation of ψ to be entailed by E, which can be arbitrarily complicated depending on the literals of ψ and the size of E. Reynolds et al. provide an algorithm to successively turn ψ into a set of matching constraints from which they eventually extract substitutions leading to conflicting instances. It has been implemented in CVC4 and is used as a preliminary phase for trigger based instantiation: the latter is invoked only when the former fails to produce conflicting instances. This led to a significant increase in the number of problems solved by the solver, while generally notably reducing the number of necessary instances to do so.

Example 3.7 ([Rey16])**.** Let $E = \{a \not\simeq c,\ f(b) \simeq b,\ g(b) \simeq a,\ f(a) \simeq a,\ h(f(a)) \simeq d,\ h(b) \simeq c\}$ and consider a quantified formula $\forall x.\ f(g(x)) \simeq h(f(x))$. The instance $f(g(b)) \simeq h(f(b))$ is conflicting with E, since $E \models f(g(b)) \not\simeq h(f(b))$. The search for this conflicting instance is performed by deriving constraints based on the fact that $E \models f(g(b)) \simeq a \wedge h(f(b)) \simeq c$ and $E \models a \not\simeq c$. Therefore if an assignment is provided for the variable x such that the constraints $f(g(x)) \simeq f(g(b))$ and $h(f(x)) \simeq h(f(b))$ hold then this assignment embodies a substitution leading to a conflicting instance of the quantified formula.

Propagating equalities A very interesting feature of the search for conflicting instances is that as a by-product it might propagate equalities over the terms of E, a desirable feature common in theory solvers in a CDCL($\mathcal{T}$) framework. Sometimes when conflicting instances are not found, the constraints produced during the search are equalities among terms in E. Generating instantiation lemmas based on these constraints will not refute the model, but will potentially reduce the search space. CVC4 also benefits from this technique in their instantiation module.

Example 3.8 ([Rey16])**.** If in Example 3.7 the literal $a \not\simeq c$ were not present in E then the given quantified formula would not have a conflicting instance. The instance $f(g(b)) \simeq h(f(b))$, however, together with E would entail that $a \simeq c$. Therefore deriving this instance propagates an equality for the terms in E.

Limitations The procedure presented by Reynolds et al. is efficient and greatly improves the instantiation module in CVC4, but no formal guarantees are given about the exact complexity of the problem it addresses or that conflicting instances for a given quantified formula will be found when they exist. A more general limitation for the search itself for conflicting instances is that accounting for theories beyond equality is particularly complicated. Remember that the search requires finding a substitution σ such that, for some theory $\mathcal{T}$, $E \models_{\mathcal{T}} \neg\psi\sigma$ holds. It is thus necessary to efficiently check entailment modulo $\mathcal{T}$. In a regular CDCL($\mathcal{T}$) solver this can be done very efficiently for the theory of equality by using the data structures in place for performing the congruence closure decision procedure, but the same does not apply to other theories.

Example 3.9 ([RTdM14])**.** Consider $\mathcal{T}$ to be a combination of the theories of equality and linear integer arithmetic. Let $E = \{f(a) \simeq b,\, g(a) \geq b + 1\}$ and consider the quantified formula $\forall x.\, f(x) \simeq g(x)$. The substitution $\sigma = \{x \mapsto a\}$ leads to the conflicting instance $f(a) \simeq g(a)$. However, finding it requires efficiently checking that $E \models_{\mathcal{T}} f(a) \not\simeq g(a)$, which is not derivable by equality reasoning only.

Another limitation is that quantified formulas are each considered independently. This means that there are instances from $\mathcal{Q}$ which in combination are conflicting with E but that cannot be found with this technique.

Example 3.10. Let $E = \{p(a)\}$ and $\mathcal{Q} = \{\forall x.\, q(x),\, \forall yz.\, \neg q(y) \vee \neg p(z)\}$. There are no substitutions σ, ρ such that $E \models \neg q(x)\sigma$ or $E \models q(y)\rho \wedge p(z)\rho$, even though $E \cup \mathcal{Q}$ is clearly inconsistent.

3.2.3 Model based instantiation

Model based quantifier instantiation (MBQI) is a complete instantiation technique for CDCL($\mathcal{T}$) solvers introduced by Ge and de Moura [GdM09]. It attempts to build a model for $E \cup \mathcal{Q}$ by systematically creating candidate interpretations $\mathcal{M}$ which satisfy E and evaluating whether they also satisfy $\mathcal{Q}$. The former is guaranteed by construction. The latter is checked, for each quantified formula $\forall \bar{x}_n.\ \psi \in \mathcal{Q}$, by looking for instances $\neg\psi\sigma$ which are $\mathcal{T}$-satisfied by $\mathcal{M}$, i.e. whether $\mathcal{M} \models_{\mathcal{T}} \neg\psi\sigma$ holds, given a background theory $\mathcal{T}$. If no such instance exists, then $\mathcal{M}$ satisfies $E \cup \mathcal{Q}$. Otherwise each of these instances is a witness of why $\mathcal{M}$ is not a suitable candidate. They are added to the original problem with instantiation lemmas, thus removing this candidate from consideration in the following iterations. The successive rounds of instantiation either lead to unsatisfiability or, when no conflicting instance is generated, to satisfiability with a concrete model. Ge and de Moura present several fragments of first-order logic in which MBQI is complete, including fragments containing arithmetic.

Example 3.11 ([GdM09])**.** Let $E \cup \mathcal{Q}$ be such that $E = \{f(a) \simeq 0,\ f(b) \simeq 1, a < 2\}$ and $\mathcal{Q} = \{\forall x.\ \neg(5 \leq x) \vee f(x) < 0\}$. A candidate model $\mathcal{M}$ which satisfies E could provide the following interpretations for a, b, and f

$$\begin{aligned} a^{\mathcal{M}} &= 0 \\ b^{\mathcal{M}} &= 2 \\ f^{\mathcal{M}} &= \lambda x.\ \mathit{ite}(x < 2,\ 0,\ \mathit{ite}(x < 5,\ 1,\ -1)) \end{aligned}$$

since $[\![f(a)]\!]^{\mathcal{M}} = 0$, $[\![f(b)]\!]^{\mathcal{M}} = 1$, and $[\![a]\!]^{\mathcal{M}} < 2$. To check whether the interpretation $\mathcal{M}$ satisfies $\forall x.\ \neg(5 \leq x) \vee f(x) < 0$ we consider the dual problem: is there a substitution σ such that $[\![(x > 5 \wedge f(x) \geq 0)\sigma]\!]^{\mathcal{M}}$ holds? Consider an arbitrary substitution σ, with $x\sigma$ being a term t. To satisfy the first conjunct $[\![t]\!]^{\mathcal{M}}$ must be greater than 5, therefore $[\![f(t)]\!]^{\mathcal{M}}$ must result in -1, since the conditions to the two *if-then-else* expressions in $f^{\mathcal{M}}$ will fail. Thus it is not possible to have a substitution σ to which $[\![(x > 5 \wedge f(x) \geq 0)\sigma]\!]^{\mathcal{M}}$ holds. Indeed $\mathcal{M}$ satisfies the quantified formula and therefore is a model for $E \cup \mathcal{Q}$.

The central challenges in MBQI are building the candidate interpretation and evaluating if it satisfies $\mathcal{Q}$. Ge and de Moura built candidates as above, using lambda and *if-then-else* expressions to create interpretations for function symbols. These interpretations coincide with the term-generated interpretations from E on known terms and map all other terms to given default values. The evaluation of $\mathcal{Q}$ in $\mathcal{M}$ is done by using a secondary ground CDCL($\mathcal{T}$) solver. For each quantified formula $\forall \bar{x}_n.\ \psi \in \mathcal{Q}$, the variables $\bar{x}_n$ are replaced by fresh constants $\bar{k}_n$, and then $[\![\neg\psi[\bar{k}_n]]\!]^{\mathcal{M}}$ is encoded into a ground formula and the secondary ground solver checks its satisfiability. If it is unsatisfiable then $[\![\forall \bar{x}_n.\ \psi]\!]^{\mathcal{M}}$ holds, otherwise the interpretations of $\bar{k}_n$ are used to derive a refining instantiation lemma.

Reynolds et al. [RTG+13] introduced alternative methods for constructing and checking

candidate interpretations. While their model construction is in theory similar to the technique by Ge and de Moura, their way of checking candidates is quite different. By representing candidates with a special data structure, they apply a "bottom-up" approach: an instantiation is picked for the variables of the quantified formula and an efficient evaluation engine determines whether the instantiation leads to the formula being interpreted as $\bot$, which would trigger a refining instantiation. This flavour of MBQI is combined by Reynolds et al. [RTGK13] with a model minimisation technique for E and the insertion of cardinality constraints in the solving in order to perform finite-model finding.

3.3 Other frameworks

Model Construction Satisfiability Calculus The Model-Constructing Satisfiability Calculus (mcSAT) is, analogously to CDCL($\mathcal{T}$), a generalization of CDCL that accommodates reasoning modulo theories. It was proposed by de Moura and Jovanović [dMJ13] and has been successfully applied for solving quantifier-free problems with non-linear arithmetic [JdM12] and problems with bit-vectors [ZWR16]. Differently from CDCL($\mathcal{T}$), in mcSAT the conflict-driven learning is performed not only at the propositional level but also at the theory level. The calculus incorporates *theory decisions*, i.e. heuristic assignments are made also for non-propositional variables, and therefore also the corresponding propagations and explanations of conflicts for the derivation of learned clauses at the theory level. How to incorporate first-order reasoning in the framework is yet to be explored.

Hierarchic Superposition The superposition [BG94, NR01] calculus is a powerful combination of first-order resolution with orderings and rewriting for equality reasoning. For many years it has been the go-to approach for automatic theorem proving in equational first-order logic. Performing theory reasoning, however, was not commonly associated with superposition-based systems despite the existence of a number of techniques for integrating it, as SMT solvers were generally more suitable to handle problems with theory symbols.

A prominent approach for handling theories efficiently in superposition is to extend the set of inference rules for abstracting away the theory reasoning, generating ground constraints to be later discharged. An off-the-shelf SMT solver, for instance, can be invoked to evaluate the constraints. This way the theory part of the problem is kept separated from the rest. This technique is called Hierarchic Superposition and was proposed by Bachmair et al. [BGW94] and later augmented by Baumgartner and Waldmann [BW13]. Refutational completeness is guaranteed in certain restricted fragments, but nevertheless it has been implemented successfully in the SPASS [AKW09, EKK+11] and Beagle [BBW15] theorem provers to integrate arithmetic reasoning into superposition.

AVATAR modulo theories The AVATAR architecture proposed by Voronkov [Vor14] tightly integrates a superposition theorem prover with a SAT solver in a layered approach. Similarly

to the CDCL($\mathcal{T}$) framework, the first-order problem is abstracted into a propositional one, with the SAT solver enumerating propositional assignments to be checked at the first-order level by a superposition engine. By employing an SMT solver instead of a SAT solver Reger et al. [RBSV16] extended the approach so that at the first-order level only assignments which are groundly satisfiable, modulo the fixed theories, are considered.

The architecture is implemented in the Vampire [KV13] theorem prover, using the Z3 SMT solver to check ground satisfiability. Moreover, the Vampire theorem prover is also capable of performing theory reasoning directly at the first-order level, by introducing theory axioms into the problem. These are generally quite explosive, but powerful heuristics are used so that the overall system performs well in problems containing linear and non-linear real and integer arithmetic, extensional arrays and algebraic data types.

Free-variables and theories Rümmer [Rüm08] introduced a constraint sequent calculus for first-order logic modulo linear integer arithmetic. It combines a sequent calculus, based on earlier work by Giese [Gie01] on free variable tableaux, with quantifier elimination, based on the classic Omega test integer programming algorithm [Pug92], yielding a powerful framework that is complete for first-order logic (without functions but with arbitrary uninterpreted predicates) and a decision procedure for Presburger arithmetic, i.e. quantified linear integer arithmetic, being also effective for their combination.

The final calculus enjoys similarities to CDCL($\mathcal{T}$). Among other techniques, such as using ground decision procedures for the ground parts of the problem, Rümmer integrated trigger based instantiation [Rüm12] as an *optimization* for his systematic handling of quantifiers. This way the completeness results are independent of trigger selection or other heuristics, which are used to potentially improve performance but do not hurt the formal guarantees of the framework. The calculus has been successfully implemented in the Princess [Rüm12] theorem prover.

A historical shortcoming of tableaux and sequent based calculi is the handling of equality. Backeman and Rümmer [BR15b] built on the above framework to introduce a novel way to perform efficient equality reasoning in such a framework for equational first-order logic. They introduced the problem of bounded simultaneous rigid E-unification (BREU), a decidable variant of the classic (and undecidable) problem of simultaneous rigid E-unification that is generally inherent to performing equality reasoning in these calculi. Their approach has also been successfully implemented in the Princess system.

3.4 Certificates

Most automatic provers that support the TPTP syntax for problems generate proofs in TSTP (Thousands of Solutions for Theorem Provers) format [SZS04]. A TSTP proof consists of a list of inferences. TSTP does not mandate any inference system; the meaning of the rules and the granularity of inferences vary across systems. For example, the E prover [Sch13] combines

clausification, skolemization, and variable renaming into a single inference, whereas Vampire [KV13] appears to cleanly separate preprocessing transformations. SPASS's [WDF+09] custom proof format does not record preprocessing steps; reverse engineering is necessary to make sense of its output, and optimizations ought to be disabled [BBF+16, Sect. 7.3].

Most SMT solvers can parse the SMT-LIB [BFT15] format, but each solver has its own output syntax. Z3's proofs can be quite detailed [dMB08a], but rewriting steps often combine many rewrites rules. CVC4's format is an instance of LF [HHP87] with Side Conditions (LFSC) [Stu09]; despite recent progress [HBR+15, KBT+16], neither skolemization nor quantifier instantiation are currently recorded in the proofs. Proof production in Fx7 [Mos08] is based on an inference system whose formula processing fragment is subsumed by ours; for example, skolemization is more ad hoc, and there is no explicit support for rewriting.

Proof assistants for dependent type theory, including Agda, Coq, Lean, and Matita, provide very precise proof terms that can be checked by relatively simple checkers, meeting De Bruijn's criterion [BW05]. Exploiting the Curry–Howard correspondence, a proof term is a λ-term whose type is the proposition it proves; for example, the term $\lambda x.\ x$, of type $A \to A$, is a proof that A implies A. Proof terms have also been implemented in Isabelle [BN00], but they slow down the system considerably and are normally disabled. Frameworks such as LF, LFSC, and Dedukti [CD07] provide a way to specify inference systems and proof checkers based on proof terms. Our encoding into λ-terms is vaguely reminiscent of LF.

Isabelle and the proof assistants from the HOL family (HOL4, HOL Light, HOL Zero, and ProofPower–HOL) are based on the LCF architecture [GMW79]. Theorems are represented by an abstract data type. A small set of primitive inferences derives new theorems from existing ones. This architecture is also the inspiration behind automatic systems such as Psyche [Gra13]. In Cambridge LCF, Paulson introduced an idiom, *conversions*, for expressing rewriting strategies [Pau83]. A conversion is an ML function from terms t to theorems of the form $t \simeq u$. Basic conversions perform β-reduction and other simple rewriting. Higher-order functions combine conversions. Paulson's conversion library culminates with a function that replaces Edinburgh LCF's monolithic simplifier. Conversions are still in use today in Isabelle and the HOL systems. They allow a style of programming that focuses on the terms to rewrite—the proofs arise as a side effect. Our framework is related, but we trade programmability for efficiency on very large problems. Remarkably, both Paulson's conversions and our framework emerged as replacements for earlier monolithic systems.

Over the years, there have been many attempts at integrating automatic provers into proof assistants. To reach the highest standards of trustworthiness, some of these bridges translate the proofs found by the automatic provers so that they can be checked by the proof assistant. The Tramp subsystem of ΩMEGA is one of the finest examples [Mei00]. For integrating superposition provers with Coq, De Nivelle studied how to build efficient proof terms for clausification and skolemization [dNiv02]. For SMT, the main integrations with proof reconstruction are CVC Lite in HOL Light [MBG06], haRVey (veriT's predecessor) in Isabelle/HOL [FMM+06], Z3 in HOL4

and Isabelle/HOL [BW10, BFSW11], veriT in Coq [AFG+11], and CVC4 in Coq [EKK+16]. Some of these simulate the proofs in the proof assistant using dedicated tactics, in the style of our simple checker for Isabelle (Sect. 6.3). Others employ reflection, a technique whereby the proof checker is specified in the proof assistant's formalism and proved correct; in systems based on dependent type theory, this can help keep proof terms to a manageable size. A third approach is to translate the SMT output into a proof text that can be inserted in the user's formalization; Isabelle/HOL supports veriT and Z3 in this way [BBF+16].

Proof assistants are not the only programs used to check machine-generated proofs. Otterfier invokes the Otter prover to check TSTP proofs from various sources [ZMSZ04]. GAPT imports proofs generated by resolution provers with clausifiers to a sequent calculus and uses other provers and solvers to transform the proofs [HLRR13, EHR+16]. Dedukti's $\lambda\Pi$-calculus modulo [CD07] has been used to encode resolution and superposition proofs [Bur13], among others. λProlog provides a general proof-checking framework that allows nondeterminism, enabling flexible combinations of proof search and proof checking [Mil15].

Part I

Instantiation

Chapter 4

Congruence closure with free variables

As shown in Section 3.2, a central problem when applying instantiation techniques to handle quantified formulas in the CDCL($\mathcal{T}$) framework is how to determine which instances to derive. Each of the main techniques, be it trigger, conflict or model based instantiation, contributes greatly to the efficiency of state-of-the-art solvers, yet each one is typically implemented independently, in an ad-hoc manner and within their own specific settings.

In this chapter we present a uniform framework for reasoning with quantified formulas in the theory of equality and uninterpreted functions in CDCL($\mathcal{T}$). We introduce the E-ground (dis)unification problem as the cornerstone of this framework, in which trigger, conflict and model based instantiation techniques can be cast. This problem relates to the classic problem of rigid E-unification [DV98], pervasive to sequent and tableaux based calculi for equational first-order logic. By exploiting the similarities between CDCL($\mathcal{T}$) and these frameworks, we build on conventional techniques to solve E-unification and present a decision procedure for E-ground (dis)unification: *Congruence Closure with Free Variables* (CCFV, for short), which extends the classic congruence closure algorithm [NO80, NO07], to accommodate free variables. We then show how to build on CCFV to perform trigger, conflict and model based instantiation. A detailed accounting of the implementation and experimental evaluation of CCFV and the instantiation techniques around it is given in Chapter 5. The work described in these chapters led to a joint publication with Pascal Fontaine and Andrew Reynolds [BFR17].

Conventions Throughout the chapter we assume, for simplicity, and without loss of generality, that all atomic formulas are equalities and that the term language is mono-sorted.

Given a set of ground terms $\mathbf{T}$ closed under the subterm relation and a congruence relation $\simeq$ on $\mathbf{T}$, a *congruence* over $\mathbf{T}$ is a subset of $\{s \simeq t \mid s, t \in \mathbf{T}\}$ closed under entailment. The *congruence closure* (CC, for short) of a set of equations E on a set of terms $\mathbf{T}$ is the least congruence on $\mathbf{T}$ containing E. Given a consistent set of equality literals E, two terms t_1, t_2 are said to be *congruent* iff $E \models t_1 \simeq t_2$ and *disequal* iff $E \models t_1 \not\simeq t_2$. The *congruence class* in $\mathbf{T}$ of a given term is the set of terms in $\mathbf{T}$ congruent to it. The *signature* of a term is the term itself for a nullary symbol, and $f(c_1, \ldots c_n)$ for a term $f(t_1, \ldots t_n)$, with c_i being the congruence class

of t_i. Since congruence classes are sets, function applications whose arguments are congruent have the same signature. If e.g. a and b are equal, then $f(a)$ and $f(b)$ have the same signature. The *signature class* of t is a set $[t]_E$ containing one and only one term in the congruence class of t for each signature. Considering again the previous example, since $f(a)$ and $f(b)$ have the same signature, only one of them would be in their signature class. Notice that the signature class of two terms in the same congruence class is the same set of terms, and is a subset of their congruence class. We drop the subscript in $[t]_E$ when E is clear from the context. The *set of signature classes* of E on a set of terms $\mathbf{T}$ is $E^{\text{CC}} = \{[t] \mid t \in \mathbf{T}\}$. We may overload the above notations when E is a set of equality literals, not only of equalities. In this case we consider the congruence closure with relation to the equalities in E.

We fix the entailment relation $\models$ to represent entailment modulo ground equational reasoning, i.e. given a set of ground equality literals E and ground terms s and t,

1. $E \models s \simeq t$ if and only if $[s]_E = [t]_E$, and

2. $E \models s \not\simeq t$ if and only if there is a disequality $s' \not\simeq t'$ or $t' \not\simeq s'$ in the set E', which is the closure of E with relation to disequalities, such that $[s]_E = [s']_E$ and $[t]_E = [t']_E$.

The entailment relation is extended in the expected manner to allow conjunctions and disjunctions in the right hand side.

4.1 E-ground (dis)unification

Solving E-ground (dis)unification amounts to finding substitutions such that literals containing free variables hold in a context of currently asserted ground literals. E-ground (dis)unification is intrinsically related to the classic problem of rigid E-unification and is also NP-complete. The classic problem of rigid E-unification is that of, given a set of equalities E and an equality $u \simeq v$, finding a substitution σ such that $u\sigma \simeq v\sigma$ follows from $E\sigma$ by ground equational reasoning, i.e. such that $E\sigma \models u\sigma \simeq v\sigma$ holds. This problem has been studied extensively in the context of automated theorem proving [Bec98, BS01, DV01], specially for the integration of equality reasoning into tableaux [Häh01] and sequent calculi [DV01] based procedures. The simultaneous version of the problem has to be considered in these cases, in which the same substitution σ must be a solution to E-unification problems $E_1\sigma \models u_1\sigma \simeq v_1\sigma$, ..., $E_n\sigma \models u_n\sigma \simeq v_n\sigma$. Simultaneous rigid E-unification (SREU, for short) was famously shown to be undecidable by Degtyarev and Voronkov [DV96]. Nevertheless, incomplete unification procedures can be used in such a way that an overall complete first-order calculus may be obtained, as shown by Degtyarev and Voronkov themselves [DV98] and by Giese [Gie02]. Recently Backeman and Rümmer [BR15b] provided a complete first-order calculus based on a restricted case of SREU in which the possible substitutions for variables are bound, yielding a decidable problem that they solve through an encoding into SAT, using an off-the-shelf SAT solver to compute solutions.

We define below a variant of (non-simultaneous) E-unification in which we consider equality literals instead of only equalities and in which E is ground. Moreover, we consider a conjunction of literals in the right hand side.

Definition 4.1 (E-ground (dis)unification)**.** *Given two conjunctive sets of equality literals E and L, with E ground, the* E-ground (dis)unification problem *is that of finding substitutions σ such that $E \models L\sigma$ holds.*

To show that E-ground (dis)unification is indeed a variant of rigid E-unification we prove below that any problem $E \models L\sigma$ can be recast with only equalities in E and a single equality in L:

Lemma 4.1 (Reduction to Rigid E-unification)**.** *To find solution substitutions σ for an E-ground (dis)unification problem $E \models L\sigma$ can be cast to the problem of finding substitutions σ such that $E^{eq}\sigma \models u\sigma \simeq v\sigma$, in which E^{eq} is a conjunctive set of equalities and u, v are terms.*

Proof. First, we can assume that E is closed under entailment w.r.t. disequalities. Indeed, for any set of ground equality literals E, retrieving all the disequalities that it entails can be done polynomially. Let $\top$ be a constant and $f_{\not\simeq}$ a binary function symbol, both not appearing in $E \cup L$. Each disequality $s \not\simeq t$ in $E \cup L$ is replaced with $f_{\not\simeq}(s, t) \simeq \top \wedge f_{\not\simeq}(t, s) \simeq \top$.

For the transformation to preserve the entailment relation it is necessary to show that the transformed disequalities in L are still entailed by the transformed disequalities in E modulo its equalities. Assume that there is a disequality $u \not\simeq v$ in L such that $E \models u\sigma \not\simeq v\sigma$. Since E is closed under entailment w.r.t. disequalities, there is a disequality $u' \not\simeq v'$ or $v' \not\simeq u'$ in E such that $E \models u' \simeq u\sigma \wedge v' \simeq v\sigma$. Considering the transformation of $E \cup L$ to remove disequalities, it is straightforward to check that the entailment

$$E \wedge f_{\not\simeq}(u', v') \simeq \top \wedge f_{\not\simeq}(v', u') \simeq \top \models f_{\not\simeq}(u\sigma, v\sigma) \simeq \top \wedge f_{\not\simeq}(v\sigma, u\sigma) \simeq \top \wedge L$$

continues to hold, as well as that applying the same process to all other disequalities in $E \models L\sigma$ preserves the entailment into the resulting

$$E^{eq} \models s_1\sigma \simeq t_1\sigma \wedge \cdots s_n\sigma \simeq t_n\sigma$$

which, by taking a fresh n-ary function symbol f, can then be transformed into the equivalent $E^{eq} \models f(s_1, \ldots, s_n) \simeq f(t_1, \ldots, t_n)$. Since E^{eq} contains only equations, there is a single equation in the conclusion and the removal of disequalities preserves the entailment relation, E-ground (dis)unification is shown to be an instance of rigid E-unification. □

Example 4.1. Consider the sets of equality literals $E = \{f(a) \simeq f(b),\ h(a) \simeq h(c),\ g(b) \not\simeq h(c)\}$ and $L = \{h(x_1) \simeq h(c),\ h(x_2) \not\simeq g(x_3),\ f(x_1) \simeq f(x_3),\ x_4 \simeq g(x_5)\}$. A solution σ for their E-ground (dis)unification problem $E \models L\sigma$ is $\sigma = \{x_1 \mapsto a,\ x_2 \mapsto c,\ x_3 \mapsto b,\ x_4 \mapsto g(x_5)\}$.

The above example shows that x_5 can be mapped to any term; this E-ground (dis)unification problem has infinitely many solutions. We show below how the set of all solutions of a given E-ground (dis)unification can be finitely represented. Without loss of generality, we assume that $\mathbf{T}(E \cup L)$ contains at least one ground term.

Theorem 4.2. *Given an E-ground (dis)unification problem, if a substitution σ exists such that $E \models L\sigma$, then there is an acyclic substitution σ' such that $\mathrm{ran}(\sigma') \subseteq \mathbf{T}(E \cup L)$, $\sigma'^{\star}$ is ground, and $E \models L\sigma'^{\star}$.*

Proof. First, we can assume that σ is ground. Indeed, if a non-ground substitution σ is such that $E \models \ell\sigma$, then $E \models \ell\sigma\sigma_g$ holds for any ground substitution σ_g. For convenience and without loss of generality, assume all terms in L are flat. We introduce for this proof the notations $E_\sigma = \{x \simeq x\sigma \mid x \in dom(\sigma)\}$ and $S_t = \{t' \mid t' \in \mathbf{T}(E \cup L \cup E_\sigma) \text{ and } E \cup E_\sigma \models t \simeq t'\}$, in which σ is a substitution and t is a term. Note that $E \models L\sigma^{\star}$ holds if and only if $E \cup E_\sigma \models L$ also does.

To compute the congruence closure of a set of equations E on a set of terms $\mathbf{T}$, it suffices to compute a congruence graph on $\mathbf{T}$. Two terms t, t' are equal according to E if and only if there is a path between them in the graph. There is a full edge in the graph between two terms in any equation of E, and there is a congruence edge between two terms such that all arguments are pairwise equal (i.e., there is a path between them). Some (congruence or full) redundant edges can be omitted if the extremities are already connected. The congruence graph is a least fixed point, so the explanation for the existence of a congruence edge can always be traced back to full edges only. Congruence edges can be ordered by their dependency.

Consider a congruence graph G induced by E on terms $\mathbf{T}(E \cup L \cup E_\sigma)$, and a congruence graph G' induced by furthermore adding edges for σ. Besides the edges corresponding to σ, that is, edges with a variable, the only additional edges are congruence edges; we assume no redundant edge is added to G'. Variables are at most linked to one term, and by a full edge. Other non-variable free term can only be linked either by a full edge to a variable, or to a term by a congruence edge.

All edges in $G' \setminus G$ involve at least one term u that is not (directly or indirectly) linked to any ground term in G. This is obviously true for full edges, since variables are not linked to any term in G. Consider the earliest (according to the dependency order of congruence edges) congruence edge between $f(t_1, \ldots, t_n)$ and $f(t'_1, \ldots, t'_n)$ contradicting this hypothesis. Then there is a path from $f(t_1, \ldots, t_n)$ to a ground term; this path cannot contain full edges, because this would involve two edges with a variable. Hence the path only contain congruence edges and $t_1, \ldots t_n$ are all equal to ground terms. The same holds for $f(t'_1, \ldots, t'_n)$, then t_i and t'_i are both linked to ground terms, for $i \in \{1, \ldots n\}$. One new earlier edge contradicts the hypothesis too.

As a corollary of the previous paragraph, given two terms t, t', if both terms are ground, $E \cup E_\sigma \models t \simeq t'$ if and only if $E \models t \simeq t'$.

In the following, we build an acyclic substitution σ' such that $ran(\sigma') \subseteq \mathbf{T}(E \cup L)$, and for two terms t_1, $t_2 \in \mathbf{T}(E \cup L)$, $E \cup E_\sigma \models t_1 \simeq t_2$ if and only if $E \cup E_{\sigma'} \models t_1 \simeq t_2$. Grounding σ'

is again trivial. For each variable x in $dom(\sigma)$, if S_x contains a ground term t in $\mathbf{T}(E \cup L)$, then $x\sigma' = t$. Otherwise if S_x contains a term in $u \in \mathbf{T}(L)$, for all variables y in S_x, $y\sigma' = u$. If S_x does not contain any term in $\mathbf{T}(E \cup L)$, a variable z in S_x is chosen, and for all variables y in S_x, $y\sigma' = z$. Trivially, $ran(\sigma') \subseteq \mathbf{T}(E \cup L)$.

Now we prove that $E \cup E_{\sigma'} \models L$ if and only if $E \cup E_\sigma \models L$, or further, that for any two terms t, t' in $\mathbf{T}(E \cup L)$, $E \cup E_{\sigma'} \models t \simeq t'$ if and only if $E \cup E_\sigma \models t \simeq t'$. Since $E \cup E_\sigma \models E \cup E_{\sigma'}$, the only non trivial direction is that, if $E \cup E_\sigma \models t \simeq t'$, $E \cup E_{\sigma'} \models t \simeq t'$. This has already be proven in the case of two ground terms. By construction σ and σ' induce the same partition on the variables, so this is true also if t and t' are two variables. If there is a path between t and t' with only congruence edges in the graph for $E \cup E_\sigma$, t and t' are still congruent according to $E \cup E_{\sigma'}$, thanks to the fact that equality between variables and equality between ground terms is preserved. As a corollary, any equality between terms in $\mathbf{T}(E \cup L)$ is preserved.

The substitution σ' is acyclic. Otherwise for some n,

$$E \cup E_\sigma \models x_i \simeq f_i(\dots, x_{(i+1) \bmod n}, \dots)$$

($i \in \{1, \dots n\}$) and also, since σ is ground,

$$E \cup E_\sigma \models x_1 \simeq f_1(\dots, x_{(2 \bmod n)}, \dots) \wedge x_1 \simeq t$$

for some ground term t. Thus $f_1(\dots, x_{(2 \bmod n)}, \dots)$ and $t = f_j(\dots, t', \dots)$ are congruent, therefore $j = 1$ and $x_{(2 \bmod n)}$ and t' should be equal. By successive application of the same reasoning step, there exists some x_i and a constant a such that $E \cup E_\sigma \models x_i \simeq a \simeq f_i(\dots, x_{(i+1) \bmod n}, \dots)$, which contradicts the fact that only one equality contains x_i in E_σ. □

From now on we only consider solutions σ, for a given E-ground (dis)unification problem $E \models L\sigma$, such that σ is acyclic, $ran(\sigma) \subseteq \mathbf{T}(E \cup L)$ and $E \models L\sigma'^\star$.

As a corollary of Theorem 4.2, the E-ground (dis)unification problem is in NP: it suffices indeed to guess an acyclic substitution mapping variables to terms in $\mathbf{T}(E \cup L)$ whose fixpoint is a ground substitution and check (polynomially) that it is a solution. The problem is also NP-hard, by reduction of 3-SAT.

Theorem 4.3 (E-ground (dis)unification is NP-complete)**.** *Finding solutions for E-ground (dis)unification is NP-complete.*

Proof. E-ground (dis)unification is NP since it can be verified, in polynomial time, with a classic congruence closure procedure, for instance, that a substitution σ solves the entailment problem. The proof of its NP-hardness is done through an encoding of 3-SAT into the entailment.

Let $\mathcal{C}$ be a set of 3-clauses containing literals over a set of propositions $\mathcal{P}$. Let $\top, \bot$ be constants and P a unary operator such that, for any proposition, $\mathrm{P}(p) = \top$ and $\mathrm{P}(\neg p) = \bot$. For each clause $C \in \mathcal{C}$, let f_C be a ternary function. For each proposition $p \in \mathcal{P}$, let x_p be a variable.

The set of equality literals $E \cup L$ is built such that

$$E = \bigcup_{C=\ell_1\vee\ell_2\vee\ell_3\in\mathcal{C}} \left\{ f_C(\mathrm{P}(\ell_1),\mathrm{P}(\ell_2),\mathrm{P}(\ell_3)) \simeq \top \;\middle|\; \begin{array}{l} \mathrm{P}(\ell_1)=\top \text{ or } \mathrm{P}(\ell_2)=\top \\ \text{or } \mathrm{P}(\ell_3)=\top \end{array} \right\}$$

$$L = \bigcup_{C=\ell_1\vee\ell_2\vee\ell_3\in\mathcal{C}} \left\{ f_C(x_{p_1},x_{p_2},x_{p_3}) \simeq \top \;\middle|\; \ell_i = p_i \text{ or } \ell_i = \neg p_i,\ i\in\{1..3\} \right\}$$

With this encoding the search for a substitution σ such that $E \models L\sigma$ can be seen as determining the satisfiability of $\mathcal{C}$. □

4.1.1 Recasting instantiation techniques

We show below how each of the main instantiation techniques used in the CDCL($\mathcal{T}$) framework are recast using E-ground (dis)unification. This forms the cornerstone of a framework based on a single solving method to be used in a modular way when handling instantiation techniques in CDCL($\mathcal{T}$).

4.1.1.1 Trigger based instantiation

As discussed in Section 3.2.1, a *trigger* T for a quantified formula $\forall\bar{x}_n.\,\psi$ is a set of non-ground terms which collectively contain all variables bound by the quantifier, i.e. $T = \{u_1, \ldots, u_m\}$ such that $u_1, \ldots, u_m \in \mathbf{T}(\psi)$ and $\{\bar{x}_n\} \subseteq \mathrm{FV}(u_1)\cup\cdots\cup\mathrm{FV}(u_m)$. Instantiations are determined by E-matching all terms in T with terms in $\mathbf{T}(E)$, such that resulting substitutions allow instantiating $\forall\bar{x}_n.\,\psi$ into ground formulas. Computing such substitutions amounts to solving the E-ground (dis)unification problem

$$E \models (u_1 \simeq y_1 \wedge \cdots \wedge u_m \simeq y_m)\,\sigma$$

with the further restriction that σ is acyclic, $ran(\sigma) \subseteq \mathbf{T}(E \cup L)$ and σ is ground. This forces each y_i to be grounded into a term in $\mathbf{T}(E)$, thus enumerating all possibilities for E-matching u_i. The desired instantiations are obtained by restricting the found solutions to $\bar{x}_n$.

Example 4.2. Consider the conjunctive sets $E = \{f(a) \simeq g(b),\ h(a) \simeq b,\ f(a) \simeq f(c)\}$ and $\mathcal{Q} = \{\forall x.\, f(x) \not\simeq g(h(x))\}$. Triggers from $\mathcal{Q}$ are $T_1 = \{f(x)\}$, $T_2 = \{h(x)\}$, $T_3 = \{f(x), g(h(x))\}$ and so on. The instantiations from these triggers are derived from the solutions yielded by for the respective problems:

- $E \models (f(x) \simeq y)\sigma$, which is solved by the substitutions $\sigma_1 = \{y \mapsto f(a),\ x \mapsto a\}$ and $\sigma_2 = \{y \mapsto f(c),\ x \mapsto c\}$

  By considering the part of the solutions that range over x, the instantiation lemmas generated would be, as in Example 4.2,

$$\forall x.\, f(x) \not\simeq g(h(x)) \rightarrow f(a) \not\simeq g(h(a)) \tag{4.1}$$

and

$$\forall x.\ f(x) \not\simeq g(h(x)) \to f(c) \not\simeq g(h(c)) \tag{4.2}$$

– $E \models (h(x) \simeq y)\sigma$, solved by $\sigma = \{y \mapsto h(a), x \mapsto a\}$, yielding only the instantiation lemma 4.1.

– $E \models (f(x) \simeq y_1 \wedge g(h(x)) \simeq y_2)\sigma$, by $\sigma = \{y_1 \mapsto f(a), y_2 \mapsto g(b), x \mapsto a\}$, also yielding only the instantiation lemma 4.1.

4.1.1.2 Conflict based instantiation

As discussed in Section 3.2.2, given a ground model $E \cup \mathcal{Q}$, a conflicting instance with E for a quantified formula $\forall \bar{x}_n.\ \psi \in \mathcal{Q}$ is some ground $\psi\sigma$ such that $E \models \neg\psi\sigma$. Finding a conflicting instance amounts to solving the E-ground (dis)unification problem

$$E \models \neg\psi\sigma, \text{ for some } \forall \bar{x}_n.\ \psi \in \mathcal{Q}$$

since $\neg\psi$ is a conjunction of equality literals.

Example 4.3. Let E and $\mathcal{Q}$ be as in Example 4.2. Finding conflicting instances corresponds to solving the problem

$$E \models (f(x) \simeq g(h(x)))\,\sigma$$

whose unique solution is $\sigma = \{x \mapsto a\}$. It leads to the instantiation lemma

$$\forall x.\ f(x) \not\simeq h(g(x)) \to f(a) \not\simeq h(g(a))$$

which forces the derivation of a new candidate model $E \cup \mathcal{Q}$.

4.1.1.3 Model based instantiation (MBQI)

As discussed in Section 3.2.3, the satisfiability of $E \cup \mathcal{Q}$ is assessed by attempting to build a model and checking whether it indeed satisfies both E and $\mathcal{Q}$. The interesting case is the satisfaction of $\mathcal{Q}$. It is determined by independently evaluating each of its quantified formulas in the candidate interpretation. This evaluation is performed in much the same way as the search for conflicting instances described in Section 3.2.2 and revisited above in terms of E-ground (dis)unification. We build on this observation to present below an alternative way of performing MBQI. We extend E and look for conflicting instances between $\mathcal{Q}$ and this extension through an entailment check defined in terms of E-ground (dis)unification. This way one can evaluate if every model of the resulting extension E_{TOT} is also a model of $E \cup \mathcal{Q}$ and, in the negative case, derive instantiations that refine the construction of E_{TOT}. Here we follow the model construction guidelines by Reynolds et al. [RTG+13].

Model construction A distinguished term e^τ is associated to each sort $\tau \in \mathcal{S}$. For each $f \in \mathcal{F}$ with sort $\langle \tau_1, \ldots, \tau_n, \tau \rangle$ a *default value* ξ_f is defined such that

$$\xi_f = \begin{cases} f(t_1, \ldots, t_n) \in \mathbf{T}(E) & \text{if } [t_1] = [e^{\tau_1}], \ldots, [t_n] = [e^{\tau_n}] \\ \text{some } t \in \mathbf{T}(E) & \text{otherwise} \end{cases}$$

The extension is built such that all ground terms not occurring in E which might be considered when evaluating $\mathcal{Q}$ are present in the congruence closure of E_{TOT}, according to the respective default values; and all terms in $\mathbf{T}(E)$ not asserted equal are explicitly asserted disequal, i.e.

$$\begin{aligned} E_{\text{TOT}} \quad = E \cup \; & \bigcup\nolimits_{t_1, t_2 \in \mathbf{T}(E)} \{t_1 \not\simeq t_2 \mid E \not\models t_1 \simeq t_2\} \\ & \bigcup\nolimits_{\forall \bar{x}_n.\, \psi \in \mathcal{Q}, \mathbf{t} \in \mathbf{T}(E)} \left\{ f(\bar{s}_n)\sigma \simeq \xi_f \;\middle|\; \begin{array}{l} \sigma = \{\bar{x} \mapsto \bar{t}\}, f(\bar{s}_n) \in \mathbf{T}(\psi) \text{ and} \\ f(\bar{s}_n)\sigma \text{ is not in the CC of } E. \end{array} \right\} \end{aligned}$$

Finding model conflicting instances As above, finding conflicting instances amounts to solving the E-ground (dis)unification problem

$$E_{\text{TOT}} \models \neg\psi\sigma, \text{ for some } \forall \bar{x}_n.\, \psi \in \mathcal{Q}$$

Example 4.4. Let $E = \{f(a) \simeq g(b),\ h(a) \simeq b\}$, $\mathcal{Q} = \{\forall x.\ f(x) \not\simeq g(x),\ \forall xy.\ \psi\}$ and $e = a$, with all terms having the same sort. The computed default values of the function symbols are $\xi_f = f(a), \xi_g = a, \xi_h = h(a)$. For simplicity, the extension E_{TOT} is shown explicitly only for $\forall x.\ f(x) \not\simeq g(x)$,

$$\begin{aligned} E_{\text{TOT}} \quad = E \cup \; & \{a \not\simeq b, a \not\simeq f(a), b \not\simeq f(a)\} \\ \cup \; & \{f(b) \simeq f(a), f(f(a)) \simeq f(a), g(a) \simeq a, g(f(a)) \simeq a\} \cup \{\ldots\} \end{aligned}$$

Solving the E-ground (dis)unification

$$\{\ldots, f(a) \simeq g(b), f(b) \simeq f(a), \ldots\} \models f(x) \simeq g(x)\sigma$$

leads to the solution $\sigma = \{x \mapsto b\}$, from which the instantiation lemma

$$\forall x.\, f(x) \not\simeq g(x) \rightarrow f(a) \not\simeq g(a)$$

will prevent the derivation of the candidate model E_{TOT} again.

4.2 Calculus

The intrinsic relation between E-unification and congruence closure has been investigated by Goubault [Gou93] and Tiwari et al. [TBR00]. They integrate variations of the classic procedure with first-order rewriting techniques and search for solutions guided by the structure of the terms. We build on these ideas and propose a calculus to find substitutions σ solving an E-ground (dis)unification problem $E \models L\sigma$. This calculus, *Congruence Closure with Free Variables*

(CCFV), uses a congruence closure algorithm as a core element to guide the search and build solutions. We share with these related works the belief that the congruence closure algorithms, being very efficient at checking solutions, can also be the core of efficient algorithms to discover them. CCFV however differs from those previous techniques notably, since it handles disequalities and more importantly since the search for solutions is pruned based on the structure of a ground model, which makes it most suitable for a CDCL($\mathcal{T}$) context.

The calculus proceeds by building a conjunctive set of equations E_σ such that $E \cup E_\sigma \models L$, in which E_σ corresponds to a solution substitution built based on the entailment conditions of the literals in L. These conditions are determined according to the structure of the literals and to the signature classes in E. We restrict ourselves to signature classes rather than congruent classes so that less possibilities for solutions need to be considered, since terms with the same signature are indistinguishable with relation to entailment conditions.

Theorem 4.4 (Entailment conditions)**.** *Given a consistent conjunctive set of ground equality literals E and an equality literal ℓ, a ground substitution σ, as defined in Theorem 4.2, exists such that $E \models \ell\sigma$ if and only if*

1. *ℓ is $x \simeq y$ and*
   a) *$x\sigma = y\sigma$ or*
   b) *there are ground terms t_1, t_2 such that $x\sigma \in [t_1]$, $y\sigma \in [t_2]$, and $[t_1] = [t_2]$*
2. *ℓ is $x \simeq f(s_1, \ldots, s_n)$, x occurs in $f(s_1, \ldots, s_n)$, and there are ground terms t_1, $t_2 \in \mathbf{T}(E)$ such that $x\sigma \in [t_1]$, $f(s_1, \ldots, s_n)\sigma \in [t_2]$, and $[t_1] = [t_2]$*
3. *ℓ is $x \simeq f(s_1, \ldots, s_n)$, x does not occur in $f(s_1, \ldots, s_n)$ and*
   a) *$x\sigma = f(s_1, \ldots, s_n)\sigma$ or*
   b) *there are ground terms t_1, t_2 such that $x\sigma \in [t_1]$, $f(s_1, \ldots, s_n)\sigma \in [t_2]$, and $[t_1] = [t_2]$*
4. *ℓ is $f(u_1, \ldots, u_n) \simeq g(v_1, \ldots, v_n)$ and*
   a) *$f = g$ and $E \models u_1\sigma \simeq v_1\sigma, \ldots, E \models u_n\sigma \simeq v_n\sigma$ or*
   b) *there are ground terms t_1, $t_2 \in \mathbf{T}(E)$ such that $[t_1] = [t_2]$, $f(u_1, \ldots, u_n)\sigma \in [t_1]$, and $g(v_1, \ldots, v_n)\sigma \in [t_2]$*
5. *ℓ is $u \not\simeq v$ and there are ground terms t_1, $t_2 \in \mathbf{T}(E)$ such that $E \models t_1 \not\simeq t_2$, $u\sigma \in [t_1]$, and $v\sigma \in [t_2]$*

Proof. Since ℓ is either an equality or a disequality, each term is either a variable or a function application, and a variable either occurs or not in a term, the cases shown are exhaustive for the structure of ℓ. For each case we show that if, for some arbitrary substitution σ, $E \models \ell\sigma$, then one of the side conditions must hold, and that if any side condition holds, then $E \models \ell\sigma$.

Case 1: Condition (1*a*) states that $x\sigma$ and $y\sigma$ are the same term, while (1*b*) that they are (possibly different) congruent terms. Each implies that the entailment holds. If $E \models \ell\sigma$ and both (1*a*) and (1*b*) fail, then $x\sigma$ and $y\sigma$ are not congruent terms, which is a contradiction.

Case 2: the side condition implies the entailment. If $E \models \ell\sigma$, and since x occurs in $f(s_1, \ldots, s_n)$, then $x\sigma$ and $f(s_1, \ldots, s_n)\sigma$ are different terms occurring in the same congruence class. This class must contain both a term t and a term $f(t_1, \ldots, t_n)$ on which the former is congruent to a subterm of the latter. This can only be the case if both t and $f(t_1, \ldots, t_n)$ occur in E.

Case 3: Similarly to 1, both conditions imply that the entailment holds and it holding while the conditions do not is a contradiction. The conditions are independent of x occurring on $f(s_1, \ldots, s_n)$, since (3*b*) does not restrict the ground terms t_1, t_2.

Case 4: By congruence, if condition (4*a*) holds then the function applications are congruent, while (4*b*) directly implies they are. If $E \models \ell\sigma$ and since it is not possible for two different function applications not in $\mathbf{T}(E)$ to be congruent, then $f = g$ and $f(u_1, \ldots, u_n)\sigma$ and $g(v_1, \ldots, v_n)\sigma$ are the same function application over congruent arguments, or $f(u_1, \ldots, u_n)\sigma$ and $g(v_1, \ldots, v_n)\sigma$ are ground terms occurring in E and in the same congruence class in E^{CC}.

Case 5: Since two terms not in $\mathbf{T}(E)$ cannot be entailed disequal unless they are congruent to two terms in $\mathbf{T}(E)$ which are disequal, $E \models \ell\sigma$ holds if and only if $u\sigma$ and $v\sigma$ are in the congruence classes of disequal terms occurring in E. □

Note that in cases (1), (3) and (4*a*) the entailment does not necessarily depend on E, i.e. a solution that make the terms syntactically equal is possible. The other cases, however, can only be entailed if there are certain terms in E into which the non-ground terms can be made equal to by the substitution.

Example 4.5. Considering again E and L as in Example 4.1, the calculus should find a substitution σ such that

$$f(a) \simeq f(b) \wedge h(a) \simeq h(c) \wedge g(b) \not\simeq h(c) \models \left(h(x_1) \simeq h(c) \wedge h(x_2) \not\simeq g(x_3) \wedge f(x_1) \simeq f(x_3) \wedge x_4 \simeq g(x_5)\right)\sigma$$

This substitution can be built by analyzing the entailment conditions for each of the literals in L and how they can be consistently combined. A conjunctive set of equations E_σ from which such a substitution can be derived, i.e. such that $E \cup E_\sigma \models L$, is built considering that:

- $h(x_1) \simeq h(c)$: either $x_1 \simeq c$, due to condition (4*a*), or $x_1 \simeq a$, due to (4*b*), appears in E_σ;
- $h(x_2) \not\simeq g(x_3)$: either $x_2 \simeq c \wedge x_3 \simeq b$ or $x_2 \simeq a \wedge x_3 \simeq b$, both due to condition (5), appears in E_σ;
- $f(x_1) \simeq f(x_3)$: either $x_1 \simeq x_3$, due to (4*a*), or $x_1 \simeq a \wedge x_3 \simeq b$ or $x_1 \simeq b \wedge x_3 \simeq a$, both due to (4*b*), must be in E_σ;
- $x_4 \simeq g(x_5)$: due to condition (3*a*), the literal itself must be in E_σ.

A possible solution is thus $E_\sigma = \{x_1 \simeq a, x_2 \simeq a, x_3 \simeq b, x_4 \simeq g(x_5)\}$, corresponding to the acyclic substitution $\sigma = \{x_1 \mapsto a, x_2 \mapsto a, x_3 \mapsto b, x_4 \mapsto g(x_5)\}$. Note that, for any ground term $t \in \mathbf{T}(E \cup L)$, $\sigma_g = \sigma \cup \{x_5 \mapsto t\}$ is such that $ran(\sigma_g) \subseteq \mathbf{T}(E \cup L)$, ${\sigma_g}^\star$ is ground, and $E \models L{\sigma_g}^\star$.

Given an E-ground (dis)unification problem $E \models L\sigma$, the CCFV calculus computes the various possible E_σ corresponding to a coverage of all substitution solutions, i.e. such that $E \cup E_\sigma \models L$, whose range is in $\mathbf{T}(E \cup L)$. We describe the calculus as a set of rules that operate on states of the form $E_\sigma \Vdash_E C$, in which C is a formula stemming from the decomposition of L into an AND-OR combination of simpler constraints, and E_σ is a conjunctive set of equalities representing a partial solution for a given combination. Starting from the initial state $\top \Vdash_E L$ the right hand side of the state is progressively decomposed, whereas the left side is step by step augmented with new equalities building the candidate solution. $\top$ represents an empty conjunctive set. Moreover, as the partial solution is built the remaining constraints are updated to reflect the current assignments of variables, therefore searching only for further solutions which are compatible with the current one. Example 4.5 shows that, for a literal to be entailed by $E \cup E_\sigma$, sometimes several solutions E_σ exist, thus the calculus involves branching. To simplify the presentation, the rules do not apply branching directly, but build disjunctions on the right part of the state which later lead to branching. A branch is closed when its constraint is decomposed into either $\bot$ or $\top$. The latter are branches for which $E \cup E_\sigma \models L$ holds.

The set of CCFV derivation rules is presented in Table 4.1. As a convention, t stands for a ground term, x, y for variables, u for non-ground terms, $u_1, \ldots, u_n$ for terms such that at least one is non-ground and s, $s_1, \ldots, s_n$ for terms in general. Rules are applied top-down, the symmetry of equality being used implicitly. Each rule simplifies the constraint of the right hand side of the state, and as a consequence any derivation strategy is terminating (Theorem 4.5).

When an equality is added to the left hand side of a state $E_\sigma \Vdash_E C$ (rule Assign), the constraint C is normalized with respect to congruence closure to reflect the assignments to variables. That is, all terms in C are representatives of classes in the congruence closure of $E \cup E_\sigma$. We write

$$rep(E_\sigma, x) = \begin{cases} \text{some chosen } y \in [x]_{E_\sigma} & \text{if all terms in } [x]_{E_\sigma} \text{ are variables} \\ rep(E_\sigma, f(\bar{s}_n)) & \text{otherwise, for some } f(\bar{s}_n) \in [x]_{E_\sigma} \end{cases}$$

$$rep(E_\sigma, f(s_1, \ldots, s_n)) = \begin{cases} f(s_1, \ldots, s_n) & \text{if } f(s_1, \ldots, s_n) \text{ is ground} \\ f(rep(E_\sigma, s_1), \ldots, rep(E_\sigma, s_n)) & \text{otherwise} \end{cases}$$

and write $rep(E_\sigma, C)$ to denote the result of applying rep according to $E \cup E_\sigma$ on both sides of each literal $s \simeq s'$ or $s \not\simeq s'$ in C. The above definition of rep leaves room for some choice of representative, but soundness and completeness are not impacted by the choice. What actually matters is whether the representative is a variable, a ground term or a non-ground function application. The Assign rule adds equations from the right side of the state into the tentative solution in the left side of the state: it extends E_σ with the mapping for a variable. Because C

$$\frac{E_\sigma \Vdash_E x \simeq s \wedge C}{E_\sigma \cup \{x \simeq s\} \Vdash_E rep(\{x \simeq s\}, C)}\ \textsc{Assign} \qquad \text{if } x \notin \mathrm{FV}(s)$$

$$\frac{E_\sigma \Vdash_E x \simeq f(\bar{u}_n) \wedge C}{E_\sigma \Vdash_E \bigvee_{[t] \in E^{\mathrm{cc}},\ f(\bar{t}_n) \in [t]} (x \simeq t \wedge u_1 \simeq t_1 \wedge \cdots \wedge u_n \simeq t_n \wedge C)}\ \textsc{Uvar} \quad \text{if } x \in \mathrm{FV}(f(\bar{u}_n))$$

$$\frac{E_\sigma \Vdash_E f(\bar{u}_n) \simeq f(\bar{s}_n) \wedge C}{\begin{array}{ll} E_\sigma & \Vdash_E \quad (u_1 \simeq s_1 \wedge \cdots \wedge u_n \simeq s_n \wedge C) \vee \\ & \quad \bigvee_{[t] \in E^{\mathrm{cc}},\, f(\bar{t}_n) \in [t],\, f(\bar{t}'_n) \in [t]} \left(\begin{array}{l} u_1 \simeq t_1 \wedge \cdots \wedge u_n \simeq t_n \wedge \\ s_1 \simeq t'_1 \wedge \cdots \wedge s_n \simeq t'_n \ \wedge C \end{array} \right) \end{array}}\ \textsc{Ucomp}$$

$$\frac{E_\sigma \Vdash_E f(\bar{u}_n) \simeq g(\bar{s}_m) \wedge C}{E_\sigma \quad \Vdash_E \quad \bigvee_{\substack{[t] \in E^{\mathrm{cc}}, \\ f(\bar{t}_n) \in [t],\, g(\bar{t}'_m) \in [t]}} \left(\begin{array}{l} u_1 \simeq t_1 \wedge \cdots \wedge u_n \simeq t_n \ \wedge \\ s_1 \simeq t'_1 \wedge \cdots \wedge s_m \simeq t'_m \wedge C \end{array} \right)}\ \textsc{Ugen} \quad \text{if } f \neq g$$

$$\frac{E_\sigma \Vdash_E x \not\simeq y \wedge C}{E_\sigma \Vdash_E \bigvee_{[t],\, [t'] \in E^{\mathrm{cc}},\, E \models t \not\simeq t'} (x \simeq t \wedge y \simeq t' \wedge C)}\ \textsc{Dvar}$$

$$\frac{E_\sigma \Vdash_E x \not\simeq f(\bar{s}_n) \wedge C}{E_\sigma \quad \Vdash_E \quad \bigvee_{\substack{[t],\, [t'] \in E^{\mathrm{cc}}, \\ E \models t \not\simeq t',\, f(\bar{t}'_n) \in [t']}} (x \simeq t \wedge s_1 \simeq t'_1 \wedge \cdots \wedge s_n \simeq t'_n \wedge C)}\ \textsc{Dfapp}$$

$$\frac{E_\sigma \Vdash_E f(\bar{u}_n) \not\simeq g(\bar{s}_m) \wedge C}{E_\sigma \quad \Vdash_E \quad \bigvee_{\substack{[t],\, [t'] \in E^{\mathrm{cc}}, E \models t \not\simeq t', \\ f(\bar{t}_n) \in [t],\, g(\bar{t}'_m) \in [t']}} \left(\begin{array}{l} u_1 \simeq t_1 \wedge \cdots \wedge u_n \simeq t_n \ \wedge \\ s_1 \simeq t'_1 \wedge \cdots \wedge s_m \simeq t'_m \wedge C \end{array} \right)}\ \textsc{Dgen}$$

$$\frac{E_\sigma \Vdash_E C_1 \vee C_2}{E_\sigma \Vdash_E C_1 \qquad E_\sigma \Vdash_E C_2}\ \textsc{Split} \qquad \frac{E_\sigma \Vdash_E \ell \wedge C}{E_\sigma \Vdash_E C}\ \textsc{Yield} \quad \text{if } E \models \ell$$

$$\frac{E_\sigma \Vdash_E \ell \wedge C}{E_\sigma \Vdash_E \bot}\ \textsc{Fail} \quad \text{if } \ell \text{ is ground and } E \not\models \ell$$

Table 4.1: The CCFV calculus in equational FOL. E is fixed from a problem $E \models L\sigma$.

is replaced by $rep(\{x \simeq s\}, C)$, one variable (either x, or s, if it is a variable, depending on the representative chosen) disappears from the right side.

The other rules can be divided into two categories. The branching rules (UVAR through DGEN) enumerate all possibilities for deriving the entailment of some literal from C. The rules are based on the entailment conditions from Theorem 4.4. Condition (1) is represented by rule ASSIGN. Condition (2) by rule UVAR. Condition (3) by rule ASSIGN. Condition (4*b*) is represented by rules UCOMP and UGEN, with the first also accounting for condition (4*a*). Condition (5) is handled by rules DVAR, DFAPP and DGEN, which restrict the terms to consider according to the structure of the constraint.

Example 4.6. The rule UCOMP enumerates the possibilities for which a literal of the form $f(u_1, \ldots, u_n) \simeq f(s_1, \ldots, s_n)$ is entailed, which may be due to their arguments being congruent, since both terms have the same top symbol, or by E-matching f-terms occurring in the same signature class of E^{CC}. The generated entailment conditions are that either $u_1 \simeq s_1, \ldots, u_n \simeq s_n$ can be entailed or there is a term $t \in \mathbf{T}(E)$ into which both $f(u_1, \ldots, u_n)$ and $f(s_1, \ldots, s_n)$ can be entailed equal. For the second condition to hold there must be f-terms $f(t_1, \ldots, t_n)$ and $f(t'_1, \ldots, t'_n)$ in $[t]$ such that $u_1 \simeq t_1, \ldots, u_n \simeq t_n$ and $s_1 \simeq t'_1, \ldots, s_n \simeq t'_n$ can be entailed.

The structural rules (SPLIT, FAIL and YIELD) create or close branches. SPLIT creates branches when there are disjunctions in the constraint. FAIL closes a branch when it is no longer possible to build on the current solution to entail the remaining constraints, i.e. a ground constraint was derived that cannot be entailed by E. YIELD removes constraints from the right hand side of the state. When no more constraints remain to be entailed, the empty conjunction $\top$ is derived. In this case E_σ embodies a set of solutions for the given E-ground (dis)unification problem.

Each conjunctive set E_σ defines a substitution $\sigma = \{x \mapsto rep(E_\sigma, x) \mid x \in \text{FV}(L)\}$. If a branch is closed with YIELD, the set $\text{SOLS}(E_\sigma)$ of all ground solutions extractable from E_σ is composed of substitutions σ_g which extend σ by mapping all variables in $ran(\sigma^\star)$ into ground terms in $\mathbf{T}(E \cup L)$, such that each σ_g is acyclic, $\sigma_g^\star$ is ground and $E \models L\sigma_g^\star$.

4.2.1 Strategy

A possible derivation strategy for CCFV, given an initial state $\top \Vdash_E L$, is to apply the sequence of steps described below at each state $E_\sigma \Vdash_E C$. Let SEL be a function that selects a literal from a conjunction according to some heuristic, such as selecting first literals with less variables or literals whose top symbols have less ground signatures in E^{CC}. The result of SEL is denoted *selected literal*. Since no two rules can be applied on the same literal, the function SEL effectively enforces an order on the application of the rules. Moreover, the branch selection strategy assumes that each branch is explored to the end before a new one is considered.

1. *Select branch*: While C is a disjunction, apply SPLIT and consider the leftmost branch, by

convention.

2. *Simplify constraint*: Apply the rule for which $\text{SEL}(C)$ is amenable.
3. *Discard failure*: If FAIL was applied or a branching rule had the empty disjunction as a result, discard this branch and consider the next open branch.
4. *Mark success*: If the right hand side has been reduced to $\top$, all remaining constraints in the branch are entailed by $E \cup E_\sigma$. Mark the branch as successful and then consider the next open branch.

A solution σ for the E-ground (dis)unification problem $E \models L\sigma$ can be extracted at each successful branch.

Example 4.7. Consider again the sets $E = \{f(a) \simeq f(b), h(a) \simeq h(c), g(b) \not\simeq h(c)\}$ and $L = \{h(x_1) \simeq h(c), h(x_2) \not\simeq g(x_3), f(x_1) \simeq f(x_3), x_4 \simeq g(x_5)\}$, as in Example 4.1. The set of signature classes of E, which coincides with its set of congruence classes, is

$$E^{\text{CC}} = \{\{a\}, \{b\}, \{c\}, \{f(a), f(b)\}, \{h(a), h(c)\}, \{g(b)\}\}$$

with the disequalities entailed by E being $g(b) \not\simeq h(c)$ and $g(b) \not\simeq h(a)$.

Let SEL select first literals in C with the minimal number of variables. The derivation tree produced by CCFV for this problem is shown below. Selected literals are underlined. Disjunctions and the application of SPLIT are kept implicit to simplify the presentation, as is the handling of $x_4 \simeq g(x_5)$. Its entailment does not relate with the other literals in L and it can be solved by an early application of ASSIGN.

$$\dfrac{\top \Vdash_E \underline{h(x_1) \simeq h(c)} \wedge h(x_2) \not\simeq g(x_3) \wedge f(x_1) \simeq f(x_3)}{\mathcal{A} \qquad\qquad \mathcal{B}}\ \textsc{Ucomp}$$

Since $h(x_1) \simeq h(c)$ leads to the constraints $x_1 \simeq c \vee x_1 \simeq a$ and a subsequent splitting of the derivation, $\mathcal{A}$ is

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\top \Vdash_E \underline{x_1 \simeq c} \wedge h(x_2) \not\simeq g(x_3) \wedge f(x_1) \simeq f(x_3)}{\{x_1 \simeq c\} \Vdash_E h(x_2) \not\simeq g(x_3) \wedge \underline{f(c) \simeq f(x_3)}}\ \textsc{Assign}}{\{x_1 \simeq c\} \Vdash_E h(x_2) \not\simeq g(x_3) \wedge \underline{x_3 \simeq c}}\ \textsc{Ucomp}}{\{x_1 \simeq c, x_3 \simeq c\} \Vdash_E h(x_2) \not\simeq g(c)}\ \textsc{Assign}}{\{x_1 \simeq c, x_3 \simeq c\} \Vdash_E \bot}\ \textsc{Dgen}}{\{x_1 \simeq c, x_3 \simeq c\} \Vdash_E \bot}\ \textsc{Fail}$$

and $\mathcal{B}$ is

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\top \Vdash_E \underline{x_1 \simeq a} \wedge h(x_2) \not\simeq g(x_3) \wedge f(x_1) \simeq f(x_3)}{\{x_1 \simeq a\} \Vdash_E h(x_2) \not\simeq g(x_3) \wedge \underline{f(a) \simeq f(x_3)}}\ \textsc{Assign}}{\{x_1 \simeq a\} \Vdash_E h(x_2) \not\simeq g(x_3) \wedge \underline{x_3 \simeq a}}\ \textsc{Assign}}{\{x_1 \simeq a, x_3 \simeq a\} \Vdash_E h(x_2) \not\simeq g(a)}\ \textsc{Dgen}}{\{x_1 \simeq a, x_3 \simeq a\} \Vdash_E \bot}\ \textsc{Fail}}{\{x_1 \simeq a, x_3 \simeq a\} \Vdash_E \bot}\ \textsc{Ucomp}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\top \Vdash_E \underline{x_1 \simeq a} \wedge h(x_2) \not\simeq g(x_3) \wedge f(x_1) \simeq f(x_3)}{\{x_1 \simeq a\} \Vdash_E h(x_2) \not\simeq g(x_3) \wedge \underline{f(a) \simeq f(x_3)}}\ \textsc{Assign}}{\{x_1 \simeq a\} \Vdash_E h(x_2) \not\simeq g(x_3) \wedge \underline{x_3 \simeq b}}\ \textsc{Assign}}{\{x_1 \simeq a, x_3 \simeq b\} \Vdash_E h(x_2) \not\simeq g(b)}\ \textsc{Dgen}}{\mathcal{C}_1 \qquad\qquad \mathcal{C}_2}\ \textsc{Ucomp}$$

with $\mathcal{C}_1$ and $\mathcal{C}_2$ being the following derivations split from the disjunction $x_2 \simeq a \vee x_2 \simeq c$ derived from $h(x_2) \simeq g(b)$:

$$\dfrac{\dfrac{\{x_1 \simeq a,\, x_3 \simeq b\} \Vdash_E x_2 \simeq a}{\{x_1 \simeq a,\, x_2 \simeq a,\, x_3 \simeq b\} \Vdash_E \top}\,\textsc{Assign}}{\{x_1 \simeq a,\, x_2 \simeq a,\, x_3 \simeq b\} \Vdash_E \top}\,\textsc{Yield} \qquad \dfrac{\dfrac{\{x_1 \simeq a,\, x_3 \simeq b\} \Vdash_E x_2 \simeq c}{\{x_1 \simeq a\, x_2 \simeq c,\, x_3 \simeq b\} \Vdash_E \top}\,\textsc{Assign}}{\{x_1 \simeq a,\, x_2 \simeq c,\, x_3 \simeq b\} \Vdash_E \top}\,\textsc{Yield}$$

A solution $E_\sigma = \{x_1 \simeq a,\, x_2 \simeq a,\, x_3 \simeq b,\, x_4 \simeq g(x_5)\}$ such that $E \cup E_\sigma \models L$ holds is produced by $\mathcal{C}_1$, corresponding to the same E_σ respecting the entailment conditions shown in Example 4.5. Note that $\mathcal{C}_2$ also provides a solution, differing from E_σ only on the assignment to x_2.

4.2.2 Correctness

Theorem 4.5 (Termination)**.** *All derivations in* CCFV *are finite.*

Proof. The Split rule is the only one to increase the width of a derivation tree. Its application is bounded by the size of the disjunctions, which in turn is bound by the number of signature classes in E^{CC}. Therefore Split can only be applied a finite number of times. It then suffices to show that the depth of the tree is also bounded. For simplicity, but without any fundamental effect on the proof, let us assume that all rules but Split are applied on conjunctions. Let $d(C)$ be the sum of the depths of all occurrences of variables in the literals of the conjunction C. The Assign rule decreases the number of variables of C. The Fail rule closes a branch. The Yield removes constraints, of which there are only finitely many. All remaining rules decrease d in the transition from $E_\sigma \Vdash_E C$ to $E'_\sigma \Vdash_E C'_1 \vee \cdots \vee C'_n$, i.e. $d(C) > d(C'_1), \ldots, d(C) > d(C'_n)$. At each node, $d(C)$ or the number of variables in C are decreasing, except at the Split, Fail or Yield steps. Since no branch can contain infinite sequences of these applications, the depth of the any derivation tree is always finite. □

Lemma 4.6 (Solutions produce ground substitutions)**.** *Given a computed solution E_σ for an E-ground (dis)unification problem $E \models L\sigma$, each $\sigma_g \in \textsc{Sols}(E_\sigma)$ is an acyclic substitution such that* $\operatorname{ran}(\sigma_g) \subseteq \mathbf{T}(E \cup L)$ *and $\sigma_g^\star$ is ground.*

Proof. The Assign rule is the only one that augments E_σ, adding a term from $\mathbf{T}(L)$ to the congruence class of a variable in E_σ^{CC}. The occurs check in the side condition, together with the application of *rep* after the assignment on the remaining constraints in the branch, ensures that a variable is not assigned to a term in which it occurs. Therefore, by induction on the structure of derivation trees, the substitution $\sigma = \{x \mapsto \mathit{rep}(E_\sigma, x) \mid x \in \operatorname{FV}(L)\}$ built from E_σ is acyclic and $\mathit{ran}(\sigma) \subseteq \mathbf{T}(E \cup L)$. Thus each

$$\sigma_g = \left\{ x \mapsto t_{\mathit{rep}(E_\sigma, x)} \;\middle|\; \begin{array}{l} x \in \operatorname{FV}(L),\, t_{\mathit{rep}(E_\sigma, x)} \text{ is } \mathit{rep}(E_\sigma, x) \text{ if } \mathit{rep}(E_\sigma, x) \text{ is non-variable or} \\ \text{some chosen ground } t \in \mathbf{T}(E \cup L) \text{ otherwise} \end{array} \right\}$$

is also acyclic, $\sigma_g^\star$ is ground, and $\mathit{ran}(\sigma_g) \subseteq \mathbf{T}(E \cup L)$. □

Lemma 4.7 (Rules capture entailment conditions)**.** *There is an applicable rule in the calculus for any arbitrary equality literal ℓ.*

Proof. If ℓ is ground, then Yield or Fail are applicable, according to whether it is entailed by E. We otherwise proceed by case analysis on the structure of the literal, according to the cases in Theorem 4.4.

Case 1, $\ell = x \simeq y$: If $x = y$, the rule Yield is applicable, otherwise it is Assign.

Case 2, $\ell = x \simeq f(s_1, \ldots, s_n)$, and x occurs in $f(s_1, \ldots, s_n)$: The rule Uvar is the only applicable one.

Case 3, $\ell = x \simeq f(s_1, \ldots, s_n)$ and x does not occur in $f(s_1, \ldots, s_n)$: the only rule applicable is Assign.

Case 4, $\ell = f(u_1, \ldots, u_n) \simeq g(v_1, \ldots, v_n)$: if $f = g$ the rule Ucomp is applied, otherwise Ugen is.

Case 5, $\ell = u \not\simeq v$: The rules Dvar, Dfapp, or Dgen are applied depending on the structure of u and v. Since a term is either a variable or a function application, all options are considered. □

Lemma 4.8 (Rules preserve entailment conditions)**.** *For each rule*

$$\frac{E_\sigma \Vdash_E C}{E'_\sigma \Vdash_E C'} \text{ R}$$

if there is a ground substitution ρ such that $\mathrm{FV}(E_\sigma) \cap \mathrm{FV}(E_\rho) = \varnothing$ and $E \cup E_\sigma \cup E_\rho \models C$, then there is a ground substitution ρ' with $\mathrm{FV}(E'_\sigma) \cap \mathrm{FV}(E'_\rho) = \varnothing$ such that

(i) $E \cup E_\sigma \cup E_\rho \models C'$ *and*

(ii) $E \cup E'_\sigma \cup E'_\rho \models C$

Proof. The second condition holds trivially for all rules but Assign, since it is the only one that modifies E_σ. We then first analyze Assign for both conditions, starting with the second one. Let $C = C_1 \wedge x \simeq s$. Therefore $E'_\sigma = E_\sigma \cup \{x \simeq s\}$. C' is the result of replacing x, and s if it is also a variable, in C_1 by its representative, which is either s or a variable from $[x]_{E'_\sigma}$. Let E'_ρ be E_ρ but without the equality for the variable which disappeared from C_1 in C'. Thus, since $E \cup E_\sigma \cup E_\rho \models C$ and $E'_\sigma \cup E'_\rho$ effectively have the same congruence closure as $E_\sigma \cup E_\rho$, it follows that $E \cup E'_\sigma \cup E'_\rho \models C$. For the first condition, since C' coincides with C_1 but for the replacement of the assigned variable, it also follows from the premise that $E \cup E_\sigma \cup E_\rho \models C'$.

We verify the first condition for the remaining rules by case analysis.

Case Split: Due to entailing a disjunction of the considered constraints guaranteeing the entailment of at least one of the disjuncts.

Case Yield: Follows trivially from the properties of conjunction.

Case Uvar: By congruence. Since x occurs in $f(\bar{u}_n)$, $E \cup E_\sigma \cup E_\rho \models x \simeq f(\bar{u}_n)$ implies that there are terms t and $f(\bar{t}'_n)$ in $\mathbf{T}(E)$ which are the representatives, in $E_\sigma \cup E_\rho$, for x and $f(\bar{u}_n)$,

respectively, such that $E \models t \simeq f(\bar{t}_n)$ and t occurs in $f(\bar{t}_n)$ modulo E. Thus in C' there is one disjunct such that

$$x \simeq t \wedge u_1 \simeq t_1 \wedge \cdots u_n \simeq t_n \wedge C''$$

which is built according to the class $[t]_E$ and is trivially entailed by E, since $E \cup E_\sigma \cup E_\rho$ entailing each of its conjuncts follows from the construction of the disjunct and from the premise.

Proceeding analogously the condition can be shown to also hold for the remaining branching rules. □

Theorem 4.9 (Completeness)**.** *Let σ be a solution for an E-ground (dis)unification problem $E \models L\sigma$. Then there exists a derivation tree starting on $\varnothing \Vdash_E L$ with at least one branch closed with* Yield *such that $\sigma_g \in \textsc{Sols}(E_\sigma)$ and $E \models L\sigma_g^\star$.*

Proof. Theorem 4.2 ensures that there is an acyclic substitution σ_g corresponding to σ such that $ran(\sigma_g) \subseteq \mathbf{T}(E \cup L)$, $\sigma_g^\star$ is ground and $E \models L\sigma_g^\star$. It remains to be shown that CCFV can produce such a substitution. By Lemma 4.7, for any equality literal in L, no matter its shape, a given rule is applicable to assess its entailment. By Lemma 4.8, all rules in CCFV preserve the entailment conditions according to ground substitutions, therefore, by induction on the structure of derivation trees, if there is a solution for L, then there is a branch in the derivation tree starting from $\varnothing \Vdash_E L$ whose leaf is $E_\sigma \Vdash_E \top$ and $\sigma_g \in \textsc{Sols}(E_\sigma)$ is such that $E \models L\sigma_g^\star$. □

Theorem 4.10 (Soundness)**.** *Whenever a branch is closed with* Yield*, every $\sigma_g \in \textsc{Sols}(E_\sigma)$ is such that $E \models L\sigma_g^\star$.*

Proof. We show by induction on the structure of derivation trees that a resulting ground substitution σ from a branch closed with Yield is indeed a solution for the initial problem at the root. The base case is that at the leaf any substitution $\sigma_g \in \textsc{Sols}(E_\sigma)$ is ground and such that $E \models \top\sigma_g^*$. Lemma 4.6 ensures that $\sigma_g^\star$ is ground and the entailment holds trivially. Our induction hypothesis is that if the condition holds at the succedent $E'_\sigma \Vdash_E C'$ of a rule it holds that $E \models C'\sigma$, then it holds for premise $E_\sigma \Vdash_E C$ that $E \models C\sigma$. We proceed by case analysis, for each rule.

Case Split: Follows by entailing a disjunct implying the entailment of the disjunction.

Case Assign: Let $C = C_1 \cup \{x \simeq s\}$. Again, C' is the result of replacing a variable in C_1 according to the assignment $x \simeq s$ being added to the current partial solution. Since σ stems from a successful branch deriving from C', it maps x and s to a ground term congruent to the representative they were replaced by in C_1. Therefore, from $E \models C'\sigma$ it follows that $E \models C_1\sigma$. Moreover, trivially $E \models \{x\sigma \simeq s\sigma\}$. Thus $E \models C\sigma$.

Case Uvar: From the premise and by the properties of disjunction, there is at least one disjunct in $C'\sigma$ such that, for some $[t] \in E^{\text{CC}}$ and $f(t_1, \ldots, t_n) \in [t]$,

$$E \models x\sigma \simeq t\sigma \wedge u_1\sigma \simeq t_1\sigma \wedge \cdots \wedge u_n\sigma \simeq t_n\sigma$$

By congruence, $E \models f(\bar{u}_n)\sigma \simeq f(\bar{t}_n)\sigma$ holds. Since we have that $E \models t \simeq f(\bar{t}_n)$, $E \models C'\sigma$ and $C\sigma = C'\sigma \cup \{x \simeq f(\bar{u}_n)\}\sigma$, it follows that $E \models C\sigma$.

Proceeding analogously the condition can be shown to also hold for the remaining branching rules. Since the induction hypothesis is valid, the condition $E \models (C \wedge E_\sigma)\sigma^\star_g$ also holds at the root, in which C is L and E_σ is the empty set. Thus $E \models L\sigma^\star_g$. □

Corollary 4.11 (CCFV decides E-ground (dis)unification)**.** *Any derivation strategy based on the* CCFV *calculus is a decision procedure for E-ground (dis)unification.*

Even though CCFV always finds a solution for an E-ground (dis)unification problem if it has solutions, it only provides the ground solutions σ^* such that $E \models L\sigma^*$, σ is acyclic and $ran(\sigma) \subseteq \mathbf{T}(E \cup L)$.

4.2.3 Instantiating with CCFV

We here show how to use CCFV for computing substitutions for the instantiation techniques cast with E-ground (dis)unification, as described in Section 4.1.1. We also present how CCFV facilitates certain extensions to these techniques.

Trigger based instantiation The restricted E-ground (dis)unification

$$E \models (u_1 \simeq y_1 \wedge \cdots \wedge u_n \simeq y_n)\,\sigma$$

for a given quantified formula in $\mathcal{Q}$ can be solved with CCFV by adding a side condition to Assign that s must be a ground term and removing the side condition of Uvar. This will lead to the application of Uvar in each $u_i \simeq y_i$. The desired instantiations are obtained by considering the substitutions extracted from the solutions from branches closed with Yield, ignoring the assignments for $\bar{y}_n$.

Discarding entailed instances Trigger based instantiation may produce instances which are already entailed by the ground model. Such instances most probably will not contribute to the solving, so they should be discarded. Checking this, however, is not straightforward with processing techniques. CCFV, on the other hand, allows it by simply checking, given an instantiation σ for a quantified formula $\forall \bar{x}_n.\ \psi$, whether there is a literal $\ell \in \psi$ such that $E \cup E_\sigma \models \ell$, with $E_\sigma = \{x \simeq x\sigma \mid x \in dom(\sigma)\}$. This can be checked straightforwardly by determining whether Yield is applicable in a state $E_\sigma \Vdash_E rep(E_\sigma, \ell)$.

Example 4.8. Let $E \cup \mathcal{Q}$ be such that $E \models p(c, d) \simeq \top$ and $\mathcal{Q}$ has a quantified formula $\forall xy.\ \psi$ containing a literal $p(x, y)$. If an instantiation $\sigma = \{x \mapsto c, y \mapsto d\}$ is derived for $\forall xy.\ \psi$, it can be quickly determined that, since $E \models p(x, y)\sigma \simeq \top$, this instantiation should be discarded.

Conflict based instantiation CCFV can be applied directly in the search for conflicting instances for $E \cup \mathcal{Q}$. The negation of each quantified formula in $\mathcal{Q}$ comprises a conjunction L of equality literals for which a solution within the context E is searched for, starting on the initial state $\varnothing \Vdash_E L$. Differently from the algorithm given by Reynolds et al. [RTdM14], CCFV finds all conflicting instantiations for a given quantified formula by being a decision procedure for the corresponding E-ground (dis)unification problem $E \models L\sigma$.

Propagating equalities As discussed in Section 3.2.2, even when the search for conflicting instances fails it is still possible to propagate relevant equalities. Let us consider how to propagate these equalities with CCFV. Given some $\neg\psi = \ell_1 \wedge \cdots \wedge \ell_n$, let σ be a ground substitution such that $E \models \ell_1\sigma \wedge \cdots \wedge \ell_{k-1}\sigma$ and all remaining literals $\ell_k\sigma, \ldots, \ell_n\sigma$ not entailed are ground disequalities with $(\mathbf{T}(\ell_k\sigma) \cup \cdots \cup \mathbf{T}(\ell_n\sigma)) \subseteq \mathbf{T}(E)$. The instantiation $\forall \bar{x}_n.\, \psi \to \psi\sigma$ introduces a disjunction of equalities constraining $\mathbf{T}(E)$. CCFV can generate such propagating substitutions if the side conditions of Fail and Yield are relaxed w.r.t. ground disequalities whose terms occur in $\mathbf{T}(E)$ and originally had variables: the former is not applied based on them and the latter is.

Example 4.9. Consider the conjunctive set $E = \{f(a) \simeq t,\ t' \simeq g(a)\}$ and a quantified formula $\forall x.\ f(x) \not\simeq t \vee f(x) \simeq g(x)$. When applying CCFV in the problem

$$E \models (f(x) \simeq t \wedge f(x) \not\simeq g(x))\,\sigma$$

to entail the first literal a candidate solution $E_\sigma = \{x \simeq a\}$ is produced. The second literal would then be normalized to $f(a) \not\simeq g(a)$, which would lead to the application of Fail, since it is not entailed by E. However, as it is a disequality whose terms are in $\mathbf{T}(E)$ and originally had variables, instead the rule Yield is applied. The resulting substitution $\sigma = \{x \mapsto a\}$ leads to propagating the equality $f(a) \simeq g(a)$, which merges two classes previously different in E^{CC}.

MBQI CCFV can be applied directly in the search for conflicting instances for $E_{\text{TOT}} \cup \mathcal{Q}$. However, it is not necessary to explicitly build E_{TOT}, which can be quite large. The search can be performed on E and when a solution is not found, "definitions" for new terms can be added lazily to E_{TOT} as they are required by CCFV for building potential solutions. Differently from the model based approaches of Ge and de Moura [GdM09] and Reynolds et al. [RTG+13], which allow integration of theories beyond equality, CCFV for now only handles the equational case.

4.3 A non-backtracking CCFV

The calculus in Table 4.1 is inherently backtracking because it considers one solution at a time. This is the case since conjunctive constraints can only be solved by building solutions extending a current one. In this section we present an alternative calculus for CCFV that searches for

solutions in a non-backtracking manner. Sets of solutions may be computed independently and combined according to the dependency between the solved constraints. This way one is not restricted to search only for solutions extending a fixed one.

This new calculus is similar to the previous one, but now a state is an AND-OR combination of *solving statements*, denoted $\mathfrak{S}$. Each solving statement has the form $\Sigma \Vdash_E C$, in which Σ is a set of conjunctive equalities representing partial solutions and C, as before, is an AND-OR combination of the remaining constraints to entail. The calculus is shown in Table 4.2. The symbols $\sqcap$, $\sqcup$ and $\wedge$, $\vee$ are used for writing combinations of solving statements and constraints, respectively. Instead of a derivation tree, the new calculus operates on a single state that is modified in a top-down manner by each rule application. From an initial state $\{\top\} \Vdash_E L$, a fair application of the rules eventually derives either $\varnothing \Vdash_E \bot$ or $\Sigma \Vdash_E \top$. In the latter case all solutions for the E-ground (dis)unification problem $E \models L\sigma$ are contained in Σ. The former case indicates that the problem has no solutions.

The main differences in the calculi come from the structural rules. The branching behave mostly as before. Assign changes only so that now it updates not a single solution but a set of them, keeping the remaining constraints normalised according to all of them. While before disjunctions in the right hand side were abstracted with branching, now the rules explicitly operate on them. Moreover, conjunctions may also be split. The rules $\textsc{Split}_\wedge$ and $\textsc{Split}_\vee$ split the search for solutions into simultaneous or orthogonal components, respectively, as represented by the combination of solving statements with $\sqcap$ or $\sqcup$. On the other hand, Meet and Join combine components based on their interdependency. Orthogonal components can be combined only after they have been solved. The rules from Table 4.1 that simplified conjunctive constraints based on entailment from E are extended for simplifying disjunctive constraints. Therefore $\textsc{Fail}_\wedge$ and $\textsc{Yield}_\wedge$ coincide with the previous calculus, while $\textsc{Fail}_\vee$ and $\textsc{Yield}_\vee$ apply the dual simplifications for disjunctions. These extended structural rules allow each part of the problem to be solved as independently as desired and the resulting solutions to be later properly combined.

Combining orthogonal solutions, done by Join, is straightforward. It suffices to merge the sets of complete solutions and recover the disjunction in the constraints. The case of simultaneous solutions, however, considers arbitrary constraints and partial solutions and requires closer attention. From two solving statements, Meet generates all possible pairwise combinations of solutions from their respective solution sets. To ensure compatibility, one of the two solutions being combined is chosen and both the remaining constraints and the other solution are normalized according to it. This way if two solutions are not compatible, at least one literal that cannot be entailed will be in the constraints of the generated state. Since the constraints are conjunctive, the faulty statement will eventually be discarded. The normalisation also keeps the invariant that variables which are not their own representatives do not appear in the right hand side of the solving statement.

$$\frac{\mathfrak{S} \sqcap \Sigma \Vdash_E x \simeq s \wedge C}{\mathfrak{S} \sqcap \{E_\sigma \wedge x \simeq s \mid E_\sigma \in \Sigma\} \Vdash_E \mathit{rep}(x \simeq s,\, C)}\ \textsc{Assign} \quad \text{if } x \notin \mathrm{FV}(s)$$

$$\frac{\mathfrak{S} \sqcap \Sigma \Vdash_E x \simeq f(\bar{u}_n) \wedge C}{\mathfrak{S} \sqcap \Sigma \Vdash_E \bigvee_{[t] \in E^{\mathrm{cc}},\, f(\bar{t}_n) \in [t]} (x \simeq t \wedge u_1 \simeq t_1 \wedge \cdots \wedge u_n \simeq t_n \wedge C)}\ \textsc{Uvar} \quad \text{if } x \in \mathrm{FV}(f(\bar{u}_n))$$

$$\frac{\mathfrak{S} \sqcap \Sigma \Vdash_E f(\bar{u}_n) \simeq f(\bar{s}_n) \wedge C}{\mathfrak{S} \sqcap \Sigma \Vdash_E \begin{array}{l}(u_1 \simeq s_1 \wedge \cdots \wedge u_n \simeq s_n \wedge C) \vee \\ \quad \bigvee_{\substack{[t] \in E^{\mathrm{cc}}, \\ f(\bar{t}_n) \in [t],\, f(\bar{t}'_n) \in [t]}} \left(\begin{array}{l} u_1 \simeq t_1 \wedge \cdots \wedge u_n \simeq t_n \wedge \\ s_1 \simeq t'_1 \wedge \cdots \wedge s_n \simeq t'_n \wedge C \end{array} \right)\end{array}}\ \textsc{Ucomp}$$

$$\frac{\mathfrak{S} \sqcap \Sigma \Vdash_E f(\bar{u}_n) \simeq g(\bar{s}_m) \wedge C}{\mathfrak{S} \sqcap \Sigma \Vdash_E \bigvee_{\substack{[t] \in E^{\mathrm{cc}}, \\ f(\bar{t}_n) \in [t],\, g(\bar{t}'_m) \in [t]}} \left(\begin{array}{l} u_1 \simeq t_1 \wedge \cdots \wedge u_n \simeq t_n \wedge \\ s_1 \simeq t'_1 \wedge \cdots \wedge s_m \simeq t'_m \wedge C \end{array} \right)}\ \textsc{Ugen} \quad \text{if } f \neq g$$

$$\frac{\mathfrak{S} \sqcap \Sigma \Vdash_E x \not\simeq y \wedge C}{\mathfrak{S} \sqcap \Sigma \Vdash_E \bigvee_{[t],\, [t'] \in E^{\mathrm{cc}},\, E \models t \not\simeq t'} (x \simeq t \wedge y \simeq t' \wedge C)}\ \textsc{Dvar}$$

$$\frac{\mathfrak{S} \sqcap \Sigma \Vdash_E x \not\simeq f(\bar{s}_n) \wedge C}{\mathfrak{S} \sqcap \Sigma \Vdash_E \bigvee_{\substack{[t],\, [t'] \in E^{\mathrm{cc}}, \\ E \models t \not\simeq t',\, f(\bar{t}'_n) \in [t']}} (x \simeq t \wedge s_1 \simeq t'_1 \wedge \cdots \wedge s_n \simeq t'_n \wedge C)}\ \textsc{Dfapp}$$

$$\frac{\mathfrak{S} \sqcap \Sigma \Vdash_E f(\bar{u}_n) \not\simeq g(\bar{s}_m) \wedge C}{\mathfrak{S} \sqcap \Sigma \Vdash_E \bigvee_{\substack{[t],\, [t'] \in E^{\mathrm{cc}},\, E \models t \not\simeq t', \\ f(\bar{t}_n) \in [t],\, g(\bar{t}'_m) \in [t']}} \left(\begin{array}{l} u_1 \simeq t_1 \wedge \cdots \wedge u_n \simeq t_n \wedge \\ s_1 \simeq t'_1 \wedge \cdots \wedge s_m \simeq t'_m \wedge C \end{array} \right)}\ \textsc{Dgen}$$

$$\frac{\mathfrak{S} \sqcap \Sigma \Vdash_E C_1 \wedge C_2}{\mathfrak{S} \sqcap (\Sigma \Vdash_E C_1) \sqcap (\Sigma \Vdash_E C_2)}\ \textsc{Split}_\wedge \qquad \frac{\mathfrak{S} \sqcap \Sigma \Vdash_E C_1 \vee C_2}{\mathfrak{S} \sqcap (\Sigma \Vdash_E C_1) \sqcup (\Sigma \Vdash_E C_2)}\ \textsc{Split}_\vee$$

$$\frac{\mathfrak{S} \sqcap (\Sigma \Vdash_E C \sqcap \Sigma' \Vdash_E C')}{\mathfrak{S} \sqcap \left(\bigsqcup_{E_\sigma \in \Sigma,\, E'_\sigma \in \Sigma'} \left(\{E_\sigma\} \Vdash_E C \wedge \mathit{rep}(E_\sigma,\, E'_\sigma) \wedge \mathit{rep}(E_\sigma,\, C') \right) \right)}\ \textsc{Meet}$$

$$\frac{\mathfrak{S} \sqcap (\Sigma \Vdash_E c_1 \sqcup \Sigma' \Vdash_E c_2)}{\mathfrak{S} \sqcap \Sigma \cup \Sigma' \Vdash_E c_1 \vee c_2}\ \textsc{Join} \quad \text{with } c_1,\, c_2 \in \{\bot,\, \top\}$$

$$\frac{\mathfrak{S} \sqcap \Sigma \Vdash_E \ell \wedge C}{\mathfrak{S} \sqcap \varnothing \Vdash_E \bot}\ \textsc{Fail}_\wedge \quad \text{if } \ell \text{ is ground and } E \not\models \ell \qquad \frac{\mathfrak{S} \sqcap \Sigma \Vdash_E \ell \wedge C}{\mathfrak{S} \sqcap \Sigma \Vdash_E C}\ \textsc{Yield}_\wedge \quad \text{if } E \models \ell$$

$$\frac{\mathfrak{S} \sqcap \Sigma \Vdash_E \ell \vee C}{\mathfrak{S} \sqcap \Sigma \Vdash_E C}\ \textsc{Fail}_\vee \quad \text{if } \ell \text{ is ground and } E \not\models \ell \qquad \frac{\mathfrak{S} \sqcap \Sigma \Vdash_E \ell \vee C}{\mathfrak{S} \sqcap \Sigma \Vdash_E \top}\ \textsc{Yield}_\vee \quad \text{if } E \models \ell$$

Table 4.2: Non-backtracking CCFV calculus. E is fixed from a problem $E \models L\sigma$.

Example 4.10. Consider the application of MEET on the state

$$\{\{x \simeq f(y)\}\} \Vdash_E g(z) \simeq t \sqcap \{\{x \simeq y,\, z \simeq t\}\} \Vdash_E \top$$

Since both sets of solutions are singletons, there is only one case to consider. Let $\{x \simeq f(y)\}$ be the solution chosen for the normalisation. The representatives of the variable x, y, z in it are $f(y)$, y, and z, respectively. Thus the resulting state is

$$\{\{x \simeq f(y)\}\} \Vdash_E g(z) \simeq t \wedge f(y) \simeq y \wedge z \simeq t \wedge \top$$

with the normalisation of the solution and the constraint from the combined state. This shows that the two solutions are compatible only if $f(y) \simeq y$ can be solved, since x was assigned in both solutions.

4.3.1 Strategy

A simple derivation strategy for this new CCFV calculus is to decompose the entailment problems until they depend only on variable assignments and ground reasoning, and then combine the found solutions accordingly. Given an initial state $\{\top\} \Vdash_E L$, branching rules and splitting rules are applied exhaustively, until all solving statements have the form $\{\top\} \Vdash_E A \wedge C$, in which A and C are conjunctive sets of variable equations and of ground constraints, respectively. By applying a series of ASSIGN and $\text{YIELD}_\wedge$ or $\text{FAIL}_\wedge$ rules, each solving statement will become either $\Sigma \Vdash_E \top$, if the assignments in A are compatible and E entails C, or $\varnothing \Vdash_E \bot$, otherwise. The resulting solving statements are successively combined and simplified with series of applications of JOIN and MEET, and both cases of YIELD and FAIL. One obtains either a state $\varnothing \Vdash_E \bot$, if the solutions are not compatible, or a state $\Sigma' \Vdash_E \top$, otherwise. In the latter case all solutions in Σ' are solutions for the E-ground (dis)unification problem.

Example 4.11. Consider again E, L, and E^{CC} as in Example 4.7:

$$\begin{aligned} E &= \{f(a) \simeq f(b),\, h(a) \simeq h(c),\, g(b) \not\simeq h(c)\} \\ L &= \{h(x_1) \simeq h(c),\, h(x_2) \not\simeq g(x_3),\, f(x_1) \simeq f(x_3),\, x_4 \simeq g(x_5)\} \\ E^{\text{CC}} &= \{\{a\},\, \{b\},\, \{c\},\, \{f(a), f(b)\},\, \{h(a), h(c)\},\, \{g(b)\}\} \end{aligned}$$

The derivation on the respective E-ground (dis)unification problem, with the above strategy, is shown below. As before, the solving of $x_4 \simeq g(x_5)$ is kept implicit to simplify the presentation. Steps with double lines represent numerous applications of the indicated rules, e.g.:

$$\frac{\{\top\} \Vdash_E f(x_1) \simeq f(x_3) \wedge h(x_2) \not\simeq g(x_3) \wedge h(x_1) \simeq h(c)}{\{\top\} \Vdash_E f(x_1) \simeq f(x_3) \sqcap \{\top\} \Vdash_E h(x_2) \not\simeq g(x_3) \sqcap \{\top\} \Vdash_E h(x_1) \simeq h(c)}\ \text{SPLIT}^*_\wedge$$

We present the derivation on each solving statement independently. The statement for the first constraint, $\{\top\} \Vdash_E f(x_1) \simeq f(x_3)$, is reduced in the following manner:

$$\dfrac{\dfrac{\dfrac{\dfrac{\{\top\} \Vdash_E f(x_1) \simeq f(x_3)}{\{\top\} \Vdash_E x_1 \simeq x_3 \vee (x_1 \simeq a \wedge x_3 \simeq b) \vee (x_1 \simeq b \wedge x_3 \simeq a)}\,\textsc{Ucomp}}{\{\top\} \Vdash_E x_1 \simeq x_3 \sqcup \{\top\} \Vdash_E x_1 \simeq a \wedge x_3 \simeq b \sqcup \{\top\} \Vdash_E x_1 \simeq b \wedge x_3 \simeq a}\,\textsc{Split}^*_\vee}{\{\{x_1 \simeq x_3\}\} \Vdash_E \top \sqcup \{\{x_1 \simeq a,\, x_3 \simeq b\}\} \Vdash_E \top \sqcup \{\{x_1 \simeq b,\, x_3 \simeq a\}\} \Vdash_E \top}\,\textsc{Assign}^*}{\{\{x_1 \simeq x_3\},\, \{x_1 \simeq a,\, x_3 \simeq b\},\, \{x_1 \simeq b,\, x_3 \simeq a\}\} \Vdash_E \top}\,\textsc{Join}^*,\, \textsc{Yield}_\vee$$

while $\{\top\} \Vdash_E h(x_2) \not\simeq g(x_3)$ is reduced as

$$\dfrac{\dfrac{\dfrac{\dfrac{\{\top\} \Vdash_E h(x_2) \not\simeq g(x_3)}{\{\top\} \Vdash_E (x_2 \simeq c \wedge x_3 \simeq b) \vee (x_2 \simeq a \wedge x_3 \simeq b)}\,\textsc{Dgen}}{\{\top\} \Vdash_E x_2 \simeq c \wedge x_3 \simeq b \sqcup \{\top\} \Vdash_E x_2 \simeq a \wedge x_3 \simeq b}\,\textsc{Split}_\vee}{\{\{x_2 \simeq c,\, x_3 \simeq b\}\} \Vdash_E \top \sqcup \{\{x_2 \simeq a,\, x_3 \simeq b\}\} \Vdash_E \top}\,\textsc{Assign}^*}{\{\{x_2 \simeq c,\, x_3 \simeq b\},\, \{x_2 \simeq a,\, x_3 \simeq b\}\} \Vdash_E \top}\,\textsc{Join}^*$$

and $\{\top\} \Vdash_E h(x_1) \simeq h(c)$ as

$$\dfrac{\dfrac{\dfrac{\dfrac{\{\top\} \Vdash_E h(x_1) \simeq h(c)}{\{\top\} \Vdash_E x_1 \simeq c \vee x_1 \simeq a}\,\textsc{Ucomp}}{\{\top\} \Vdash_E x_1 \simeq c \sqcup \{\top\} \Vdash_E x_1 \simeq a}\,\textsc{Split}_\vee}{\{\{x_1 \simeq c\}\} \Vdash_E \top \sqcup \{x_1 \simeq a\} \Vdash_E \top}\,\textsc{Assign}^*}{\{\{x_1 \simeq c\},\, \{x_1 \simeq a\}\} \Vdash_E \top}\,\textsc{Join}$$

The global derivation is resumed below with the results from the above ones. The remaining steps to perform in the strategy are to combine the interdependent solutions and simplify the resulting constraints. To improve the presentation, let

$$\begin{aligned}
\Sigma_1 &= \{\{x_1 \simeq x_3\},\, \{x_1 \simeq a,\, x_3 \simeq b\},\, \{x_1 \simeq b,\, x_3 \simeq a\}\} \\
\Sigma_2 &= \{\{x_2 \simeq c,\, x_3 \simeq b\},\, \{x_2 \simeq a,\, x_3 \simeq b\}\} \\
\Sigma_3 &= \{\{x_1 \simeq c\},\, \{x_1 \simeq a\}\}
\end{aligned}$$

Moreover, the presentations of the combinations are collapsed. The results from Meet lead to several solving statements to be combined with Join, besides the built constraints being amenable to assignments and conjunctive and disjunctive simplifications. Thus only the final results are depicted for the combinations of Σ_2 with Σ_3 and then of its result with Σ_1.

$$\dfrac{\dfrac{\dfrac{\{\top\} \Vdash_E f(x_1) \simeq f(x_3) \wedge h(x_2) \not\simeq g(x_3) \wedge h(x_1) \simeq h(c)}{\vdots}}{\Sigma_1 \Vdash_E \top \sqcap \Sigma_2 \Vdash_E \top \sqcap \Sigma_3 \Vdash_E \top}}{\Sigma_1 \Vdash_E \top \sqcap \left\{ \begin{array}{l} \{x_1 \simeq c,\, x_2 \simeq c,\, x_3 \simeq b\}, \\ \{x_1 \simeq c,\, x_2 \simeq a,\, x_3 \simeq b\}, \\ \{x_1 \simeq a,\, x_2 \simeq c,\, x_3 \simeq b\}, \\ \{x_1 \simeq a,\, x_2 \simeq a,\, x_3 \simeq b\} \end{array} \right\} \Vdash_E \top}$$

$$\dfrac{\Sigma_1 \Vdash_E \top \sqcap \left\{ \begin{array}{l} \{x_1 \simeq c,\, x_2 \simeq c,\, x_3 \simeq b\}, \\ \{x_1 \simeq c,\, x_2 \simeq a,\, x_3 \simeq b\}, \\ \{x_1 \simeq a,\, x_2 \simeq c,\, x_3 \simeq b\}, \\ \{x_1 \simeq a,\, x_2 \simeq a,\, x_3 \simeq b\} \end{array} \right\} \Vdash_E \top}{\left\{ \begin{array}{l} \{x_1 \simeq a,\, x_2 \simeq c,\, x_3 \simeq b\}, \\ \{x_1 \simeq a,\, x_2 \simeq a,\, x_3 \simeq b\} \end{array} \right\} \Vdash_E \top}$$

which has the same solutions as in Example 4.7.

This strategy can be optimised in numerous ways, such as applying $\textsc{Fail}_\wedge$ as soon as possible and then "propagating" the failure through other statements with Meet. Similarly, applying Meet as soon as a small set of solutions is obtained, which would then limit the number of possibilities to search in the interdependent statements, is another point of improvement. The main advantages of this calculus are in providing more flexibility for applying different search strategies while solving E-ground (dis)unification. Indeed it can emulate the previous calculus with a strategy that never applies the rule $\textsc{Split}_\wedge$, and therefore also not Meet. This ensures that each branch is solved independently and has at most one solution. The last steps in the solving would be to apply a series of Join to gather all found solutions and close the derivation with series of $\textsc{Yield}_\vee$ or $\textsc{Fail}_\vee$ applications.

Chapter 5

Implementation

CCFV has been implemented in the veriT [BdODF09] and CVC4 [BCD+11] SMT solvers. We here describe the specifics of the implementation in veriT. As is common in CDCL($\mathcal{T}$) based solvers, an E-graph data structure is kept internally to represent the set of signature classes E^{CC} and efficiently check ground entailment modulo the theory of equality. This data structure is generated by the congruence closure decision procedure for ground equational first-order logic. The E-graph is the basis for the indexing techniques described below. They are paramount for a practical procedure, allowing fast retrieval of candidates when searching for solutions for a given E-ground (dis)unification problem.

An experimental evaluation measures the impact of optimizations and instantiation techniques based on CCFV in veriT and CVC4. Comparisons are shown with previous versions and with the instantiation based SMT solver Z3 [dMB08b].

Limitations Efficient implementations of the congruence closure decision procedure are currently guaranteed to be closed under entailment only for congruent terms, not for disequal ones. E.g. $g(f(a), h(b)) \not\approx g(f(b), h(a)) \in E$ does not lead to the addition of $a \not\approx b$ to the data structure. A complete implementation of CCFV requires the derivation of all entailed disequalities from E. Therefore for now we implement incomplete versions of the rules Dvar, Dfapp, and Dgen. However, as our experiments show, they are still effective in practice. Another limitation, but less relevant, is that the E-graph allows checking in constant time for equality only for terms known by the data structure. These are generally the ground terms appearing in the original formula. Checking whether e.g. two ground function applications $f(t_1, \ldots, t_n)$, $f(t'_1, \ldots, t'_n)$, that do not occur in the E-graph but may appear when analyzing L, are congruent requires checking whether all arguments, position by position, are congruent. Therefore the operation is worst-case linear in the size of the terms. As before, this does impact significantly the performance of the implementation.

5.1 Indexing

Performing E-ground (dis)unification requires dealing with many terms, which makes the use of efficient indexing techniques for fast retrieval of candidates paramount for efficiency. Besides the indexing, we also discuss how the ground model E is minimised to reduce the number of terms that need to be considered. All operations below rely on terms being represented internally as directed acyclic graphs (DAGs) with maximal sharing, also known as "hash-consing", of common expressions. Therefore terms can be traversed as DAGs and syntactic equality between terms can be checked in constant time.

Top symbol indexing of E^{CC} The set of signature classes E^{CC}, as defined in Chapter 4, is indexed by top symbols. The index consists of entries of the form

$$f \to \begin{cases} f(t_{11}, \ldots, t_{1n}) \\ \vdots \\ f(t_{m1}, \ldots, t_{mn}) \end{cases}$$

in which f is a function symbol of arity n and each function application $f(t_{i1}, \ldots, t_{in})$ is a term occurring in E^{CC}.

We define below several operations based on this index and on the E-graph that are used in the E-ground (dis)unification solving.[4] Let f be a function symbol and s a term:

- *class*(s): yields the set of terms in the signature class of s, i.e. all terms occurring in $[s]$ from E^{CC}. These results can be directly retrieved from the E-graph in constant time.

- *find*(s): yields the representative of *class*(s), i.e. a distinguished term from the signature class of s in E^{CC}. Also directly retrievable from the E-graph in constant time.

  This operation is what allows the efficient checking of whether two ground terms t_1 and t_2 are congruent. If they both occur in the E-graph, it suffices to check whether it holds that *find*(t_1) = *find*(t_2), i.e. whether the DAG representation of both representatives is syntactically equal. Otherwise it is necessary to check if the terms have the same top symbol and if arguments of matching position are congruent.

- *reps*(f): yields all representatives whose respective signature class in E^{CC} contains f-applications. The representatives are collected from the entries in the index for f. This operation is linear on the size of the index result for f.

- *apps*(f, s): yields all f-applications in *class*(s). They are retrieved by traversing the set of terms in *class*(s) and collecting those whose top symbol is f. This operation is linear on the size of *class*(s).

[4] This presentation is inspired by the presentation of the indexing techniques used for E-matching by de Moura and Bjørner [dMB07].

– *apps*(f): yields all f-applications occurring in E^{CC}, which amounts to collecting all terms indexed for f. This operation is constant time, since f is mapped in the index to what corresponds to *apps*(f).

– *diseqs*(s): yields all representatives of terms in E^{CC} disequal to s, according to E. This operation is implemented in constant time by associating, to each signature class, a previously computed ordered list of the representatives of classes containing terms disequal to it.

  This way it is possible to check if two ground terms t_1 and t_2 are disequal by assessing whether *find*(t_1) occurs in *diseqs*(t_2), or vice-versa, depending on which one is smaller. By keeping an ordered list, the occurrence check can be done on the logarithm of the list's size.

  To enumerate all terms in E^{CC} disequal to s it suffices to collect

$$\bigcup_{t \in \mathit{diseqs}(s)} \mathit{class}(t)$$

  which can be done in linear time on the size of *diseqs*(s).

To optimise these operations we use approximated sets, as in the Simplify [DNS05] and Z3 [dMB07] systems. They behave much in the same way as sets, except that membership and overlap tests may return false positives. Each signature class is associated with an approximated set containing the functions symbols which have applications in that class. This technique allows checking if a symbol has applications in a signature class in constant time for a subset of the total set of symbols. If e.g. one is querying with *apps*(f, s) for all f-applications in [s], this result cannot be empty if f is in the approximated set of [s]. These approximated sets are implemented with bit masks: each symbol is assigned an arbitrary bit, and the mask for the class is a vector of bits. The set positions are those for which the respective symbol has applications in the class.

Model Minimisation For efficiency reasons, the E-graph in veriT, as in other CDCL($\mathcal{T}$) solvers, contains not only the terms occurring in the current E but all ground terms from the original input formula. This means that a top symbol index built directly from the E-graph will generally contain more terms than actually appear in E. When using operations such as *apps*(f) more terms will be considered, and therefore generally more instances generated, than it would be the case if one were to consider only the terms in E. This has the side effect of potentially worsening the already explosive nature of trigger based instantiation.

To tackle this issue we have two alternative ways of building the top symbol indexing in veriT. The first way considers the E-graph, as explained above. The second one collects terms directly from E by traversing the currently asserted literals. As with the E-graph, only a single term is kept per signature. Dealing directly with the ground model also has the advantage of allowing its minimisation. Since the SAT solver generally asserts more literals than necessary, the

propositional abstraction of $E \cup \mathcal{Q}$ generally can be pruned while still propositionally entailing the original formula. This minimisation is done by computing a *prime implicant*, a minimal partial assignment. Its computation can be performed in linear time (see e.g. Déharbe et al. [DFLM13]). By having a smaller E one reduces the search space to consider when solving an E-ground (dis)unification problem. Another possibility is to minimise E^{CC}, i.e. complementing E with equalities between its terms until the number of signature classes is as small as possible. This minimisation problem, however, is NP-complete [RTGK13], so we do not attempt it. Moreover, it is not necessary for the effectiveness of the instantiation techniques we implement.

We also clean the CNF overhead: we remove part of the propositional variables introduced by the non-equivalency preserving CNF transformation the formula undergoes. They have the side effect that the prime implicant for the CNF is not necessarily prime for the original formula. We apply the same process as de Moura and Bjørner [dMB07] for *Relevancy*.

Example 5.1 ([dMB07])**.** To illustrate the issue, consider a clause $\ell_1 \vee (\ell_2 \wedge \ell_3)$. Its clausification using a Tseitin [Tse83] style algorithm yields the set of clauses

$$\{\ell_1, \ell_{\text{aux}}\}, \quad \{\ell_{\text{aux}}, \neg\ell_2, \neg\ell_3\}, \quad \{\ell_2, \neg\ell_{\text{aux}}\}, \quad \{\ell_3, \neg\ell_{\text{aux}}\}$$

Now, suppose that ℓ_1 is assigned true. In this case, ℓ_2 and ℓ_3 are clearly irrelevant and truth assignments to ℓ_2 and ℓ_3 need not be used, but this fact is occulted by the Tseitin encoding, which creates a set of clauses.

To determine which literals are relevant one can traverse the original formula using the Boolean valuations derived from the prime implicant. Then only the literals which are required to make the formula true are marked as relevant and kept in the propositional assignment from which $E \cup \mathcal{Q}$ is derived.

5.2 Finding solutions

We present in Figure 5.1 an abstract algorithm to solve E-ground (dis)unification. It implements the backtracking CCFV calculus described in Section 4.2 with the strategy from Section 4.2.1. The same notation conventions for ground and non-ground terms from Section 4 are used. The algorithm takes as input a partial solution and a conjunctive set of constraints to be entailed. It produces the set of all complete solutions for the input constraints which extend the input solution. Of course, this set may be empty if there is no such extension. Thus, an initial call CCFV($\top$, L) yields the set of solutions $\{E^1_\sigma, \ldots, E^n_\sigma\}$, from which all ground solutions σ^*, such that σ is acyclic and $ran(\sigma^*) \subseteq \mathbf{T}(E \cup L)$, are derivable. From now on we ambiguously refer to the algorithm as CCFV, clarifying when necessary whether we mean the algorithm or the calculus.

The set of constraints on which CCFV operates are kept as a stack of pairs of terms to be entailed equal of disequal. The most simple implementation of SEL is to merely pop the

```
function CCFV(E_σ, C) is
  if C = ∅ then                                   // All constraints were discharged, return solution
    return {E_σ}
  c ← SEL(C); C ← C \ {c}; SOLS ← ∅                                          // Select constraint
  match c with
    s ≃ s : return CCFV(E_σ, C)                                              // Syntactic equality
    t1 ≄ t2 :                                                                // Ground disequality
      if t1 ∉ diseqs(t2) then return ∅
      return CCFV(E_σ, C)
    t1 ≃ t2 :                                                                // Ground equality
      if t1, t2 ∈ T(E) then
        if find(t1) = find(t2) then return CCFV(E_σ, C)
      else if t1 = f(s̄_n) and t2 = f(r̄_n) then
        return CCFV(E_σ, C ∪ {s1 ≃ r1, ..., sn ≃ rn})
      return ∅
    x ≃ s :
      if x ∉ FV(s) then                                                      // Assignment
        E'_σ ← E_σ ∪ {x ≃ s};   C ← rep(C)
        return CCFV(E'_σ, C)
      else                                        // s is a non-ground function application
        let s = f(ū_n) in
          for t ∈ reps(f) do                                                 // UVAR
            SOLS ← SOLS ∪ CCFV(E_σ, C ∪ {f(ū_n) ≃ t, x ≃ t})
        return SOLS
    f(ū_n) ≃ t :                                                             // E-matching
      for f(t̄_n) ∈ apps(f, t) do
        SOLS ← SOLS ∪ CCFV(E_σ, C ∪ {u1 ≃ t1, ..., un ≃ tn})
      return SOLS
    f(ū_n) ≃ g(ū'_n) :
      if f = g then                                                          // E-unify arguments
        SOLS ← SOLS ∪ CCFV(E_σ, C ∪ {u1 ≃ u'1, ..., un ≃ u'n})
      for t ∈ reps(f) do                                  // E-matching into same class
        SOLS ← SOLS ∪ CCFV(E_σ, C ∪ {f(ū_n) ≃ t,  g(ū'_n) ≃ t})
      return SOLS
    x ≄ t :
      for t' ∈ diseqs(t) do                                // Assignment into a disequal term
        SOLS ← SOLS ∪ CCFV(E_σ, C ∪ {x ≃ t'})
      return SOLS
    f(ū_n) ≄ t :                                          // E-matching into a disequal term
      for t' ∈ diseqs(t) do
        SOLS ← SOLS ∪ CCFV(E_σ, C ∪ {f(ū_n) ≃ t'})
      return SOLS
    x ≄ f(ū_n) :                                                             // DFAPP
      for t ∈ reps(f) do
        SOLS ← SOLS ∪ CCFV(E_σ, C ∪ {f(ū_n) ≃ t, x ≄ t})
      return SOLS
    f(ū_n) ≄ g(ū'_n) :                                    // E-matching into disequal terms
      for t ∈ reps(f) do
        SOLS ← SOLS ∪ CCFV(E_σ, C ∪ {f(ū_n) ≃ t, g(ū'_n) ≄ t})
      return SOLS
    x ≄ y :                                                                  // DVAR
      for each signature t in E^CC do
        for t' ∈ diseqs(t) do
          SOLS ← SOLS ∪ CCFV(E_σ, C ∪ {x ≃ t, y ≃ t'})
      return SOLS
```

Figure 5.1: Backtracking CCFV algorithm

constraint on the top of the stack. The partial solutions generated from the variable assignments are represented as fixed-sized arrays, depending on how many free variables are on L, of "variable valuations". Each valuation consists of:

- a field for the respective variable;
- a flag to whether that variable is the representative of its congruence class;
- a field for, if the variable is a representative, the ground term or non-ground function application it is equal to; otherwise a pointer to the variable it is equal to, the default being itself.

These solutions are manipulated with a UNION-FIND algorithm with path-compression. The *union* operation is made modulo the congruence closure on the ground terms and the current assignments to the variables in that solution. The operation keeps the invariant that the solution is a consistent set of equalities. CCFV keeps a single global candidate solution that is updated by adding or removing assignments, according to how the search proceeds. An specialised E-graph is used for the terms in L, such that constraints are considered modulo the current solution. The E-graph is updated as the current solution changes, merging classes or backtracking these merges as assignments are added or removed from the solution. When an actual solution is found, i.e. when all constraints have been solved for a given candidate solution, a copy of the found solution is stored in a global accumulator of solutions.

CCFV performs a depth-first search for solutions. Given a partial solution and a conjunctive set of constraints, it selects a given constraint; determines how to handle it according to its structure; augments the current solution, and thus also the remaining constraints, if necessary, or try all possibilities for entailing the respective constraint according to the indexing of E^{CC}; and then proceeds to solve the remaining constraints. Each loop through different unification possibilities effectively introduces "break-points" in the search. The remaining constraints are augmented and the search proceeds, but after that possibility is explored the executing will return to the break-point. All assignments that may have been added to the current solution will have been removed, all modifications to the E-graph backtracked, and the remaining constraints to consider will be as they were before the new ones had been added. This can be seen as a "branch" in the search being closed and a new one being picked to be explored. The solutions obtained from closed branches are stored in the solution accumulator. When all branches have been explored, i.e. all cases in the loop have been considered, the execution returns to the previous break-point, backtracking the solution and the E-graph accordingly, and then proceeds, until they have all been exhausted and the execution halts.

To optimise the search, the algorithm applies more case distinctions than the rules in Table 4.1, such as differentiating between E-matching and E-unification. It also makes use of the operations on the top symbol index and E-graph to reduce the number of terms that need to be considered in the search for solutions.

Example 5.2. Consider again the conjunctive sets E and L as in Example 4.1, i.e.

$$\begin{aligned} E &= \{f(a) \simeq f(b),\, h(a) \simeq h(c),\, g(b) \not\simeq h(c)\} \\ L &= \{h(x_1) \simeq h(c),\, h(x_2) \not\simeq g(x_3),\, f(x_1) \simeq f(x_3),\, x_4 \simeq g(x_5)\} \end{aligned}$$

with the set of signature classes of E being

$$E^{\text{CC}} = \{\{a\},\, \{b\},\, \{c\},\, \{f(a), f(b)\},\, \{h(a), h(c)\},\, \{g(b)\}\}$$

We assume that all the indexing operations from Section 5.1 are available. The execution of the CCFV algorithm would start with CCFV($\top$, L). To ease the presentation, let us follow the selection order used in Example 4.7. First the literal $h(x_1) \simeq h(c)$ is selected from the set of constraints. Its pattern matching leads to an E-matching scenario, in which the arguments of $h(x_1)$ will be E-matched with the arguments of each term in $\mathit{apps}(h, h(c))$, i.e. each h-applications in $\mathit{class}(h(c))$. Let $h(a)$ be the first of these terms. As a consequence, the following call will be with CCFV($\top$, $(L \setminus \{h(x_1) \simeq h(c)\}) \cup \{x_1 \simeq a\}$). By proceeding analogously and following the same selection criteria from Example 4.7, the CCFV algorithm will not only find the same solutions (which, is worth noting, would all be found, independently from the selection strategy) but create "break-points", i.e. recursive calls to CCFV from within loops, in the same points that the derivation tree branches, in the same order.

Ordering constraints The CCFV algorithm, as the calculus it implements, allows any arbitrary order for selecting literals in constraints. A possible selection heuristic is to order constraints according to their "branching potential": select first the constraints whose entailment check requires less branching. This number depends on the structure of the terms in the constraint literal and on E^{CC}, and can be approximated with different levels of precision (number of branches generated after applying one rule on the literal, number of branches after two rules, on the literal and in the results, etc.). The trade-off between classifying constraints and the gain from such classification should be taken into account. The first literals to be selected would be those whose entailment check does not require branching, i.e. equalities between syntactically equal terms, ground equalities between terms known by the E-graph or whose function symbols are different.

Discarding unsuitable branches Matching a term $f(\bar{u}_n)$ with a ground term $f(\bar{t}_n)$ fails unless all their ground arguments of matching position are congruent. Therefore, after an assignment and the subsequent update of the E-graph, if an argument of a term $f(\bar{u}_n)$ occurring in one of the remaining constraints becomes ground, it can be checked whether there is a ground term $f(\bar{t}_n) \in \mathbf{T}(E)$ such that, for every ground argument u_i, $E \models u_i \simeq t_i$, i.e. $\mathit{find}(u_i) = \mathit{find}(t_i)$. If no such term exists and $f(\bar{u}_n)$ is not in a literal compatible with a UCOMP scenario, the whole "branch" can be eagerly discarded.

Example 5.3. Considering the CCFV calculus, in a state $E_\sigma \Vdash_E x \simeq t \wedge f(g(x), y) \simeq h(z) \wedge C$,

assume that ASSIGN is applied. Then the term $f(g(x), y)$ becomes $f(g(t), y)$ in the branch. A necessary condition for the literal $f(g(t), y) \simeq h(z)$ to be entailed is that there exists some $f(t_1, t_2) \in \mathbf{T}(E)$ such that $E \models g(t) \simeq t_1$.

To efficiently implement this technique it is necessary to query with a function symbol and pairs of ground terms and positions whether there is any term in E^{CC} with the same top symbol and whose arguments are congruent with the given ground terms in the respective positions. The index operation would have the form $\mathit{apps}(f, \langle w_1, \ldots, w_n \rangle)$, in which each w_i is either a ground term or a nullary symbol $*$, and yield all terms $f(t_1, \ldots, t_n)$ in E^{CC} such that, at each position $i \in \{1..n\}$ in which $w_i \neq *$, $E \models t_i \simeq w_i$. In the example above, the query would be whether $\mathit{apps}(f, \langle g(t), * \rangle)$ is empty. The operation $\mathit{apps}(f, \langle w_1, \ldots, w_n \rangle)$ can be implemented by building a forest in which each tree has as root a function symbol, each edge is annotated with a term, and whose leafs are the applications of that symbol in E^{CC} whose arguments are the annotations in the edges leading to that leaf, in the respective order. Alternatively the hash table kept internally for efficiently manipulating the E-graph could be used. This would require adding 2^n entries for each function symbol into the table, in which n is the arity of the respective function symbol. Therefore it is necessary to carefully analyse which function symbols can benefit from this check.

5.2.1 Breadth-first CCFV

The above algorithm implements a depth-first search for solutions, following the backtracking CCFV calculus. Here we present an abstract algorithm in Figure 5.2. that implements the non-backtracking CCFV calculus described in Section 4.3 with the strategy from Section 4.3.1. The algorithm, henceforth CCFV-BREADTH, takes as input a conjunctive set of constraints to be entailed. It produces the set of all solutions for the input constraints. Of course, as before, this set may be empty if there are no solutions. CCFV-BREADTH uses the same data structures as its depth-first counterpart. However, it explicitly manipulates the different partial solutions produced during the solving. Moreover, since in this strategy the variable assignments only occur when the remaining constraints are ground, there is no need to have an E-graph for the terms in L to be updated according to assignments.

The algorithm also has break-points with recursive calls from within loops of independent possibilities. However, the recursive call contains only "local constraints", generated from the analysis of the currently selected constraint. After the call is finished, the results are combined with those from previously selected constraints. This way, constraints are solved independently and the found solutions are combined according to how these constraints have been generated. The crucial operations in CCFV-BREADTH are the combination operations, i.e. the implementations of JOIN and MEET. They are represented in the algorithm by the symbols $\sqcap$ and $\sqcup$, respectively, and are performed directly on sets of solutions. For now we have naïve implementations of these combinations. The data structure we use for representing solutions is not optimised

function CCFV(C) **is**
  SOLS $\leftarrow \{\top\}$
  while $C \neq \varnothing$ and SOLS $\neq \varnothing$ **do** // Select constraint
    $c \leftarrow \text{SEL}(C)$; $C \leftarrow C \setminus \{c\}$; C-SOLS $\leftarrow \{\top\}$
    match c **with**
      $t_1 \not\simeq t_2$: // Ground disequality
        if $t_1 \notin \mathit{diseqs}(t_2)$ **then** C-SOLS $\leftarrow \varnothing$
      $t_1 \simeq t_2$: // Ground equality
        if $t_1, t_2 \in \mathbf{T}(E)$ **then**
          if $\mathit{find}(t_1) \neq \mathit{find}(t_2)$ **then** C-SOLS $\leftarrow \varnothing$
        else if $t_1 = f(\bar{s}_n)$ and $t_2 = f(\bar{r}_n)$ **then**
          C-SOLS $\leftarrow$ CCFV($\{s_1 \simeq r_1, \ldots, s_n \simeq r_n\}$)
        else
          C-SOLS $\leftarrow \varnothing$
      $x \simeq s$:
        if $x \notin \text{FV}(s)$ **then** // Assignment
          C-SOLS $\leftarrow \{x \simeq s\}$
        else // s is a non-ground function application
          let $s = f(\bar{u}_n)$ **in**
            for $t \in \mathit{reps}(f)$ **do** // UVAR
              C-SOLS $\leftarrow$ C-SOLS $\sqcup$ CCFV($\{f(\bar{u}_n) \simeq t, x \simeq t\}$)
      $f(\bar{u}_n) \simeq t$: // E-matching
        for $f(\bar{t}_n) \in \mathit{apps}(f, t)$ **do**
          C-SOLS $\leftarrow$ C-SOLS $\sqcup$ CCFV($\{u_1 \simeq t_1, \ldots, u_n \simeq t_n\}$)
      $f(\bar{u}_n) \simeq g(\bar{u}'_n)$:
        if $f = g$ **then** // E-unify arguments
          C-SOLS $\leftarrow$ C-SOLS $\sqcup$ CCFV($\{u_1 \simeq u'_1, \ldots, u_n \simeq u'_n\}$)
        for $t \in \mathit{reps}(f)$ **do** // E-matching into same class
          C-SOLS $\leftarrow$ C-SOLS $\sqcup$ CCFV($\{f(\bar{u}_n) \simeq t,\ g(\bar{u}'_n) \simeq t\}$)
      $x \not\simeq t$:
        for $t' \in \mathit{diseqs}(t)$ **do** // Assignment into a disequal term
          C-SOLS $\leftarrow$ C-SOLS $\sqcup$ CCFV($\{x \simeq t'\}$)
      $f(\bar{u}_n) \not\simeq t$: // E-matching into a disequal term
        for $t' \in \mathit{diseqs}(t)$ **do**
          C-SOLS $\leftarrow$ C-SOLS $\sqcup$ CCFV($\{f(\bar{u}_n) \simeq t'\}$)
      $x \not\simeq f(\bar{u}_n)$: // DFAPP
        for $t \in \mathit{reps}(f)$ **do**
          C-SOLS $\leftarrow$ C-SOLS $\sqcup$ CCFV($\{f(\bar{u}_n) \simeq t, x \not\simeq t\}$)
      $f(\bar{u}_n) \not\simeq g(\bar{u}'_n)$: // E-matching into disequal terms
        for $t \in \mathit{reps}(f)$ **do**
          C-SOLS $\leftarrow$ C-SOLS $\sqcup$ CCFV($\{f(\bar{u}_n) \simeq t, g(\bar{u}'_n) \not\simeq t\}$)
      $x \not\simeq y$: // DVAR
        for each signature t in E^{CC} **do**
          for $t' \in \mathit{diseqs}(t)$ **do**
            C-SOLS $\leftarrow$ C-SOLS $\sqcup$ CCFV($\{x \simeq t, y \simeq t'\}$)
    SOLS $\leftarrow$ SOLS $\sqcap$ C-SOLS // Merge solutions from constraint
  return SOLS // Yield found solutions

Figure 5.2: Breadth-first CCFV algorithm

for it. Moskal et al. [MŁK08] have a somewhat similar approach for their E-matching algorithm in which sets of partial solutions need to be combined. By representing these sets as ordered trees they are able to efficiently implement combination operations. We would need to extend such data structures for our more complex solution representations, since we may have variables associated with other variables or non-ground terms, rather than only with ground terms.

Example 5.4. Consider again E, L and E^{CC} as in Example 5.2 and assume that all the indexing operations from Section 5.1 are available. The execution of the CCFV-BREADTH algorithm would start with CCFV(L). Let $h(x_1) \simeq h(c)$ be the first constraint considered in the outer **while** loop. The possibilities to solve $h(x_1) \simeq h(c)$ are enumerated in the E-matching case. This constraint is solved independently and the obtained orthogonal solutions are

$$\text{C-SOLS} = \{\{x_1 \simeq a\}, \{x_1 \simeq c\}\}$$

Let the selection function next consider the constraint $h(x_2) \not\simeq g(x_3)$. Independently solving this constraint yields

$$\text{C-SOLS} = \{\{x_2 \simeq c, x_3 \simeq b\}, \{x_2 \simeq a, x_3 \simeq b\}\}$$

These solutions are then merged with the solutions from the previous constraint, forming the global set of solution

$$\begin{aligned} \text{SOLS} &= \{\{x_1 \simeq a\}, \{x_1 \simeq c\}\} \sqcap \{\{x_2 \simeq c, x_3 \simeq b\}, \{x_2 \simeq a, x_3 \simeq b\}\} \\ &= \left\{ \begin{array}{l} \{x_1 \simeq c, x_2 \simeq c, x_3 \simeq b\}, \\ \{x_1 \simeq c, x_2 \simeq a, x_3 \simeq b\}, \\ \{x_1 \simeq a, x_2 \simeq c, x_3 \simeq b\}, \\ \{x_1 \simeq a, x_2 \simeq a, x_3 \simeq b\} \end{array} \right\} \end{aligned}$$

By proceeding analogously the CCFV-BREADTH algorithm will find the same solutions as in Example 4.11 pretty much in the same way.

Memoization Quite often CCFV-BREADTH needs to look for solutions for the same E-ground (dis)unification problem. To avoid duplication of work, we store the result of solving constraints. Thus whenever that constraint is met again the respective solutions may be directly retrieved from the memory instead of recomputed. For now we have only a coarse index for constraints, but the technique is nevertheless effective in making CCFV-BREADTH faster.

A more ambitious application of memoization is to keep the results of solving constraints across different calls to the instantiation module. This would allow updating the stored results according to how the candidate assignment $E \cup \mathcal{Q}$ changed, as the theory solvers do in the CDCL($\mathcal{T}$) framework. This is a possibility for having an incremental CCFV algorithm, however its viability has not been tested in a practical implementation.

5.2.2 Applying instantiation techniques

We here describe how to implement trigger and conflict based instantiation on top of CCFV. We leave the implementation of model based instantiation, whose theoretical suitability we have outlined in the previous chapter, for future work. In veriT we have also not yet implemented the functionality to discard instances generated from triggers that are already entailed nor the eventual propagation of equalities as a byproduct of failed search for conflicting instances.

Trigger based instantiation We follow the guidelines from Leino and Pit-Claudel [LP16] outlined in Section 3.2.1 for selecting triggers. For a given quantified formula with a trigger $T = \{u_1, \ldots, u_n\}$, solving the E-ground (dis)unification

$$E \models (u_1 \simeq y_1 \wedge \cdots \wedge u_n \simeq y_n)\,\sigma$$

with the restriction that all y_i must be grounded, provides all the desired trigger instantiations. Instead of changing the CCFV algorithm to perform the restricted search, we use an outer loop that directly builds the E-ground (dis)unification problems in which $\bar{y}_n$ have been grounded. The terms enumerated for grounding $\bar{y}_n$ are retrieved from $apps(f)$. CCFV can then be applied straightforwardly to collect the solutions. All found solutions are used for building instantiation lemmas.

Conflict based instantiation To apply conflict based instantiation it suffices to have an outer loop that produces the conjunctive negation of a quantified formula's body, which compose a set of constraints to be directly solved by CCFV. One point to consider is which instances to generate from the found solutions to the E-ground (dis)unification problem. One conflicting instance is sufficient to rule out the current assignment $E \cup \mathcal{Q}$, and this is indeed the strategy used in CVC4, the first SMT solver to use conflict based instantiation. In veriT, however, instantiating the quantified formulas with all found conflicting substitutions has been, so far, more effective than generating the minimal amount of conflicting instances.

5.3 Experiments

Here we evaluate the impact of optimizations and instantiation techniques based on CCFV. We make comparisons with previous versions of the SMT solvers veriT and CVC4, as well as with the state-of-the-art instantiation based solver Z3 [dMB08b]. Different configurations are identified in this section according to which techniques and algorithms they have activated:

- **t** : trigger instantiation through CCFV (see "Trigger based instantiation" in Section 5.2.2);
- **c** : conflict based instantiation through CCFV (see "Conflict based instantiation" in Section 5.2.2);

b : implements the breadth-first version of CCFV rather than the depth-first one (see Section 5.2.1);

e : optimization for eagerly discarding branches with unmatchable applications (see "Discarding unsuitable branches" in Section 5.2);

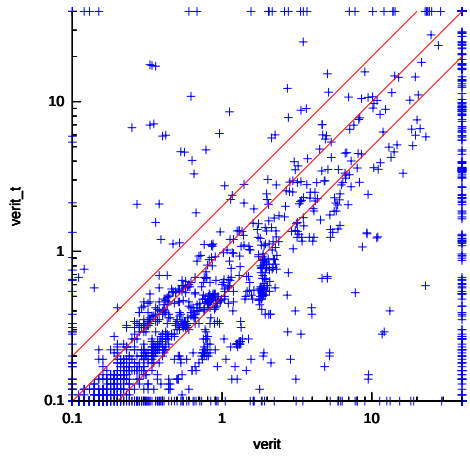
d : discards already entailed trigger based instances (see "Discarding entailed instances" in Section 4.2.3)

The configuration **verit** refers to the previous version of veriT, which only offered support for quantified formulas through naïve trigger instantiation, without further optimizations. The configuration **cvc** refers to version 1.5 of CVC4, which applies **t** and **c** by default, as well as propagation of equalities. Both veriT and CVC4 implement efficient term indexing and apply a simple selection heuristic, checking ground and reflexive literals first but otherwise considering the conjunction of constraints as a queue. The breadth-first variation of CCFV is implemented only in veriT, using the simple strategy described in Sections 5.4 and 4.3.1.

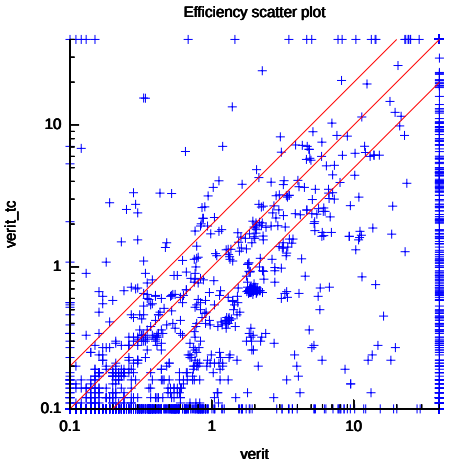
The evaluation was made on the UF, UFLIA, UFLRA and UFIDL categories of SMT-LIB [BFT15], with 10 495 benchmarks annotated as *unsatisfiable*[5], mostly stemming for verification and ITP platforms. The categories with bit vectors and non-linear arithmetic are currently not supported by veriT and in those in which uninterpreted functions are not predominant the techniques shown here are not as effective. Since veriT cannot produce models for formulas with quantifiers and the finite-model finding techniques in CVC4 are not affected by CCFV, only unsatisfiable problems were considered. Our experiments were conducted using machines with 2 CPUs Intel Xeon E5-2630 v3, 8 cores/CPU, 126GB RAM, 2x558GB HDD. The timeout was set for 30 seconds, since our goal is evaluating SMT solvers as back-ends of verification and ITP platforms, which require fast answers.

Figure 5.3 exhibits an important impact of CCFV and the techniques and optimizations built on top of it. **verit**+**t** performs much better than **verit**, solely due to CCFV. Moreover, **verit**+**tc** presents a significant improvement in terms of problems solved (474 more against 36 less) by the use of the conflict based instantiation, but it also shows a less clear gain of time. Besides the difficulty to predict how combining any given technique with trigger based instantiation will affect performance, we believe that this less clear gain in time is due to the more expensive search performed: trying to falsify quantified formulas and handling full E-ground (dis)unification, which, in the context of SMT, has a much bigger search space than simply performing E-matching for pattern-matching instantiation. Not always the "better quality" of the conflicting instances offsets the time it took to compute them, which indicates the necessity to identify beforehand such cases and avoid the more expensive search when counter-productive. A comparison of both flavours of CCFV is shown in Figure 5.4. Both variations perform well, with the depth-first CCFV outperforming its breadth-first counterpart by a small margin. This shows that both approaches are viable. **cvc**+**d** and **cvc**+**e** improve significantly over **cvc**, exhibiting

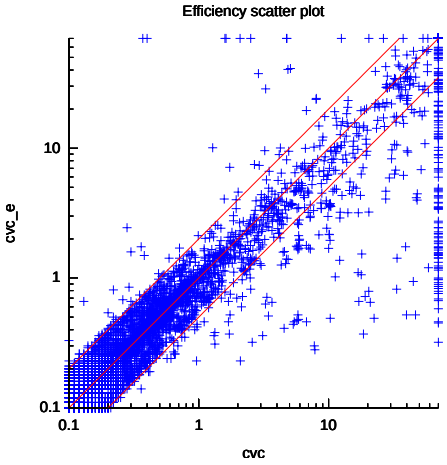
[5] As of 2016.



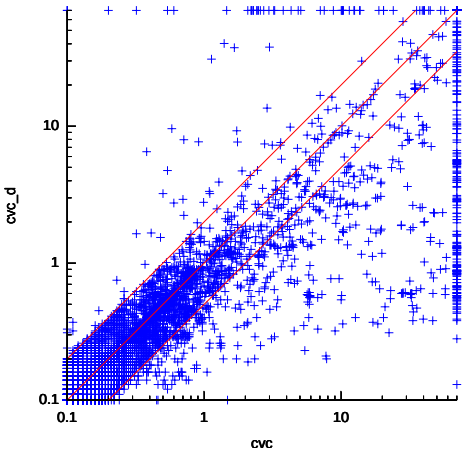
(a) **verit+t** vs. **verit**



(b) **verit+tc** vs. **verit**



(c) **cvc+e** vs. **cvc**



(d) **cvc+d** vs. **cvc**

Figure 5.3: Improvements in veriT and CVC4

Logic	Class	Z3	**cvc+d**	**cvc+e**	**cvc**	**verit+tc**	**verit+tcb**	**verit+t**	**verit**
UF	grasshopper	418	411	420	415	430	**435**	418	413
	sledgehammer	1249	1438	**1456**	1428	1277	1278	1134	1066
UFIDL	all	**62**	**62**	**62**	**62**	58	58	58	58
UFLIA	boogie	**852**	844	834	801	706	690	660	661
	sexpr	**26**	12	11	11	7	7	5	5
	grasshopper	341	322	326	319	356	**361**	340	335
	sledgehammer	1581	1944	**1953**	1929	1790	1799	1620	1569
	simplify	**831**	766	706	705	803	801	735	690
	simplify2	**2337**	2330	2292	2286	2307	2303	2291	2177
	Total	7697	**8129**	8060	7956	7734	7736	7261	6916

Table 5.1: Instantiation based SMT solvers on SMT-LIB benchmarks

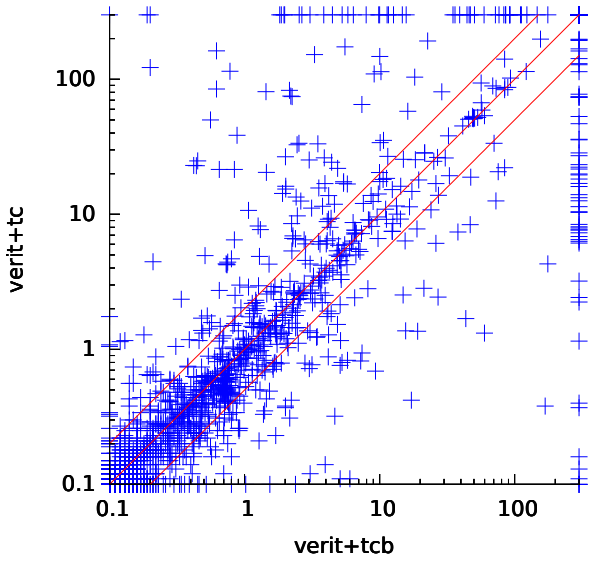


Figure 5.4: Depth-first versus breadth-first CCFV

the advantage of techniques based on the CCFV features for entailment checking on the presence of free variables.

The comparison between the different configurations of veriT and CVC4 with the SMT solver Z3 (version 4.4.2) is summarised in Table 5.1, excluding categories (such as UFLRA) whose problems are trivially solved by all systems, which leaves 8 701 problems for consideration. The table exhibits the total number of problems solved by each configuration within the timeout. **verit+tc** solves approximately 800 more problems than **verit**, solving approximately the same total number of problems than Z3. Many of these new problems come from the *sledgehammer* benchmarks, which contain less theory symbols and whose solving relies more on equational first-order logic. Moreover, **verit+tc** performs best in the *grasshopper* families, stemming from the heap verification tool GRASShopper [PWZ14]. Considering the overall performance, both **cvc+d** and **cvc+e** solve significantly more problems than **cvc**, specially in benchmarks from verification platforms, approaching the performance of Z3 in these families. Both these techniques, as well as the propagation of equalities, are fairly important points in the performance of CVC4, so their implementation is a clear direction for improvements in veriT.

Part II

Proof Production

Chapter 6

Processing calculus

An increasing number of automatic theorem provers can generate certificates, or proofs, that justify the formulas they derive. These proofs can be checked by other programs and shared across reasoning systems. Some users will also want to inspect this output to understand why a formula holds. Proof production is generally well understood for the core proving methods and for many theories commonly used in satisfiability modulo theories (SMT). But most automatic provers also perform some formula processing or preprocessing—such as clausification and rewriting with theory-specific lemmas—and proof production for this aspect is less mature.

For most provers, the code for processing formulas is lengthy and deals with a multitude of cases, some of which are rarely executed. Although it is crucial for efficiency, this code tends to be given much less attention than other aspects of provers. Developers are reluctant to invest effort in producing detailed proofs for such processing, since this requires adapting a lot of code. As a result, the granularity of inferences for formula processing is often coarse. Sometimes, processing features are even disabled to avoid gaps in proofs, at a high cost in proof search performance.

Fine-grained proofs are important for a variety of applications. We propose a framework to generate such proofs without slowing down proof search. This chapter describes a proof calculus in which to generate fine-grained proofs for processing formulas. The next chapter introduces an scalable framework for generating such proofs. The work described in both chapters led to a joint publication with Jasmin Blanchette and Pascal Fontaine [BBF17].

Conventions For the following chapters, we assume that our language for many-sorted first-order logic with equality contains two more binders besides the quantifiers: Hilbert's choice operator $\varepsilon x.\, \varphi$ and a 'let' construct, let $\bar{x}_n \simeq \bar{s}_n$ in t, which simultaneously assigns n variables that can be used in the body t.

As before, we use the symbol $=$ for syntactic equality on terms. The symbol $=_\alpha$ stands for syntactic equality up to renaming of bound variables. We reserve the names $\mathsf{a}, \mathsf{c}, \mathsf{f}, \mathsf{g}, \mathsf{p}, \mathsf{q}$ for function symbols; x, y, z for variables; r, s, t, u for terms (which may be formulas); φ, ψ for formulas; and Q for quantifiers ($\forall$ and $\exists$).

The notation $t[\bar{x}_n]$ stands for a term that may depend on distinct variables $\bar{x}_n$; $t[\bar{s}_n]$ is the

corresponding term where the terms $\bar{s}_n$ are simultaneously substituted for $\bar{x}_n$; bound variables in t are renamed to avoid capture. Following these conventions, Hilbert choice and 'let' are characterized by

$$\models \exists x.\, \varphi[x] \longrightarrow \varphi[\varepsilon x.\, \varphi] \tag{ε_1}$$

$$\models (\forall x.\, \varphi \simeq \psi) \longrightarrow (\varepsilon x.\, \varphi) \simeq (\varepsilon x.\, \psi) \tag{ε_2}$$

$$\models (\text{let } \bar{x}_n \simeq \bar{s}_n \text{ in } t[\bar{x}_n]) \simeq t[\bar{s}_n] \tag{let}$$

The property (ε_2) may seem unnecessary, but it provides determinism for the choice operator. In particular, it ensures that reflexivity holds for terms involving ε, such that the substitution of equals for equals is correct for such terms.

6.1 Inference system

The inference rules used by our framework depend on a notion of *context* defined by the grammar

$$\Gamma ::= \varnothing \mid \Gamma, x \mid \Gamma, \bar{x}_n \mapsto \bar{s}_n$$

The empty context $\varnothing$ is also denoted by a blank. Each context entry either *fixes* a variable x or defines a *substitution* $\{\bar{x}_n \mapsto \bar{s}_n\}$. Any variables arising in the terms $\bar{s}_n$ will normally have been introduced in the context Γ on the left. If a context introduces the same variable several times, the rightmost entry shadows the others.

Abstractly, a context Γ fixes a set of variables and specifies a substitution $\mathit{subst}(\Gamma)$. The substitution is the identity for $\varnothing$ and is defined as follows in the other cases:

$$\mathit{subst}(\Gamma, x) = \mathit{subst}(\Gamma)[x \mapsto x] \qquad \mathit{subst}(\Gamma, \bar{x}_n \mapsto \bar{t}_n) = \mathit{subst}(\Gamma) \circ \{\bar{x}_n \mapsto \bar{t}_n\}$$

In the first equation, the $[x \mapsto x]$ update shadows any replacement of x induced by Γ. The examples below illustrate this subtlety:

$$\mathit{subst}(x \mapsto 7, x \mapsto \mathsf{g}(x)) = \{x \mapsto \mathsf{g}(7)\} \qquad \mathit{subst}(x \mapsto 7, x, x \mapsto \mathsf{g}(x)) = \{x \mapsto \mathsf{g}(x)\}$$

We write $\Gamma(t)$ to abbreviate the capture-avoiding substitution $\mathit{subst}(\Gamma)(t)$.

Transformations of terms (and formulas) are justified by judgments of the form $\Gamma \rhd t \simeq u$, where Γ is a context, t is an unprocessed term, and u is the corresponding processed term. The free variables in t and u must appear in the context Γ. Semantically, the judgment expresses the equality of the terms $\Gamma(t)$ and u for all variables fixed by Γ. Crucially, the substitution applies only on the left-hand side of the equality.

The inference rules for the transformations covered in this thesis are presented below, followed by explanations.

$$\frac{}{\Gamma \rhd t \simeq u}\ \textsc{Taut}_{\mathcal{T}} \quad \text{if } \models_{\mathcal{T}} \Gamma(t) \simeq u \qquad \frac{\Gamma \rhd s \simeq t \quad \Gamma \rhd t \simeq u}{\Gamma \rhd s \simeq u}\ \textsc{Trans} \quad \text{if } \Gamma(t) = t$$

$$\frac{\big(\Gamma \rhd t_i \simeq u_i\big)_{i=1}^{n}}{\Gamma \rhd \mathsf{f}(\bar{t}_n) \simeq \mathsf{f}(\bar{u}_n)}\ \textsc{Cong} \qquad \frac{\Gamma,\, y,\, x \mapsto y \rhd \varphi \simeq \psi}{\Gamma \rhd (Qx.\, \varphi) \simeq (Qy.\, \psi)}\ \textsc{Bind} \quad \text{if } y \notin \mathrm{FV}(Qx.\, \varphi)$$

$$\frac{\Gamma,\, x \mapsto (\varepsilon x.\, \varphi) \rhd \varphi \simeq \psi}{\Gamma \rhd (\exists x.\, \varphi) \simeq \psi}\ \textsc{Sko}_{\exists} \qquad \frac{\Gamma,\, x \mapsto (\varepsilon x.\, \neg\varphi) \rhd \varphi \simeq \psi}{\Gamma \rhd (\forall x.\, \varphi) \simeq \psi}\ \textsc{Sko}_{\forall}$$

$$\frac{\big(\Gamma \rhd r_i \simeq s_i\big)_{i=1}^{n} \quad \Gamma,\, \bar{x}_n \mapsto \bar{s}_n \rhd t \simeq u}{\Gamma \rhd (\text{let } \bar{x}_n \simeq \bar{r}_n \text{ in } t) \simeq u}\ \textsc{Let} \quad \text{if } \Gamma(s_i) = s_i \text{ for all } i \in [n]$$

– Taut$_{\mathcal{T}}$ relies on an oracle $\models_{\mathcal{T}}$ to derive arbitrary lemmas in a theory $\mathcal{T}$. In practice, the oracle will produce some kind of certificate to justify the inference. An important special case, for which we use the name Refl, is syntactic equality (up to renaming of bound variables); the side condition is then $\Gamma(t) =_\alpha u$. (We use $=_\alpha$ instead of $=$ because applying a substitution can rename bound variables.)

– Trans needs the side condition because the term t appears both on the left-hand side of $\simeq$ (where it is subject to Γ's substitution) and on the right-hand side (where it is not). Without the side condition, the two occurrences of t in the antecedent could denote different terms.

– Cong can be used for any function symbol f, including the logical connectives.

– Bind is a congruence rule for quantifiers. The rule also justifies the renaming of the bound variable(from x to y). The side condition prevents an unwarranted variable capture. In the antecedent, the renaming is expressed by a substitution in the context. If $x = y$, the context is $\Gamma,\, x,\, x \mapsto x$, which has the same meaning as $\Gamma,\, x$.

– Sko$_{\exists}$ and Sko$_{\forall}$ exploit (ε_1) to replace a quantified variable with a suitable witness, simulating skolemization. We can think of the ε expression in each rule abstractly as a fresh function symbol that takes any fixed variables it depends on as arguments. In the antecedents, the replacement is performed by the context.

– Let exploits (let) to expand a 'let' expression. Again, a substitution is used. The terms $\bar{r}_n$ assigned to the variables $\bar{x}_n$ can be transformed into terms $\bar{s}_n$.

The antecedents of all the rules inspect subterms structurally, without modifying them. Modifications to the term on the left-hand side are delayed; the substitution is applied only in Taut. This is crucial to obtain compact proofs that can be checked efficiently. Some of the side conditions may look computationally expensive, but there are ways to compute them fairly efficiently.

Furthermore, by systematically renaming variables in Bind, we can satisfy most side conditions trivially, as we will prove in Sect. 6.2.

The set of rules can be extended to cater for arbitrary transformations that can be expressed as equalities, using Hilbert choice to represent fresh symbols if necessary. The usefulness of Hilbert choice for proof reconstruction is well known [dNiv05, PS07, BW10], but we push the idea further and use it to simplify and uniformize the inference system.

Example 6.1. The following derivation tree justifies the expansion of a 'let' expression:

$$\dfrac{\dfrac{}{\rhd \mathsf{a} \simeq \mathsf{a}}\,\textsc{Cong} \qquad \dfrac{\dfrac{}{x \mapsto \mathsf{a} \rhd x \simeq \mathsf{a}}\,\textsc{Refl} \qquad \dfrac{}{x \mapsto \mathsf{a} \rhd x \simeq \mathsf{a}}\,\textsc{Refl}}{x \mapsto \mathsf{a} \rhd \mathsf{p}(x,x) \simeq \mathsf{p}(\mathsf{a},\mathsf{a})}\,\textsc{Cong}}{\rhd (\mathrm{let}\ x \simeq \mathsf{a}\ \mathrm{in}\ \mathsf{p}(x,x)) \simeq \mathsf{p}(\mathsf{a},\mathsf{a})}\,\textsc{Let}$$

It is also possible to further process the substituted term, as in this derivation:

$$\dfrac{\dfrac{}{\rhd \mathsf{a} + 0 \simeq \mathsf{a}}\,\textsc{Taut}_{+} \qquad \dfrac{\vdots}{x \mapsto \mathsf{a} \rhd \mathsf{p}(x,x) \simeq \mathsf{p}(\mathsf{a},\mathsf{a})}\,\textsc{Cong}}{\rhd (\mathrm{let}\ x \simeq \mathsf{a} + 0\ \mathrm{in}\ \mathsf{p}(x,x)) \simeq \mathsf{p}(\mathsf{a},\mathsf{a})}\,\textsc{Let}$$

Example 6.2. The following derivation tree, in which ε_x abbreviates $\varepsilon x.\, \neg\, \mathsf{p}(x)$, justifies the skolemization of the quantifier in the formula $\neg\, \forall x.\, \mathsf{p}(x)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x \mapsto \varepsilon_x \rhd x \simeq \varepsilon_x}\,\textsc{Refl}}{x \mapsto \varepsilon_x \rhd \mathsf{p}(x) \simeq \mathsf{p}(\varepsilon_x)}\,\textsc{Cong}}{\rhd (\forall x.\, \mathsf{p}(x)) \simeq \mathsf{p}(\varepsilon_x)}\,\textsc{Sko}_{\forall}}{\rhd (\neg\, \forall x.\, \mathsf{p}(x)) \simeq \neg\, \mathsf{p}(\varepsilon_x)}\,\textsc{Cong}$$

The Cong inference above $\textsc{Sko}_{\forall}$ is optional; we could have directly closed the derivation with Refl. In a prover, the term ε_x would be represented by a fresh Skolem constant c, and we would ignore c's connection to ε_x during proof search.

Skolemization can be applied regardless of polarity. Normally, we skolemize only positive existential quantifiers and negative universal quantifiers. However, skolemizing other quantifiers is sound in the context of proving. The trouble is that it is generally incomplete, if we introduce Skolem symbols and forget their definitions in terms of Hilbert choice. To paraphrase Orwell, all quantifiers are skolemizable, but some quantifiers are more skolemizable than others.

Example 6.3. The next derivation tree illustrates the interplay between the theory rule $\textsc{Taut}_{\mathcal{T}}$

and the equality rules Trans and Cong:

$$\dfrac{\dfrac{\dfrac{}{\rhd \mathsf{k} \simeq \mathsf{k}}\,\textsc{Cong} \quad \dfrac{}{\rhd 1{\times}0 \simeq 0}\,\textsc{Taut}_{\times}}{\rhd \mathsf{k}+1{\times}0 \simeq \mathsf{k}+0}\,\textsc{Cong} \quad \dfrac{}{\rhd \mathsf{k}+0 \simeq \mathsf{k}}\,\textsc{Taut}_{+}}{\rhd \mathsf{k}+1{\times}0 \simeq \mathsf{k}}\,\textsc{Trans} \quad \dfrac{}{\rhd \mathsf{k} \simeq \mathsf{k}}\,\textsc{Cong}}{\rhd (\mathsf{k}+1{\times}0 < \mathsf{k}) \simeq (\mathsf{k} < \mathsf{k})}\,\textsc{Cong}$$

We could extend the tree at the bottom with an extra application of Trans and $\textsc{Taut}_{<}$ to simplify $\mathsf{k} < \mathsf{k}$ further to false. The example demonstrates that theories can be arbitrarily fine-grained, which often makes proof checking easier. At the other extreme, we could have derived $\rhd (\mathsf{k}+1{\times}0 < \mathsf{k}) \simeq \mathsf{false}$ using a single $\textsc{Taut}_{+\cup\times\cup<}$ inference.

Example 6.4. The tree below illustrates what can go wrong if we ignore side conditions:

$$\dfrac{\dfrac{\dfrac{}{\Gamma_1 \rhd \mathsf{f}(x) \simeq \mathsf{f}(x)}\,\textsc{Refl} \quad \dfrac{\dfrac{}{\Gamma_2 \rhd x \simeq \mathsf{f}(x)}\,\textsc{Refl} \quad \dfrac{}{\Gamma_3 \rhd \mathsf{p}(y) \simeq \mathsf{p}(\mathsf{f}(\mathsf{f}(x)))}\,\textsc{Refl}}{\Gamma_2 \rhd (\text{let } y \simeq x \text{ in } \mathsf{p}(y)) \simeq \mathsf{p}(\mathsf{f}(\mathsf{f}(x)))}\,\textsc{Let}^{*}}{\Gamma_1 \rhd (\text{let } x \simeq \mathsf{f}(x) \text{ in let } y \simeq x \text{ in } \mathsf{p}(y)) \simeq \mathsf{p}(\mathsf{f}(\mathsf{f}(x)))}\,\textsc{Let}}{\rhd (\forall x.\ \text{let } x \simeq \mathsf{f}(x) \text{ in let } y \simeq x \text{ in } \mathsf{p}(y)) \simeq (\forall x.\ \mathsf{p}(\mathsf{f}(\mathsf{f}(x))))}\,\textsc{Bind}$$

In the above, $\Gamma_1 = x,\, x \mapsto x$; $\Gamma_2 = \Gamma_1,\, x \mapsto \mathsf{f}(x)$; and $\Gamma_3 = \Gamma_2,\, y \mapsto \mathsf{f}(x)$. The inference marked with an asterisk (*) is illegal, because $\Gamma_2(\mathsf{f}(x)) = \mathsf{f}(\mathsf{f}(x)) \neq \mathsf{f}(x)$. We exploit this to derive an invalid judgment, with a spurious application of f on the right-hand side. To apply Let legally, we must first rename the universally quantified variable x to a fresh variable z using the Bind rule:

$$\dfrac{\dfrac{\dfrac{}{\Gamma_1 \rhd \mathsf{f}(x) \simeq \mathsf{f}(z)}\,\textsc{Refl} \quad \dfrac{\dfrac{}{\Gamma_2 \rhd x \simeq \mathsf{f}(z)}\,\textsc{Refl} \quad \dfrac{}{\Gamma_3 \rhd \mathsf{p}(y) \simeq \mathsf{p}(\mathsf{f}(z))}\,\textsc{Refl}}{\Gamma_2 \rhd (\text{let } y \simeq x \text{ in } \mathsf{p}(y)) \simeq \mathsf{p}(\mathsf{f}(z))}\,\textsc{Let}}{\Gamma_1 \rhd (\text{let } x \simeq \mathsf{f}(x) \text{ in let } y \simeq x \text{ in } \mathsf{p}(y)) \simeq \mathsf{p}(\mathsf{f}(z))}\,\textsc{Let}}{\rhd (\forall x.\ \text{let } x \simeq \mathsf{f}(x) \text{ in let } y \simeq x \text{ in } \mathsf{p}(y)) \simeq (\forall z.\ \mathsf{p}(\mathsf{f}(z)))}\,\textsc{Bind}$$

This time, we have $\Gamma_1 = z,\, x \mapsto z$; $\Gamma_2 = \Gamma_1,\, x \mapsto \mathsf{f}(z)$; and $\Gamma_3 = \Gamma_2,\, y \mapsto \mathsf{f}(z)$. Let's side condition is satisfied: $\Gamma_2(\mathsf{f}(z)) = \mathsf{f}(z)$.

Example 6.5. The dangers of capture are illustrated by the following tree, where ε_y stands for $\varepsilon y.\, \mathsf{p}(x) \wedge \forall x.\, \mathsf{q}(x, y)$:

$$\dfrac{\dfrac{\dfrac{}{x, y \mapsto \varepsilon_y \rhd (\mathsf{p}(x) \wedge \forall x.\, \mathsf{q}(x, y)) \simeq (\mathsf{p}(x) \wedge \forall x.\, \mathsf{q}(x, \varepsilon_y))}\,\textsc{Refl}^{*}}{x \rhd (\exists y.\, \mathsf{p}(x) \wedge \forall x.\, \mathsf{q}(x, y)) \simeq (\mathsf{p}(x) \wedge \forall x.\, \mathsf{q}(x, \varepsilon_y))}\,\textsc{Sko}_{\exists}}{\rhd (\forall x.\, \exists y.\, \mathsf{p}(x) \wedge \forall x.\, \mathsf{q}(x, y)) \simeq (\forall x.\, \mathsf{p}(x) \wedge \forall x.\, \mathsf{q}(x, \varepsilon_y))}\,\textsc{Bind}$$

The inference marked with an asterisk would be legal if Refl's side condition were stated using a capturing substitution. The final judgment is unwarranted, because the free variable x in the first conjunct of ε_y is captured by the inner universal quantifier on the right-hand side.

To avoid the capture, we rename the inner bound variable x to z. Then it does not matter if we use a capture-avoiding or a capturing substitution:

$$\dfrac{\dfrac{}{x, y \mapsto \varepsilon_y \rhd \mathsf{p}(x) \simeq \mathsf{p}(x)}\,\textsc{Refl} \quad \dfrac{\dfrac{}{x, y \mapsto \varepsilon_y, x \mapsto z \rhd \mathsf{q}(x, y) \simeq \mathsf{q}(z, \varepsilon_y)}\,\textsc{Refl}}{x, y \mapsto \varepsilon_y \rhd (\forall x.\, \mathsf{q}(x, y)) \simeq (\forall z.\, \mathsf{q}(z, \varepsilon_y))}\,\textsc{Bind}}{\dfrac{x, y \mapsto \varepsilon_y \rhd (\mathsf{p}(x) \wedge \forall x.\, \mathsf{q}(x, y)) \simeq (\mathsf{p}(x) \wedge \forall z.\, \mathsf{q}(z, \varepsilon_y))}{\dfrac{x \rhd (\exists y.\, \mathsf{p}(x) \wedge \forall x.\, \mathsf{q}(x, y)) \simeq (\mathsf{p}(x) \wedge \forall z.\, \mathsf{q}(z, \varepsilon_y))}{\rhd (\forall x.\, \exists y.\, \mathsf{p}(x) \wedge \forall x.\, \mathsf{q}(x, y)) \simeq (\forall x.\, \mathsf{p}(x) \wedge \forall z.\, \mathsf{q}(z, \varepsilon_y))}\,\textsc{Bind}}\,\textsc{Sko}_{\exists}}\,\textsc{Cong}$$

6.2 Soundness

To prove the soundness of the processing calculus we start by encoding the judgments in a well-understood theory of binders: the simply typed λ-calculus. A context and a term will be encoded together as a single λ-term. We call these somewhat nonstandard λ-terms *metaterms*. They are defined by the grammar

$$M ::= \boxed{t} \mid \lambda x.\, M \mid (\lambda \bar{x}_n.\, M)\, \bar{t}_n$$

where x_i and t_i are of the same sort for each $i \in [n]$. A metaterm is either a term t decorated with a box $\boxed{\ }$, a λ-abstraction, or the application of an n-tuple of terms to an uncurried λ-abstraction that simultaneously binds n distinct variables. We let $=_{\alpha\beta}$ denote syntactic equality modulo α- and β-equivalence (i.e., up to renaming of bound variables and reduction of applied λ-abstractions). We use the letters M, N, P to refer to metaterms. The notion of type is as expected for simply typed λ-terms: The type of $\boxed{t}$ is the sort of t; the type of $\lambda x.\, M$ is $\sigma \to \tau$, where σ is the sort of x and τ the type of M; and the type of $(\lambda \bar{x}_n.\, M)\, \bar{t}_n$ is the type of M. It is easy to see that all metaterms contain exactly one boxed term. $M[t]$ denotes a metaterm whose box contains t, and $M[N]$ denotes the same metaterm after its box has been replaced with the metaterm N.

Encoded judgments will have the form $M \simeq N$. The λ-abstractions and applications represent the context, whereas the box stores the term. To invoke the theory oracle $\models_{\mathcal{T}}$, we will need to *reify* equalities on metaterms—i.e., map them to equalities on terms. Let M, N be metaterms of type $\sigma_1 \to \cdots \to \sigma_n \to \sigma$. We define *reify*$(M \simeq N)$ as $\forall \bar{x}_n.\, t \simeq u$, where $M =_{\alpha\beta} \lambda x_1 \ldots \lambda x_n.\, \boxed{t}$ and $N =_{\alpha\beta} \lambda x_1 \ldots \lambda x_n.\, \boxed{u}$. Thanks to basic properties of the λ-calculus, t and u are always defined uniquely up to the names of the bound variables. For example, if $M = \lambda u.\, (\lambda v.\, \boxed{\mathsf{p}(v)})\, u$ and $N = \lambda w.\, \boxed{\mathsf{q}(w)}$, we have $M =_{\alpha\beta} \lambda x.\, \boxed{\mathsf{p}(x)}$ and $N =_{\alpha\beta} \lambda x.\, \boxed{\mathsf{q}(x)}$, and the reification of $M \simeq N$ is $\forall x.\, \mathsf{p}(x) \simeq \mathsf{q}(x)$.

The inference rules presented in Sect. 6.1 can now be encoded as follows. We refer to these new rules collectively as the *encoded inference system*:

$$\frac{}{M \simeq N}\ \textsc{Taut}_{\mathcal{T}} \quad \text{if } \models_{\mathcal{T}} \mathit{reify}(M \simeq N)$$

$$\frac{M \simeq N \quad N' \simeq P}{M \simeq P}\ \textsc{Trans} \quad \text{if } N =_{\alpha\beta} N' \qquad \frac{\big(M[t_i] \simeq N[u_i]\big)_{i=1}^{n}}{M[\mathsf{f}(\bar{t}_n)] \simeq N[\mathsf{f}(\bar{u}_n)]}\ \textsc{Cong}$$

$$\frac{M[\lambda y.\ (\lambda x.\ \varphi)\ y] \simeq N[\lambda y.\ \psi]}{M[Qx.\ \varphi] \simeq N[Qy.\ \psi]}\ \textsc{Bind} \quad \text{if } y \notin \mathrm{FV}(Qx.\ \varphi)$$

$$\frac{M[(\lambda x.\ \varphi)\ (\varepsilon x.\ \varphi)] \simeq N}{M[\exists x.\ \varphi] \simeq N}\ \textsc{Sko}_{\exists} \qquad \frac{M[(\lambda x.\ \varphi)\ (\varepsilon x.\ \neg\varphi)] \simeq N}{M[\forall x.\ \varphi] \simeq N}\ \textsc{Sko}_{\forall}$$

$$\frac{\big(M[r_i] \simeq N[s_i]\big)_{i=1}^{n} \quad M[(\lambda \bar{x}_n.\ t)\ \bar{s}_n] \simeq N[u]}{M[\text{let } \bar{x}_n \simeq \bar{r}_n \text{ in } t] \simeq N[u]}\ \textsc{Let} \quad \text{if } M[s_i] =_{\alpha\beta} N[s_i] \text{ for all } i \in [n]$$

Lemma 6.1. *If the judgment $M \simeq N$ is derivable using the encoded inference system with the theories $\mathcal{T}_1, \ldots, \mathcal{T}_n$, then $\models_{\mathcal{T}} \mathit{reify}(M \simeq N)$ with $\mathcal{T} = \mathcal{T}_1 \cup \cdots \cup \mathcal{T}_n \cup {\simeq} \cup \varepsilon_1 \cup \varepsilon_2 \cup \mathrm{let}$.*

Proof. By structural induction on the derivation of $M \simeq N$. For each inference rule, we assume that $\models_{\mathcal{T}} \mathit{reify}(M_i \simeq N_i)$ holds for each judgment $M_i \simeq N_i$ in the antecedent and show that $\models_{\mathcal{T}} \mathit{reify}(M \simeq N)$. Most of the cases implicitly depend on basic properties of the λ-calculus to reason about *reify*.

Case $\textsc{Taut}_{\mathcal{T}'}$: Trivial because $\mathcal{T}' \subseteq \mathcal{T}$ by definition of $\mathcal{T}$.

Cases Trans, Cong, and Bind: These follow from the theory of equality ($\simeq$).

Cases $\textsc{Sko}_{\exists}$, $\textsc{Sko}_{\forall}$, and Let: These follow from (ε_1) and (ε_2) or (let) and from the congruence of equality. □

A judgment $\Gamma \rhd t \simeq u$ is encoded as $L(\Gamma)[t] \simeq R(\Gamma)[u]$, where

$$\begin{aligned} L(\varnothing)[t] &= \boxed{t} & R(\varnothing)[u] &= \boxed{u} \\ L(x, \Gamma)[t] &= \lambda x.\ L(\Gamma)[t] & R(x, \Gamma)[u] &= \lambda x.\ R(\Gamma)[u] \\ L(\bar{x}_n \mapsto \bar{s}_n, \Gamma)[t] &= (\lambda \bar{x}_n.\ L(\Gamma)[t])\ \bar{s}_n & R(\bar{x}_n \mapsto \bar{s}_n, \Gamma)[u] &= R(\Gamma)[u] \end{aligned}$$

The L function encodes the substitution entries of Γ as λ-abstractions applied to tuples. Reducing the applied λ-abstractions effectively applies $\mathit{subst}(\Gamma)$. For example:

$$\begin{aligned} L(x \mapsto 7,\ x \mapsto \mathsf{g}(x))[x] &= (\lambda x.\ (\lambda x.\ \boxed{x})\ (\mathsf{g}(x)))\ 7 =_{\alpha\beta} \boxed{\mathsf{g}(7)} \\ L(x \mapsto 7,\ x,\ x \mapsto \mathsf{g}(x))[x] &= (\lambda x.\ \lambda x.\ (\lambda x.\ \boxed{x})\ (\mathsf{g}(x)))\ 7 =_{\alpha\beta} \lambda x.\ \boxed{\mathsf{g}(x)} \end{aligned}$$

For any derivable judgment $\Gamma \rhd t \simeq u$, the terms t and u must have the same sort, and the metaterms $L(\Gamma)[t]$ and $R(\Gamma)[u]$ must have the same type. Another property is that $L(\Gamma)[t]$ is of the form $M[t]$ for some M that is independent of t and similarly for $R(\Gamma)[u]$, motivating the suggestive brackets around L's and R's term argument.

Lemma 6.2. *Let $\bar{x}_n$ be the list of variables fixed by the context Γ in order of occurrence. Then $L(\Gamma)[t] =_{\alpha\beta} \lambda x_1. \ldots \lambda x_n. \boxed{\Gamma(t)}$.*

Proof. By induction on Γ.

Case $\varnothing$: $L(\varnothing)[t] = \boxed{t} = \boxed{\varnothing(t)}$.

Case x, Γ: Let $\bar{y}_n$ be the variables fixed by Γ.

$$\begin{aligned} L(x, \Gamma)[t] &= \lambda x.\, L(\Gamma)[t] \\ &=_{\alpha\beta} \lambda x. \lambda y_1. \ldots \lambda y_n. \boxed{\Gamma(t)} && \{\text{by the induction hypothesis}\} \\ &= \lambda x. \lambda y_1. \ldots \lambda y_n. \boxed{(x, \Gamma)(t)} && \{\text{by } (*)\} \end{aligned}$$

where $(*)$ is the property that $\mathit{subst}(\Gamma) = \mathit{subst}(x, \Gamma)$ for all x and Γ, which is easy to prove by structural induction on Γ.

Case $\bar{x}_n \mapsto \bar{s}_n$, Γ: Let $\bar{y}_n$ be the variables fixed by Γ, and let $\rho = \{\bar{x}_n \mapsto \bar{s}_n\}[\bar{y}_n \mapsto \bar{y}_n]$.

$$\begin{aligned} L(\bar{x}_n \mapsto \bar{s}_n, \Gamma)[t] &= (\lambda \bar{x}_n.\, L(\Gamma)[t])\ \bar{s}_n \\ &=_{\alpha\beta} (\lambda \bar{x}_n. \lambda y_1. \ldots \lambda y_n. \boxed{\Gamma(t)})\ \bar{s}_n && \{\text{by the induction hypothesis}\} \\ &=_{\alpha\beta} \lambda y_1. \ldots \lambda y_n. \boxed{\rho(\Gamma(t))} && \{\text{by } \beta\text{-reduction}\} \\ &= \lambda y_1. \ldots \lambda y_n. \boxed{(\bar{x}_n \mapsto \bar{s}_n, \Gamma)(t)} && \{\text{by } (**)\} \end{aligned}$$

where $(**)$ is the property that $\rho \circ \mathit{subst}(\Gamma) = \mathit{subst}(\bar{x}_n \mapsto \bar{s}_n, \Gamma)$ for all $\bar{x}_n$, $\bar{s}_n$, and Γ, which is easy to prove by structural induction on Γ. □

Lemma 6.3. *If the judgment $\Gamma \rhd t \simeq u$ is derivable using the original inference system, then $L(\Gamma)[t] \simeq R(\Gamma)[u]$ is derivable using the encoded inference system.*

Proof. By structural induction on the derivation of $\Gamma \rhd t \simeq u$.

Case Taut$_\mathcal{T}$: We have $\models_\mathcal{T} \Gamma(t) \simeq u$. Using Lemma 6.2, we can easily show that $\models_\mathcal{T} \Gamma(t) \simeq u$ is equivalent to $\models_\mathcal{T} \mathit{reify}(L(\Gamma)[t] \simeq R(\Gamma)[u])$, the side condition of the encoded Taut$_\mathcal{T}$ rule.

Case Bind: The encoded antecedent is $M[\lambda y.\ (\lambda x.\ \varphi)\ y] \simeq N[\lambda y.\ \psi]$ (i.e., $L(\Gamma, y, x \mapsto y)[\varphi] \simeq R(\Gamma, y, x \mapsto y)[\psi]$), and the encoded succedent is $M[Qx.\, \varphi] \simeq N[Qy.\, \psi]$. By the induction hypothesis, the encoded antecedent is derivable. Thus, by the encoded Bind rule, the encoded succedent is derivable.

Cases Cong, Sko$_\exists$, and Sko$_\forall$: Similar to Bind.

Case Trans: If $\Gamma(t) = t$, the substitution entries of Γ affect only variables that do not occur free in t. Hence, $R(\Gamma)[t] =_{\alpha\beta} L(\Gamma)[t]$, as required by the encoded Trans rule.

Case Let: Similar to Trans. □

Incidentally, the converse of Lemma 6.3 does not hold, since the encoded inference rules allow metaterms that contain applied λ-abstractions on the right-hand side of $\simeq$.

Theorem 6.4 (Soundness of Inferences)**.** *If the judgment* $\Gamma \rhd t \simeq u$ *is derivable using the original inference system with the theories* $\mathcal{T}_1, \ldots, \mathcal{T}_n$*, then the entailment* $\models_{\mathcal{T}} \Gamma(t) \simeq u$ *holds, with* $\mathcal{T} = \mathcal{T}_1 \cup \cdots \cup \mathcal{T}_n \cup \simeq \cup\, \varepsilon_1 \cup \varepsilon_2 \cup \mathrm{let}$.

Proof. This follows from Lemmas 6.1 and 6.3. The equivalence of $\models_{\mathcal{T}} \Gamma(t) \simeq u$ and

$$\models_{\mathcal{T}} \mathit{reify}(L(\Gamma)[t] \simeq R(\Gamma)[u])$$

can be established along the lines of case $\textsc{Taut}_{\mathcal{T}}$ of Lemma 6.3. □

6.3 A proof of concept checker

We developed a prototypical proof checker for the inference system above using Isabelle/HOL [NPW02], to convince ourselves that proofs generated according to it can easily be reconstructed.

The Isabelle/HOL proof assistant is based on classical higher-order logic (HOL) [GM93], a variant of the simply typed λ-calculus. Thanks to the availability of λ-terms, we could follow the lines of the encoded inference system of Sect. 6.2 to represent judgments in HOL. The proof checker is included in the development version of Isabelle.[6]

Derivations are represented by a recursive datatype in Standard ML, Isabelle's primary implementation language. A derivation is a tree whose nodes are labeled by rule names. Rule $\textsc{Taut}_{\mathcal{T}}$ additionally carries a theorem that represents the oracle $\models_{\mathcal{T}}$, and rules Trans and Let are labeled with the terms that occur only in the antecedent (t and $\bar{s}_n$). Terms and metaterms are translated to HOL terms, and judgments $M \simeq N$ are translated to HOL equalities $t \simeq u$, where t and u are HOL terms. Uncurried λ-applications are encoded using a polymorphic combinator $\mathsf{case}_\times : (\alpha \to \beta \to \gamma) \to \alpha \times \beta \to \gamma$; in Isabelle/HOL, $\lambda(x, y).\, t$ is syntactic sugar for $\mathsf{case}_\times\ (\lambda x.\, \lambda y.\, t)$. This scheme is iterated to support n-tuples, represented by nested pairs $(t_1, (\cdots (t_{n-1},\, t_n) \cdots))$.

To implement the inference rules, it is necessary to be able to locate any metaterm's box. There is an easy criterion: Translated metaterms are of the form $(\lambda \ldots) \ldots$ or $\mathsf{case}_\times \ldots$, which is impossible for a translated term. Because reconstruction is not verified, there are no guarantees that it will always succeed, but when it does, the result is certified by Isabelle's LCF-style inference kernel [GMW79]. We hard-coded a few dozen examples to test different cases, such as this one: Given the HOL terms

$$t = \neg\, \forall x.\, \mathsf{p} \wedge \exists x.\, \forall x.\, \mathsf{q}\ x\ x \text{ and } u = \neg\, \forall x.\, \mathsf{p} \wedge \exists x.\, \mathsf{q}\ (\varepsilon x.\, \neg\, \mathsf{q}\ x\ x)\ (\varepsilon x.\, \neg\, \mathsf{q}\ x\ x)$$

and the ML tree

$$\mathsf{N}\ (\mathsf{Cong}, [\mathsf{N}\ (\mathsf{Bind}, [\mathsf{N}\ (\mathsf{Cong}, [\mathsf{N}\ (\mathsf{Refl}, []), \mathsf{N}\ (\mathsf{Bind}, [\mathsf{N}\ (\mathsf{Sko\_All}, [\mathsf{N}\ (\mathsf{Refl}, [])])])])])])$$

[6] `http://isabelle.in.tum.de/repos/isabelle/file/00731700e54f/src/HOL/ex/veriT_Preprocessing.thy`

the reconstruction function returns the HOL theorem $t \simeq u$.

Chapter 7

Proof-producing contextual recursion

We propose a generic algorithm for term transformations, based on structural recursion. The algorithm is parameterized by a few simple plugin functions embodying the essence of the transformation. By combining compatible plugin functions, we can perform several transformations in one traversal. Transformations can depend on some context that encapsulates relevant information, such as bound variables, variable substitutions, and polarity. Each transformation can define its own notion of context that is threaded through the recursion.

The output is generated by a proof module that maintains a stack of derivation trees. The procedure *apply*(R, n, Γ, t, u) pops n derivation trees $\bar{\mathcal{D}}_n$ from the stack and pushes the tree

$$\frac{\mathcal{D}_1 \quad \cdots \quad \mathcal{D}_n}{\Gamma \rhd t \simeq u}\,\mathrm{R}$$

onto the stack.The plugin functions are responsible for invoking *apply* as appropriate.

7.1 The generic algorithm and its instantiations

The algorithm performs a depth-first postorder contextual recursion on the term to process. Subterms are processed first; then an intermediate term is built from the resulting subterms and is processed itself. The context Δ is updated in a transformation-specific way with each recursive call. It is abstract from the point of view of the algorithm.

The plugin functions are divided into two groups: *ctx-let*, *ctx-quant*, and *ctx-app* update the context when entering the body of a binder or when moving from a function symbol to one of its arguments; *build-let*, *build-quant*, *build-app*, and *build-var* return the processed term and produce the corresponding proof as a side effect.

function *process*(Δ, t)
  match t
    case x:
      return *build-var*(Δ, x)
    case $\mathsf{f}(\bar{t}_n)$:

$\bar{\Delta}'_n \leftarrow (\mathit{ctx\text{-}app}(\Delta, \mathsf{f}, \bar{t}_n, i))_{i=1}^n$
return $\mathit{build\text{-}app}\big(\Delta, \bar{\Delta}'_n, \mathsf{f}, \bar{t}_n, (\mathit{process}(\Delta'_i, t_i))_{i=1}^n\big)$
case $Qx.\ \varphi$:
$\Delta' \leftarrow \mathit{ctx\text{-}quant}(\Delta, Q, x, \varphi)$
return $\mathit{build\text{-}quant}(\Delta, \Delta', Q, x, \varphi, \mathit{process}(\Delta', \varphi))$
case let $\bar{x}_n \simeq \bar{r}_n$ in t':
$\Delta' \leftarrow \mathit{ctx\text{-}let}(\Delta, \bar{x}_n, \bar{r}_n, t')$
return $\mathit{build\text{-}let}(\Delta, \Delta', \bar{x}_n, \bar{r}_n, t', \mathit{process}(\Delta', t'))$

7.1.1 'Let' expansion

The first instance of the contextual recursion algorithm expands 'let' expressions and renames bound variables systematically to avoid capture. Skolemization and theory simplification, presented below, assume that this transformation has been performed.

The context consists of a list of fixed variables and variable substitutions, as in Sect. 6.1. The plugin functions are as follows:

function $\mathit{ctx\text{-}let}(\Gamma, \bar{x}_n, \bar{r}_n, t)$
return $\Gamma, \bar{x}_n \mapsto (\mathit{process}(\Gamma, r_i))_{i=1}^n$

function $\mathit{build\text{-}let}(\Gamma, \Gamma', \bar{x}_n, \bar{r}_n, t, u)$
$\mathit{apply}(\textsc{Let}, n+1, \Gamma, \text{let } \bar{x}_n \simeq \bar{r}_n \text{ in } t, u)$
return u

function $\mathit{ctx\text{-}quant}(\Gamma, Q, x, \varphi)$
$y \leftarrow$ fresh variable
return $\Gamma, y, x \mapsto y$

function $\mathit{build\text{-}quant}(\Gamma, \Gamma', Q, x, \varphi, \psi)$
$y \leftarrow \Gamma'(x)$
$\mathit{apply}(\textsc{Bind}, 1, \Gamma, Qx.\ \varphi, Qy.\ \psi)$
return $Qy.\ \psi$

function $\mathit{ctx\text{-}app}(\Gamma, \mathsf{f}, \bar{t}_n, i)$
return Γ

function $\mathit{build\text{-}app}(\Gamma, \bar{\Gamma}'_n, \mathsf{f}, \bar{t}_n, \bar{u}_n)$
$\mathit{apply}(\textsc{Cong}, n, \Gamma, \mathsf{f}(\bar{t}_n), \mathsf{f}(\bar{u}_n))$
return $\mathsf{f}(\bar{u}_n)$

function $\mathit{build\text{-}var}(\Gamma, x)$
$\mathit{apply}(\textsc{Refl}, 0, \Gamma, x, \Gamma(x))$
return $\Gamma(x)$

The *ctx-let* and *build-let* functions process 'let' expressions. In *ctx-let*, the substituted terms are processed further before they are added to a substitution entry in the context. In *build-let*, the LET rule is applied and the transformed term is returned. Analogously, the *ctx-quant* and *build-quant* functions rename quantified variables systematically. This ensures that any variables that arise in the range of the substitution specified by *ctx-let* will resist capture when the substitution is applied (cf. Example 6.4). Finally, the *ctx-app*, *build-app*, and *build-var* functions simply reproduce the term traversal in the generated proof; they perform no transformation-specific work.

Example 7.1. Following up on Example 6.1, assume $\varphi = \text{let } x \simeq \mathsf{a} \text{ in } \mathsf{p}(x, x)$. Given the above plugin functions, $\mathit{process}(\varnothing, \varphi)$ returns $\mathsf{p}(\mathsf{a}, \mathsf{a})$. It is instructive to study the evolution of the stack during the execution of *process*. First, in *ctx-let*, the term a is processed recursively; the call to *build-app* pushes a nullary CONG step with succedent $\triangleright\ \mathsf{a} \simeq \mathsf{a}$ onto the stack. Then the

term $\mathsf{p}(x, x)$ is processed. For each of the two occurrences of x, *build-var* pushes a Refl step onto the stack. Next, *build-app* applies a Cong step to justify rewriting under p: The two Refl steps are popped, and a binary Cong is pushed. Finally, *build-let* performs a Let inference with succedent $\triangleright\ \varphi \simeq \mathsf{p}(\mathsf{a}, \mathsf{a})$ to complete the proof: The two Cong steps on the stack are replaced by the Let step. The stack now consists of a single item: the derivation tree of Example 6.1.

7.1.2 Skolemization

Our second transformation, skolemization, assumes that 'let' expressions have been expanded and bound variables have been renamed apart. The context is a pair $\Delta = (\Gamma, p)$, where Γ is a context as defined in Sect. 6.1 and p is the polarity ($+$, $-$, or ?) of the term being processed. The main plugin functions are those that manipulate quantifiers:

function *ctx-quant*$((\Gamma, p), Q, x, \varphi)$
  if $(Q, p) \in \{(\exists, +), (\forall, -)\}$ **then**
    $\Gamma' \leftarrow \Gamma, x \mapsto$ *sko_term*(Γ, Q, x, φ)
  else
    $\Gamma' \leftarrow \Gamma, x$
  return (Γ', p)

function *build-quant*$((\Gamma, p), \Delta', Q, x, \varphi, \psi)$
  if $(Q, p) \in \{(\exists, +), (\forall, -)\}$ **then**
    apply$(\textsc{Sko}_Q, 1, \Gamma, Qx.\ \varphi, \psi)$
    return ψ
  else
    apply$(\textsc{Bind}, 1, \Gamma, Qx.\ \varphi, Qx.\ \psi)$
    return $Qx.\ \psi$

The polarity component of the context is updated by *ctx-app*, which is not shown. For example, *ctx-app*$((\Gamma, -), \neg, \varphi, 1)$ returns $(\Gamma, +)$, because if $\neg\,\varphi$ occurs negatively in a larger formula, then φ occurs positively. The plugin functions *build-app* and *build-var* are as for 'let' expansion.

Positive occurrences of $\exists$ and negative occurrences of $\forall$ are skolemized. All other quantifiers are kept as is. The *sko_term* function returns an applied Skolem function symbol following some reasonable scheme; for example, outer skolemization [NWCS01] creates an application of a fresh function symbol to all variables fixed in the context. To comply with the inference system, the application of $\textsc{Sko}_\exists$ or $\textsc{Sko}_\forall$ in *build-quant* instructs the proof module to systematically replace the Skolem term with the corresponding ε term when outputting the proof.

Example 7.2. Let $\varphi = \neg\,\forall x.\ \mathsf{p}(x)$. The call *process*$((\varnothing, +), \varphi)$ skolemizes φ into $\neg\,\mathsf{p}(\mathsf{c})$, where c is a fresh Skolem constant. The initial *process* call invokes *ctx-app* on $\neg$ as the second argument, making the context negative, thereby enabling skolemization of $\forall$. The substitution $x \mapsto \mathsf{c}$ is added to the context. Applying $\textsc{Sko}_\forall$ instructs the proof module to replace c with $\varepsilon x.\ \neg\,\mathsf{p}(x)$. The resulting derivation tree is as in Example 6.2.

7.1.3 Theory simplification

All kinds of theory simplification can be performed on formulas. We restrict our focus to a simple yet quite characteristic instance: the simplification of $u + 0$ and $0 + u$ to u. We assume that 'let' expressions have been expanded. The context is a list of fixed variables. The plugin functions

ctx-app and *build-var* are as for 'let' expansion (Sect. 7.1.1); the remaining ones are presented below:

function *ctx-quant*(Γ, Q, x, φ)
  return Γ, x
function *build-quant*(Γ, Γ', Q, x, φ, ψ)
  apply(BIND, 1, Γ, $Qx.\,\varphi$, $Qx.\,\psi$)
  return $Qx.\,\psi$

function *build-app*(Γ, $\bar{\Gamma}'_n$, f, $\bar{t}_n$, $\bar{u}_n$)
  apply(CONG, n, Γ, $\mathsf{f}(\bar{t}_n)$, $\mathsf{f}(\bar{u}_n)$)
  if $\mathsf{f}(\bar{u}_n)$ has form $u+0$ or $0+u$ **then**
    apply(TAUT$_+$, 0, Γ, $\mathsf{f}(\bar{u}_n)$, u)
    apply(TRANS, 2, Γ, $\mathsf{f}(\bar{t}_n)$, u)
    return u
  else
    return $\mathsf{f}(\bar{u}_n)$

The quantifier manipulation code, in *ctx-quant* and *build-quant*, is straightforward. The interesting function is *build-app*. It first applies the CONG rule to justify rewriting the arguments. Then, if the resulting term $\mathsf{f}(\bar{u}_n)$ can be simplified further into a term u, it performs a transitive chain of reasoning: $\mathsf{f}(\bar{t}_n) \simeq \mathsf{f}(\bar{u}_n) \simeq u$.

Example 7.3. Let $\varphi = \mathsf{k} + 1 \times 0 < \mathsf{k}$. Assuming that the framework has been instantiated with theory simplification for additive and multiplicative identity, invoking *process*($\varnothing$, φ) returns the formula $\mathsf{k} < \mathsf{k}$. The generated derivation tree is as in Example 6.3.

7.1.4 Combinations of transformations

Theory simplification can be implemented as a family of transformations, each member of which embodies its own set of theory-specific rewrite rules. If the union of the rewrite rule sets is confluent and terminating, a unifying implementation of *build-app* can apply the rules in any order until a fixpoint is reached. Moreover, since theory simplification modifies terms independently of the context, it is compatible with 'let' expansion and skolemization. This means that we can replace the *build-app* implementation from Sect. 7.1.1 or 7.1.2 with that of Sect. 7.1.3. In particular, this allows us to perform arithmetic simplification in the substituted terms of a 'let' expression in a single pass (cf. Example 6.1).

The combination of 'let' expansion and skolemization is less straightforward. Consider the formula $\varphi = \text{let } y \simeq \exists x.\,\mathsf{p}(x) \text{ in } y \to y$. When processing the subformula $\exists x.\,\mathsf{p}(x)$, we cannot (or at least should not) skolemize the quantifier, because it has no unambiguous polarity; indeed, the variable y occurs both positively and negatively in the 'let' expression's body. We can of course give up and perform two passes: The first pass expands 'let' expressions, and the second pass skolemizes and simplifies terms. The first pass also provides an opportunity to expand equivalences, which are problematic for skolemization.

There is also a way to perform all the transformations in a single instance of the framework. The most interesting plugin functions are *ctx-let* and *build-var*:

function *ctx-let*((Γ, p), $\bar{x}_n$, $\bar{r}_n$, t)
  for $i = 1$ **to** n **do**
    apply(REFL, 0, Γ, x_i, Γ(r_i))
  $\Gamma' \leftarrow \Gamma, \bar{x}_n \mapsto (\Gamma(r_i))_{i=1}^n$
  return (Γ', p)

function *build-var*((Γ, p), x)
  apply(REFL, 0, Γ, x, Γ(x))
  $u \leftarrow$ *process*((Γ, p), Γ(x))
  apply(TRANS, 2, Γ, Γ(x), u)
  return u

In contrast with the corresponding function for 'let' expansion (Sect. 7.1.1), *ctx-let* does not process the terms $\bar{r}_n$, which is reflected by the n applications of REFL, and it must thread through polarities. The call to *process* is in *build-var* instead, where it can exploit the more precise polarity information to skolemize the formula.

The *build-let* function is essentially as before. The *ctx-quant* and *build-quant* functions are as for skolemization (Sect. 7.1.2), except that the **else** cases rename bound variables apart (Sect. 7.1.1). The *ctx-app* function is as for skolemization, whereas *build-app* is as for theory simplification (Sect. 7.1.3).

For the formula φ given above, *process*((∅, +), φ) returns $(\exists x.\, \mathsf{p}(x)) \to \mathsf{p}(\mathsf{c})$, where c is a fresh Skolem constant. The substituted term $\exists x.\, \mathsf{p}(x)$ is put unchanged into the substitution used to expand the 'let' expression. It is processed each time y arises in the body $y \longrightarrow y$. The positive occurrence is skolemized; the negative occurrence is left as is. Using caching and a DAG representation of derivations, we can easily avoid the duplicated work that would arise if y occurred several times with positive polarity.

7.1.5 Scope and limitations

Other possible instances of contextual recursion are the clause normal form (CNF) transformation and the elimination of quantifiers using one-point rules. CNF transformation is an instance of rewriting of Boolean formulas and can be justified by a $\text{TAUT}_{\mathsf{Bool}}$ rule. Tseytin transformation can be supported by representing the introduced constants by the formulas they represent, similarly to our treatment of Skolem terms. One-point rules—e.g., the transformation of $\forall x.\, x \simeq \mathsf{a} \longrightarrow \mathsf{p}(x)$ into $\mathsf{p}(\mathsf{a})$—are similar to 'let' expansion and can be represented in much the same way in our framework. The rules for eliminating universal and existential quantifiers are as follows:

$$\frac{\Gamma \rhd s \simeq t \quad \Gamma, x \mapsto t \rhd \varphi \simeq \psi}{\Gamma \rhd (\forall x.\, x \simeq s \longrightarrow \varphi) \simeq \psi}\ 1\text{PT}_\forall \quad \text{if } x \notin \text{FV}(s) \text{ and } \Gamma(t) = t$$

$$\frac{\Gamma \rhd s \simeq t \quad \Gamma, x \mapsto t \rhd \varphi \simeq \psi}{\Gamma \rhd (\exists x.\, x \simeq s \wedge \varphi) \simeq \psi}\ 1\text{PT}_\exists \quad \text{if } x \notin \text{FV}(s) \text{ and } \Gamma(t) = t$$

The plugin functions used by *process* would also be similar as those for 'let' expansion, except that detecting the assignment at *ctx-quant* requires examining the body of the quantified formula to determine whether the one-point rule is applicable.

Some transformations, such as symmetry breaking [DFMP11] and rewriting based on global assumptions, require a global analysis of the problem that cannot be captured by local substitution of equals for equals. They are beyond the scope of the framework. Other transformations,

such as simplification based on associativity and commutativity of function symbols, require traversing the terms to be simplified when applying the rewriting. Since *process* visits terms in postorder, the complexity of the simplifications would be quadratic, while a processing that applies depth-first preorder traversal can perform the simplifications with a linear complexity. Hence, applying such transformations optimally is also outside the scope of the framework.

7.2 Correctness

We turn to the contextual recursion algorithm that generates derivations in that system. The first question is, *Are the derivation trees valid?* In particular, it is not obvious from the code that the side conditions of the inference rules are always satisfied. First, we need to introduce some terminology. A term is *shadowing-free* if nested binders always bind variables with different names; for example, the term $\forall x.\ (\forall y.\ \mathsf{p}(x, y)) \wedge (\forall y.\ \mathsf{q}(y))$ is shadowing-free, while $\forall x.\ (\forall x.\ \mathsf{p}(x, y)) \wedge (\forall y.\ \mathsf{q}(y))$ is not. The set of variables *fixed* by Γ is written $\mathit{fix}(\Gamma)$, and the set of variables *replaced* by Γ is written $\mathit{repl}(\Gamma)$. They are defined as follows:

$$\begin{aligned} \mathit{fix}(\varnothing) &= \{\} & \mathit{repl}(\varnothing) &= \{\} \\ \mathit{fix}(\Gamma, x) &= \{x\} \cup \mathit{fix}(\Gamma) & \mathit{repl}(\Gamma, x) &= \mathit{repl}(\Gamma) \\ \mathit{fix}(\Gamma, \bar{x}_n \mapsto \bar{s}_n) &= \mathit{fix}(\Gamma) & \mathit{repl}(\Gamma, \bar{x}_n \mapsto \bar{s}_n) &= \{x_i \mid s_i \neq x_i\} \cup \mathit{repl}(\Gamma) \end{aligned}$$

Trivial substitutions $x \mapsto x$ are ignored, since they have no effect. The set of variables *introduced* by Γ is defined by $\mathit{intr}(\Gamma) = \mathit{fix}(\Gamma) \cup \mathit{repl}(\Gamma)$. A context Γ is *consistent* if all the fixed variables are mutually distinct and the two sets of variables are disjoint—i.e., $\mathit{fix}(\Gamma) \cap \mathit{repl}(\Gamma) = \{\}$.

A judgment $\Gamma \rhd t \simeq u$ is *canonical* if Γ is consistent, $\mathrm{FV}(t) \subseteq \mathit{intr}(\Gamma)$, $\mathrm{FV}(u) \subseteq \mathit{fix}(\Gamma)$, and $\mathrm{BV}(u) \cap \mathit{intr}(\Gamma) = \{\}$. The *canonical inference system* is a variant of the system of Sect. 6.1 in which all judgments are canonical and rules Trans, Bind, and Let have no side conditions.

Lemma 7.1. *Any inference in the canonical inference system is also an inference in the original inference system.*

Proof. It suffices to show that the side conditions of the original rules are satisfied.

Case Trans: Since the first judgment in the antecedent is canonical, $\mathrm{FV}(t) \subseteq \mathit{fix}(\Gamma)$. By consistency of Γ, we have $\mathit{fix}(\Gamma) \cap \mathit{repl}(\Gamma) = \{\}$. Hence, $\mathrm{FV}(t) \cap \mathit{repl}(\Gamma) = \{\}$ and therefore $\Gamma(t) = t$.

Case Bind: Since the succedent is canonical, we have that (1) $\mathrm{FV}(Qx.\ \varphi) \subseteq \mathit{intr}(\Gamma)$ and (2) $\mathrm{BV}(Qy.\ \psi) \cap \mathit{intr}(\Gamma) = \{\}$ hold. From (2), we deduce $y \notin \mathit{intr}(\Gamma)$. Hence, by (1), we get $y \notin \mathrm{FV}(Qx.\ \varphi)$.

Case Let: Similar to Trans. □

Theorem 7.2 (Total Correctness of Recursion)**.** *For the instances presented in Sects. 7.1.1 to 7.1.3, the contextual recursion algorithm always produces correct proofs.*

Proof. The algorithm terminates because *process* is called initially on a finite input and recursive calls always have smaller inputs.

For the proof of partial correctness, only the Γ part of the context is relevant. We will write *process*(Γ, t) even if the first argument actually has the form (Γ, p) for skolemization. The pre- and postconditions of a *process*(Γ, t) call that returns the term u are

PRE1 Γ is consistent;	POST1 u is shadowing-free;
PRE2 $\mathrm{FV}(t) \subseteq \mathit{intr}(\Gamma)$;	POST2 $\mathrm{FV}(u) \subseteq \mathit{fix}(\Gamma)$;
PRE3 $\mathrm{BV}(t) \cap \mathit{fix}(\Gamma) = \{\}$;	POST3 $\mathrm{BV}(u) \cap \mathit{intr}(\Gamma) = \{\}$.

For skolemization and simplification, we may additionally assume that bound variables have been renamed apart by 'let' expansion, and hence that the term t is shadowing-free.

The initial call *process*($\varnothing$, t) trivially satisfies the preconditions on an input term t that contains no free variable. We must show that the preconditions for each recursive call *process*(Γ', t') are satisfied and that the postconditions hold at the end of *process*(Γ, t).

PRE1 (Γ' is consistent): First, we show that the fixed variables are mutually distinct. For 'let' expansion, all fixed variables are fresh. For skolemization and simplification, a precondition is that the input is shadowing-free. For any two fixed variables in Γ', the input formula must contain two quantifiers, one in the scope of the other. Hence, the variables must be distinct. Second, we show that $\mathit{fix}(\Gamma') \cap \mathit{repl}(\Gamma') = \{\}$. For 'let' expansion, all fixed variables are fresh. For skolemization, the condition is a direct consequence of the precondition that the input is shadowing-free. For simplification, we have $\mathit{repl}(\Gamma') = \{\}$.

PRE2 ($\mathrm{FV}(t') \subseteq \mathit{intr}(\Gamma')$): We have $\mathrm{FV}(t) \subseteq \mathit{intr}(\Gamma)$. The desired property holds because the *ctx-let* and *ctx-quant* functions add to the context any bound variables that become free when entering the body t' of a binder.

PRE3 ($\mathrm{BV}(t') \cap \mathit{fix}(\Gamma') = \{\}$): The only function that fixes variable is *ctx-quant*. For 'let' expansion, all fixed variables are fresh. For skolemization and simplification, the condition is a consequence of the shadowing-freedom of the input.

POST1 (u is shadowing-free): The only function that builds quantifiers is *build-quant*. The *process*(Γ', φ) call that returns the processed body ψ of the quantifier is such that $y \in \mathit{intr}(\Gamma')$ in the 'let' expansion case and $x \in \mathit{intr}(\Gamma')$ in the other two cases. The induction hypothesis ensures that ψ is shadowing-free and $\mathrm{BV}(\psi) \cap \mathit{intr}(\Gamma') = \{\}$; hence, the result of *build-quant* (i.e., $Qy.\psi$ or $Qx.\psi$) is shadowing-free. Quantifiers can also emerge when applying a substitution in *build-var*. This can happen only if *ctx-let* has added a substitution entry to the context, in which case the substituted term is the result of a call to *process* and is thus shadowing-free.

POST2 ($\mathrm{FV}(u) \subseteq \mathit{fix}(\Gamma)$): In most cases, this condition follows directly from the induction hypothesis POST2. The only case where a variable appears fixed in the context Γ' of a recursive call to *process* and not in Γ is when processing a quantifier, and then that variable is bound in the result. For variable substitution, it suffices to realize that the context in which the substituted

term is created (and which fixes all the free variables of the term) is a prefix of the context when the substitution occurs.

POST3 ($\mathrm{BV}(u) \cap \mathit{intr}(\Gamma) = \{\}$): In most cases, this condition follows directly from the induction hypothesis POST3: For every recursive call, $\mathit{intr}(\Gamma) \subseteq \mathit{intr}(\Gamma')$. Two cases require attention. For 'let' expansion, a variable may be replaced by a term with bound variables. Then the substituted term only contains variables that do not occur in the input. The variables introduced by Γ will be other fresh variables or variables from the input. The second case is when a quantified formula is built. For 'let' expansion, a fresh variable is used. For skolemization and simplification, we have $\mathrm{BV}(Qx.\, \varphi) \cap \mathit{fix}(\Gamma) = \{\}$ (PRE3); hence $x \notin \mathit{fix}(\Gamma)$. Finally, we must show that $x \notin \mathit{repl}(\Gamma)$; this is a consequence of the shadowing-freedom of the input.

It is easy to see that each *apply* call generates a rule with an antecedent and a succedent of the right form, ignoring the rules' side conditions. Moreover, all calls to *apply* generate canonical judgments thanks to the pre- and postconditions proved above. Correctness follows from Lemma 7.1. □

Observation 7.3 (Complexity of Recursion)**.** For the instances presented in Sects. 7.1.1 to 7.1.3, the '*process*' function is called at most once on every subterm of the input.

Justification. It suffices to notice that a call to $\mathit{process}(\Delta,\, t)$ induces at most one call for each of the subterms in t.

As a corollary, if all the operations performed in *process* excluding the recursive calls can be accomplished in constant time, the algorithm has linear-time complexity with respect to the input. There exist data structures for which the following operations take constant time: extending the context with a fixed variable or a substitution, accessing direct subterms of a term, building a term from its direct subterms, choosing a fresh variable, applying a context to a variable, checking if a term matches a simple template, and associating the parameters of the template with the subterms. Thus, it is possible to have a linear-time algorithm for 'let' expansion and simplification.

On the other hand, construction of Skolem terms is at best linear in the size of the context and of the input formula in *process*. Hence, skolemization is at best quadratic in the worst case. This is hardly surprising because in general, the formula $\forall x_1.\, \exists y_1. \ldots \forall x_n.\, \exists y_n.\, \varphi[\bar{x}_n,\, \bar{y}_n]$, whose size is proportional to n, is translated to $\forall x_1. \ldots \forall x_n.\, \varphi[\bar{x}_n,\, \mathsf{f}_1(\bar{x}_1), \mathsf{f}_2(\bar{x}_2), \ldots, \mathsf{f}_n(\bar{x}_n)]$, whose size is quadratic in n.

Observation 7.4 (Overhead of Proof Generation)**.** For the instances presented in Sects. 7.1.1 to 7.1.3 , the number of calls to the '*apply*' procedure is proportional to the number of subterms in the input.

Justification. This is a corollary of Observation 7.3, since the number of *apply* calls per *process*

call is bounded by a constant (3, in *build-app* for simplification).

Notice that all arguments to *apply* must be computed regardless of the *apply* calls. If an *apply* call takes constant time, the proof generation overhead is linear in the size of the input. To achieve this performance, it is necessary to use sharing to represent contexts and terms in the output; otherwise, each call to *apply* might itself be linear in the size of its arguments, resulting in a nonlinear overhead on the generation of the entire proof.

Observation 7.5 (Cost of Proof Checking)**.** Checking an inference step can be performed in constant time if checking the side condition takes constant time.

Justification. The inference rules involve only shallow conditions on contexts and terms, except in the side conditions. Using suitable data structures with maximal sharing, the contexts and terms can be checked in constant time.

The above statement may appear weak, since checking the side conditions might itself be linear, leading to a cost of proof checking that can be at least quadratic in the size of the proof (measured as the number of symbols that represent it). Fortunately, most of the side conditions can be checked efficiently. For example, for simplification (Sect. 7.1.3), the BIND rule is always applied with $x = y$, which implies the side condition $y \notin \mathrm{FV}(Qx.\, \varphi)$; and since no other rule contributes to the substitution, $\mathit{subst}(\Gamma)$ is the identity. Thus, simplification proofs can be checked in linear time.

Moreover, certifying a proof by checking each step locally is not the only possibility. An alternative is to use an algorithm similar to the *process* function to check a proof in the same way as it has been produced. Such an algorithm can exploit sophisticated invariants on the contexts and terms.

7.3 Implementation

We implemented the contextual recursion algorithm and the transformations described in Sect. 7.1 in the SMT solver veriT [BdODF09], replacing large parts of the previous non-proof-producing, hard-to-maintain code. Even though it offers more functionality (proof generation), the preprocessing module is about 20% smaller than before and consists of about 3000 lines of code. There are now only two traversal functions instead of 10. This is, for us, a huge gain in maintainability. Besides, as our experimental data attests, we had no significant detrimental impact on the solving times or success rates, despite the production of detailed proofs.

Proofs

Previously, veriT provided detailed proofs for the resolution steps performed by the SAT solver and the lemmas added by the theory solvers and instantiation module. All transformations per-

formed in preprocessing steps were represented in the proof in a very coarse manner, amounting to gaps in the proof. For example, when 'let' expressions were expanded in a formula, the only information present in the proof would be the formula before and after 'let' expansion.

When extending veriT to generate more detailed proofs, we were able to reuse its existing proof module and proof format [BFT11]. A proof is a list of inferences, each of which consists of an identifier (e.g., .c0), the name of the rule, the identifiers of the dependencies, and the derived clause. The use of identifiers makes it possible to represent proofs as DAGs. We extended the format with the inference rules of Sect. 6.1. The rules that augment the context take a sequence of inferences—a *subproof*—as a justification. The subproof occurs within the scope of the extended context. Following this scheme, the skolemization proof for the formula $\neg\,\forall x.\ \mathsf{p}(x)$ from Example 6.2 is presented as

$$
\begin{aligned}
&(.\mathrm{c0}\ (\mathsf{Sko\_All}\ :\mathrm{conclusion}\ ((\forall x.\ \mathsf{p}(x)) \simeq \mathsf{p}(\varepsilon x.\ \neg\,\mathsf{p}(x)))\\
&\qquad :\mathrm{args}\ (x \mapsto (\varepsilon x.\ \neg\,\mathsf{p}(x)))\\
&\qquad :\mathrm{subproof}\ ((.\mathrm{c1}\ (\mathsf{Refl}\ :\mathrm{conclusion}\ (x \simeq (\varepsilon x.\ \neg\,\mathsf{p}(x)))))\\
&\qquad\qquad (.\mathrm{c2}\ (\mathsf{Cong}\ :\mathrm{clauses}\ (.\mathrm{c1})\ :\mathrm{conclusion}\ (\mathsf{p}(x) \simeq \mathsf{p}(\varepsilon x.\ \neg\,\mathsf{p}(x))))))))\\
&(.\mathrm{c3}\ (\mathsf{Cong}\ :\mathrm{clauses}\ (.\mathrm{c0})\ :\mathrm{conclusion}\ ((\neg\,\forall x.\ \mathsf{p}(x)) \simeq \neg\,\mathsf{p}(\varepsilon x.\ \neg\,\mathsf{p}(x)))))
\end{aligned}
$$

Formerly, no details of these transformations would be recorded. The proof would have contained only the original formula and the skolemized result, regardless of how many quantifiers appeared in the formula.

In contrast with the abstract proof module described in Sect. 7.1, veriT leaves Refl steps implicit for judgments of the form $\Gamma \rhd t \simeq t$. The other inference rules are generalized to cope with missing Refl judgments. In addition, when printing proofs, the proof module can automatically replace terms in the inferences with some other terms. This is necessary for transformations such as skolemization and 'if–then–else' elimination. We must apply a substitution in the replaced term if the original term contains variables. In veriT, efficient data structures are available to perform this.

Transformations

The implementation of contextual recursion uses a single global context, augmented before processing a subterm and restored afterwards. The context consists of a set of fixed variables, a substitution, and a polarity. In our setting, the substitution satisfies the side conditions by construction. If the context is empty, the result of processing a subterm is cached. For skolemization, a separate cache is used for each polarity. No caching is attempted under binders.

Invoking *process* on a term returns the identifier of the inference at the root of its transformation proof in addition to the processed term. These identifiers are threaded through the recursion to connect the proof. The proofs produced by instances of contextual recursion are inserted into the larger resolution proof produced by veriT. This is achieved through an inference of the form

$$\dfrac{\varphi \quad \mathcal{D} \quad \dfrac{}{\neg(\varphi \simeq \psi) \vee \neg\varphi \vee \psi}\,\textsc{Taut}_{\simeq}}{\psi}\,\textsc{Resolve}$$

where φ is the original formula, ψ is the processed formula, and $\mathcal{D}$ is a derivation of $\rhd \varphi \simeq \psi$. The derivation $\mathcal{D}$ may itself depend on instances of rule $\textsc{Taut}_{\mathcal{T}}$, each with its own proof of the side condition that must also be included in the overall proof.

Transformations performing theory simplification were straightforward to port to the new framework: Their *build-app* functions simply apply rewrite rules until a fixpoint is reached. Porting transformations that interact with binders required special attention in handling the context and producing proofs. Fortunately, most of these aspects are captured by the inference system and the abstract contextual recursion framework, where they can be studied independently of the implementation.

Some transformations are performed outside of the framework. Proofs of CNF transformation are expressed using the inference rules of veriT's underlying SAT solver, so that any tool that can reconstruct SAT proofs can also reconstruct these proofs. Simplification based on associativity and commutativity of function symbols is implemented as a dedicated procedure, for efficiency reasons (Sect. 7.1.5). It currently produces coarse-grained proofs.

Evaluation

To evaluate the impact of the new contextual recursion algorithm and of producing detailed proofs, we compare the performance of different configurations of veriT. Our experimental data is available online.[1] We distinguish three configurations. Basic only applies transformations for which the old code provided some (coarse-grained) proofs. Extended also applies transformations for which the old code did not provide any proofs, whereas the new code provides detailed proofs. Complete applies all transformations available, regardless of whether they produce proofs.

More specifically, Basic applies the transformations for 'let' expansion, skolemization, elimination of quantifiers based on one-point rules, elimination of 'if–then–else', theory simplification for rewriting n-ary symbols as binary, and elimination of equivalences and exclusive disjunctions with quantifiers in subterms. Extended adds Boolean and arithmetic simplifications to the transformations performed by Basic. Complete performs global rewriting simplifications and symmetry breaking in addition to the transformations in Extended.

The evaluation was carried out on two main sets of benchmarks from SMT-LIB [BFT15][2]: the 20 916 benchmarks, labeled as satisfiable or unsatisfiable, in the quantifier-free (QF) categories QF_ALIA, QF_AUFLIA, QF_IDL, QF_LIA, QF_LRA, QF_RDL, QF_UF, QF_UFIDL, QF_UFLIA, and QF_UFLRA; and the 30 250 benchmarks labeled as unsatisfiable in the non-QF categories AUFLIA, AUFLIRA, UF, UFIDL, UFLIA, and UFLRA. The categories with

[1] `http://matryoshka.gforge.inria.fr/pubs/processing/`
[2] As of February 2017.

bit vectors and nonlinear arithmetic are not supported by veriT. Since veriT cannot produce models for formulas with quantifiers, only unsatisfiable problems were considered for this kind of benchmarks.

Our experiments were conducted on servers equipped with two Intel Xeon E5-2630 v3 processors, with eight cores per processor, and 126 GB of memory. Each run of the solver uses a single core. The time limit was set to 30 s, a reasonable value for interactive use within a proof assistant.

The tables below indicate the number of benchmark problems solved by each configuration for the quantifier-free and non-quantifier-free benchmarks:

	Without proofs		With proofs	
QF	Old code	New code	Old code	New code
Basic	13 489	13 496	13 360	13 352
Extended	13 539	13 537	N/A	13 414
Complete	13 826	13 819	N/A	N/A

	Without proofs		With proofs	
Non-QF	Old code	New code	Old code	New code
Basic	28 746	28 762	28 744	28 766
Extended	28 785	28 852	N/A	28 857
Complete	28 759	28 794	N/A	N/A

These results indicate that the new generic contextual recursion algorithm and the production of detailed proofs do not impact performance negatively compared with the old code and coarse-grained proofs. Moreover, allowing Boolean and arithmetic simplifications leads to some improvements. We expect that generating proofs for the global transformations would lead to substantial improvements on quantifier-free problems. On benchmarks with quantifiers, as the experimental results indicate (e.g. the new code solves slightly more problems than the old one in the proof-producing configurations), it is very hard to predict the impact of novel techniques due to the nature of heuristic instantiation. This way, marginal changes such as these observed do not weaken our claim that there is no negative impact on performance by the production of detailed proofs.

Chapter 8

Conclusions

Our first contribution in this thesis has been the introduction of CCFV, a decision procedure for E-ground (dis)unification. We have shown how the main instantiation techniques of SMT solving may be based on it. We also discussed how to efficiently implement CCFV in a CDCL($\mathcal{T}$) solver, a crucial aspect in our field of work. Our experimental evaluation shows that CCFV leads to significant improvements in the solvers CVC4 and veriT, making the former surpass the state-of-the-art in instantiation based SMT solving and the latter competitive in several benchmark libraries. The calculi presented are very general, allowing for different strategies and optimizations, as previously discussed.

There are three directions in which the work presented here can be improved: the CCFV solving, the reach of CCFV, and the instantiation techniques built on top of it. A direction for improving the solving is to use *lemma learning* in the depth-first CCFV, in a similar manner as SAT solvers do. When a branch fails to produce a solution and is discarded, analyzing the literals which led to the conflict can allow *backjumping* rather than simple backtracking, thus further reducing the solution search space. The *Complementary Congruence Closure* introduced by Backeman and Rümmer [BR15a] could be extended to perform such an analysis. More generally, we believe that making CCFV incremental would allow significant performance gains, as it does for theory solvers. A starting point would be to build on techniques such as those described by de Moura and Bjørner [dMB07] for performing incremental E-matching in Z3. The reach of CCFV can be extended by going beyond equational first-order logic. To support other theories the calculi have to be augmented with rules for the new interpreted symbols. To handle linear arithmetic, for instance, CCFV has to be able to perform AC1 unification (unification modulo associativity, commutativity and additive identity). Practical implementations will also require the ground solver to provide efficient entailment checks with relation to the new interpreted symbols, as it does for equality. Moreover, since decidability is easily lost when combining theories in quantified logic, a good balance must be achieved between decision procedures for specific fragments, such as the essentially uninterpreted fragment described by Ge and de Moura [GdM09], and effective heuristics for incomplete settings. Besides theory reasoning, higher-order unification could also be tackled by CCFV. The calculi would need to account for partial applications, functional

variables, and lambda abstractions. This would allow SMT solvers to natively handle higher-order problems, thus avoiding completeness and performance issues with clumsy encodings into first-order logic. Yet another possible extension of CCFV is to handle rigid E-unification, so it could be integrated into tableaux and sequent calculi based techniques such as BREU [BR15b] or conflict resolution [SP16]. This amounts to having non-ground equalities in E, which means significantly changing how CCFV works now. It would, however, allow integrating an efficient goal-oriented procedure into calculi that require solving E-unification problems.

CCFV can be seen as a first step towards a "theory solver" for quantified formulas to be integrated into a CDCL($\mathcal{T}$) solver. Instantiation techniques built on top of CCFV would evaluate the candidate model with quantified formulas and:

i) provide a finite-model when it exists,

ii) derive instances refuting the candidate when they exist,

iii) propagate relevant instantiations and

iv) be incremental with relation to propositional assignments.

The incrementality depends on the implementation of CCFV. Conflict based instantiation partially achieves (ii) and (iii), while trigger based instantiation fits into (iii) and model based instantiation somewhat into (i) and (ii). How to extend the instantiation techniques in order to fully accomplish these goals remains an open problem. A complete conflict based instantiation technique, at least with relation to equational first-order logic, could be achieved by considering quantified formulas simultaneously instead of independently, i.e. finding substitutions σ such that

$$E \models \neg\psi_1\sigma \vee \cdots \vee \neg\psi_m\sigma, \text{ with } \mathcal{Q} = \{\forall \bar{x}_{n_1}.\, \psi_1, \ldots, \forall \bar{x}_{n_m}.\, \psi_m\}$$

Techniques from tableaux and superposition calculi, which natively consider quantified formulas simultaneously, could be combined with CCFV for solving the entailment above.

Our second contribution was to introduce a framework to represent and generate proofs of formula processing and its implementation in veriT and Isabelle/HOL. The framework centralises the subtle issue of manipulating bound variables and substitutions soundly and efficiently, and it is flexible enough to accommodate many interesting transformations. Although it was implemented in an SMT solver, there appears to be no intrinsic limitation that would prevent its use in other kinds of first-order, or even higher-order, automatic provers. The framework covers many preprocessing techniques and can be part of a larger toolbox. It allows veriT, whose detailed proofs have been one of its defining features, to produce more detailed justifications than ever. However, there are still some global transformations for which the proofs are nonexistent or leave much to be desired. In particular, supporting rewriting based on global assumptions would be essential for proof-producing inprocessing, and symmetry breaking would be interesting in its own right. To support these transformations our notion of context would have to be extended, as well as the operations that are allowed to be performed on terms, in such a way that we

can go beyond the replacement of equals by equals. The framework can also be extended to handle different language constructs, such as lambda abstractions. Given the generality of the processing calculus and the proof-producing algorithm, operations such as beta-reduction can be easily integrated. The latter, for instance, is merely a variation of 'let' expansion.

Our focus here has been in the production of detailed proofs, aiming to guarantee proof checking to have reasonable complexity and thus facilitate effective implementations of proof checkers. A promising direction is also to reconstruct the detailed proofs in proof assistants, which provide trustworthy, machine-checkable formal proofs of theorems. A general challenge is to perform within the proof assistants the necessary reasoning to check the proofs. For some theories, such as non-linear arithmetic, this can be quite challenging. Once possible, however, fully reconstructing proofs into proof assistants would greatly increase the overall confidence on the results of automatic solvers, thus avoiding the need to trust software that mostly rely on testing to perform sound logic reasoning.

In conclusion, we believe that our contributions are beneficial to the state-of-the-art of SMT solving (and consequentially also of first-order theorem proving). Moreover, given that both the frameworks presented are quite general and extensible, we expect them to serve as the starting point for several more techniques for instantiation and proof production in SMT solving, pushing solvers into tackling ever more expressive problems and producing more trustworthy results.

Bibliography

[AKW09] Ernst Althaus, Evgeny Kruglov, and Christoph Weidenbach. "Superposition Modulo Linear Arithmetic SUP(LA)". In: *Frontiers of Combining Systems (FroCoS)*. Ed. by Silvio Ghilardi and Roberto Sebastiani. Vol. 5749. Lecture Notes in Computer Science. Springer, 2009, pp. 84–99.

[AFG+11] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. "A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses". In: *Certified Programs and Proofs*. Ed. by Jean-Pierre Jouannaud and Zhong Shao. Vol. 7086. Lecture Notes in Computer Science. Springer, 2011, pp. 135–150.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. New York, NY, USA: Cambridge University Press, 1998.

[BS01] Franz Baader and Wayne Snyder. "Unification Theory". In: *Handbook of Automated Reasoning*. Ed. by John Alan Robinson and Andrei Voronkov. Elsevier and MIT Press, 2001, pp. 445–532.

[BG94] Leo Bachmair and Harald Ganzinger. "Rewrite-Based Equational Theorem Proving with Selection and Simplification". In: *Journal of Logic and Computation* 4.3 (1994), pp. 217–247.

[BGW94] Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. "Refutational theorem proving for hierarchic first-order theories". In: *Applicable Algebra in Engineering, Communication and Computing* 5.3 (1994), pp. 193–212.

[BTV03] Leo Bachmair, Ashish Tiwari, and Laurent Vigneron. "Abstract Congruence Closure". English. In: *Journal of Automated Reasoning* 31.2 (2003), pp. 129–168.

[BR15a] Peter Backeman and Philipp Rümmer. "Efficient Algorithms for Bounded Rigid E-unification". In: *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*. Ed. by Hans de Nivelle. Vol. 9323. Lecture Notes in Computer Science. Springer, 2015, pp. 70–85.

[BR15b] Peter Backeman and Philipp Rümmer. "Theorem Proving with Bounded Rigid E-Unification". In: *Proc. Conference on Automated Deduction (CADE)*. Ed. by Amy Felty and Aart Middeldorp. Vol. 9195. Lecture Notes in Computer Science. Springer, 2015.

[BRK+15] Kshitij Bansal, Andrew Reynolds, Tim King, Clark Barrett, and Thomas Wies. "Deciding Local Theory Extensions via E-matching". English. In: *Computer Aided Verification (CAV)*. Ed. by Daniel Kroening and Corina S. Păsăreanu. Vol. 9207. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 87–105.

[Bar16] Haniel Barbosa. "Efficient Instantiation Techniques in SMT (Work In Progress)". In: *Practical Aspects of Automated Reasoning (PAAR)*. Ed. by Pascal Fontaine, Stephan Schulz, and Josef Urban. Vol. 1635. CEUR Workshop Proceedings. CEUR-WS.org, 2016, pp. 1–10.

[BBF17] Haniel Barbosa, Jasmin Christian Blanchette, and Pascal Fontaine. "Scalable Fine-Grained Proofs for Formula Processing". In: *Proc. Conference on Automated Deduction (CADE)*. Ed. by Leonardo de Moura. Vol. 10395. Lecture Notes in Computer Science. Springer, 2017, pp. 398–412.

[BFR17] Haniel Barbosa, Pascal Fontaine, and Andrew Reynolds. "Congruence Closure with Free Variables". In: *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. Ed. by Axel Legay and Tiziana Margaria. Vol. 10206. Lecture Notes in Computer Science. 2017, pp. 214–230.

[BW05] Henk Barendregt and Freek Wiedijk. "The challenge of computer mathematics". In: *Philosophical Transactions of the Royal Society of London—Series A, Mathematical and Physical Sciences* 363.1835 (2005), pp. 2351–2375.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. "CVC4". In: *Computer Aided Verification (CAV)*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Springer, 2011, pp. 171–177.

[BFT15] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.5*. Tech. rep. Available at `www.SMT-LIB.org`. Department of Computer Science, The University of Iowa, 2015.

[BSST09] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. Ed. by Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009. Chap. 26, pp. 825–885.

[BBW15] Peter Baumgartner, Joshua Bax, and Uwe Waldmann. "Beagle – A Hierarchic Superposition Theorem Prover". In: *Proc. Conference on Automated Deduction (CADE)*. Ed. by Amy Felty and Aart Middeldorp. Lecture Notes in Artificial Intelligence. To appear. Springer, 2015.

[BW13] Peter Baumgartner and Uwe Waldmann. "Hierarchic Superposition with Weak Abstraction". English. In: *Proc. Conference on Automated Deduction (CADE)*. Ed. by MariaPaola Bonacina. Vol. 7898. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 39–57.

[Bec98] Bernhard Beckert. "Ridig E-Unification". In: *Automated Deduction: A Basis for Applications. Foundations: Calculi and Methods*. Ed. by Wolfgang Bibel and P. H. Schimidt. Vol. 1. Bluwer Academic Publishers, 1998.

[BN00] Stefan Berghofer and Tobias Nipkow. "Proof Terms for Simply Typed Higher Order Logic". In: *Theorem Proving in Higher Order Logics (TPHOLs)*. Ed. by Mark Aagaard and John Harrison. Vol. 1869. Lecture Notes in Computer Science. Springer, 2000, pp. 38–52.

[BFT11] Frédéric Besson, Pascal Fontaine, and Laurent Théry. "A Flexible Proof Format for SMT: a Proposal". In: *Workshop on Proof eXchange for Theorem Proving (PxTP)*. 2011.

[BBF+16] Jasmin Christian Blanchette, Sascha Böhme, Mathias Fleury, Steffen Juilf Smolka, and Albert Steckermeier. "Semi-intelligible Isar Proofs from Machine-Generated Proofs". In: *Journal of Automated Reasoning* 56.2 (2016), pp. 155–200.

[BFSW11] Sascha Böhme, Anthony C. J. Fox, Thomas Sewell, and Tjark Weber. "Reconstruction of Z3's Bit-Vector Proofs in HOL4 and Isabelle/HOL". In: *Certified Programs and Proofs*. Ed. by Jean-Pierre Jouannaud and Zhong Shao. Vol. 7086. Lecture Notes in Computer Science. Springer, 2011, pp. 183–198.

[BW10] Sascha Böhme and Tjark Weber. "Fast LCF-Style Proof Reconstruction for Z3". In: *ITP 2010*. Ed. by Matt Kaufmann and Lawrence Paulson. Vol. 6172. Lecture Notes in Computer Science. Springer, 2010, pp. 179–194.

[BdODF09] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. "veriT: An Open, Trustable and Efficient SMT-Solver". In: *Proc. Conference on Automated Deduction (CADE)*. Ed. by Renate A. Schmidt. Vol. 5663. Lecture Notes in Computer Science. Springer, 2009, pp. 151–156.

[Bur13] Guillaume Burel. "A Shallow Embedding of Resolution and Superposition Proofs into the λΠ-Calculus Modulo". In: *Workshop on Proof eXchange for Theorem Proving (PxTP)*. Ed. by Jasmin Christian Blanchette and Josef Urban. Vol. 14. EPiC Series in Computing. EasyChair, 2013, pp. 43–57.

[CD07] Denis Cousineau and Gilles Dowek. "Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo". In: *Typed Lambda Calculi and Applications (TLCA)*. Ed. by Simona Ronchi Della Rocca. Vol. 4583. Lecture Notes in Computer Science. Springer, 2007, pp. 102–117.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. "A Machine Program for Theorem-proving". In: *Commun. ACM* 5.7 (July 1962), pp. 394–397.

[dMB08a] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Proofs and Refutations, and Z3". In: *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) Workshops*. Ed. by Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, Renate A. Schmidt, and Stephan Schulz. Vol. 418. CEUR Workshop Proceedings. CEUR-WS.org, 2008.

[dMB08b] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340.

[dMJ13] Leonardo Mendonça de Moura and Dejan Jovanovic. "A Model-Constructing Satisfiability Calculus". In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 2013, pp. 1–12.

[dMB07] Leonardo de Moura and Nikolaj Bjørner. "Efficient E-Matching for SMT Solvers". In: *Proc. Conference on Automated Deduction (CADE)*. Ed. by Frank Pfenning. Vol. 4603. Lecture Notes in Computer Science. Springer, 2007, pp. 183–198.

[dMB08c] Leonardo de Moura and Nikolaj Bjørner. "Engineering DPLL(T) + Saturation". In: *International Joint Conference on Automated Reasoning (IJCAR)*. Ed. by Alessandro Armando, Peter Baumgartner, and Gilles Dowek. Vol. 5195. Lecture Notes in Computer Science. Springer, 2008, pp. 475–490.

[dNiv02] Hans de Nivelle. "Extraction of Proofs from the Clausal Normal Form Transformation". In: *Computer Science Logic (CSL)*. Ed. by Julian Bradfield. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 584–598.

[dNiv05] Hans de Nivelle. "Translation of resolution proofs into short first-order proofs without choice axioms". In: *Information and Computation* 199.1 (2005), pp. 24–54.

[DV96] Anatoli Degtyarev and Andrei Voronkov. "The Undecidability of Simultaneous Rigid E-Unification". In: *Theor. Comput. Sci.* 166.1&2 (1996), pp. 291–300.

[DV98] Anatoli Degtyarev and Andrei Voronkov. "What You Always Wanted to Know about Rigid E-Unification". In: *Journal of Automated Reasoning* 20.1 (1998), pp. 47–80.

[DV01] Anatoli Degtyarev and Andrei Voronkov. "Equality Reasoning in Sequent-Based Calculi". In: *Handbook of Automated Reasoning*. Ed. by John Alan Robinson and Andrei Voronkov. Elsevier, 2001, pp. 611–706.

[DFLM13] David Déharbe, Pascal Fontaine, Daniel Le Berre, and Bertrand Mazure. "Computing Prime Implicants". In: *Formal Methods In Computer-Aided Design (FMCAD)*. IEEE, 2013, pp. 46–52.

[DFMP11] David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. "Exploiting Symmetry in SMT Problems". In: *Proceedings of the 23rd International Conference on Automated Deduction*. Proc. Conference on Automated Deduction (CADE). Wrocław, Poland: Springer-Verlag, 2011, pp. 222–236.

[DFP11] David Déharbe, Pascal Fontaine, and Bruno Woltzenlogel Paleo. "Quantifier Inference Rules for SMT proofs". In: *Workshop on Proof eXchange for Theorem Proving (PxTP)*. 2011.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. "Simplify: A Theorem Prover for Program Checking". In: *J. ACM* 52.3 (2005), pp. 365–473.

[DCKP13] Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich. "Adding Decision Procedures to SMT Solvers using Axioms with Triggers". 2013.

[Dut14] Bruno Dutertre. "Yices 2.2". English. In: *Computer Aided Verification (CAV)*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 737–744.

[DdM06] Bruno Dutertre and Leonardo de Moura. "A Fast Linear-Arithmetic Solver for DPLL(T)". English. In: *Computer Aided Verification (CAV)*. Ed. by Thomas Ball and Robert B. Jones. Vol. 4144. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 81–94.

[EHR+16] Gabriel Ebner, Stefan Hetzl, Giselle Reis, Martin Riener, Simon Wolfsteiner, and Sebastian Zivota. "System Description: GAPT 2.0". In: *International Joint Conference on Automated Reasoning (IJCAR)*. Ed. by Nicola Olivetti and Ashish Tiwari. Vol. 9706. Lecture Notes in Computer Science. Springer, 2016, pp. 293–301.

[EKK+11] Andreas Eggers, Evgeny Kruglov, Stefan Kupferschmid, Karsten Scheibler, Tino Teige, and Christoph Weidenbach. "Superposition Modulo Non-linear Arithmetic". In: *Frontiers of Combining Systems (FroCoS)*. Ed. by Cesare Tinelli and Viorica Sofronie-Stokkermans. Vol. 6989. Lecture Notes in Computer Science. Springer, 2011, pp. 119–134.

[EKK+16] Burak Ekici, Guy Katz, Chantal Keller, Alain Mebsout, Andrew Reynolds, and Cesare Tinelli. "Extending SMTCoq, a Certified Checker for SMT (Extended Abstract)". In: *Hammers for Type Theories*. Ed. by Jasmin Christian Blanchette and Cezary Kaliszyk. Vol. 210. EPTCS. 2016, pp. 21–29.

[Fit96] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, 1996.

[FMM+06] Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Fernanto Tiu. "Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants". In: *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. Ed. by Holger Hermanns and Jens Palsberg. Vol. 3920. Lecture Notes in Computer Science. Springer, 2006, pp. 167–181.

[GBT07] Yeting Ge, Clark Barrett, and Cesare Tinelli. "Solving Quantified Verification Conditions Using Satisfiability Modulo Theories". English. In: *Proc. Conference on Automated Deduction (CADE)*. Ed. by Frank Pfenning. Vol. 4603. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 167–182.

[GdM09] Yeting Ge and Leonardo de Moura. "Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories". In: *Computer Aided Verification (CAV)*. Ed. by Ahmed Bouajjani and Oded Maler. Vol. 5643. Lecture Notes in Computer Science. Springer, 2009, pp. 306–320.

[Gie01] Martin Giese. "Incremental Closure of Free Variable Tableaux". English. In: *International Joint Conference on Automated Reasoning (IJCAR)*. Ed. by Rajeev Goré, Alexander Leitsch, and Tobias Nipkow. Vol. 2083. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 545–560.

[Gie02] Martin Giese. "A Model Generation Style Completeness Proof for Constraint Tableaux with Superposition". English. In: *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*. Ed. by Uwe Egly and ChritianG. Fermüller. Vol. 2381. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 130–144.

[GM93] M. J. C. Gordon and T. F. Melham, eds. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[GMW79] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*. Vol. 78. LNCS. Springer, 1979.

[Gou93] Jean Goubault. "A rule-based algorithm for rigid E-unification". In: *Computational Logic and Proof Theory: Third Kurt Gödel Colloquium (KGC)*. Ed. by Georg Gottlob, Alexander Leitsch, and Daniele Mundici. Springer, 1993, pp. 202–210.

[Gra13] Stéphane Graham-Lengrand. "Psyche: A Proof-Search Engine Based on Sequent Calculus with an LCF-Style Architecture". In: *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*. Ed. by Didier Galmiche and Dominique Larchey-Wendling. Vol. 8123. Lecture Notes in Computer Science. Springer, 2013, pp. 149–156.

[HBR+15] Liana Hadarean, Clark W. Barrett, Andrew Reynolds, Cesare Tinelli, and Morgan Deters. "Fine Grained SMT Proofs for the Theory of Fixed-Width Bit-Vectors". In: *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. Ed. by Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov. Vol. 9450. Lecture Notes in Computer Science. Springer, 2015, pp. 340–355.

[Häh01] Reiner Hähnle. "Tableaux and Related Methods". In: *Handbook of Automated Reasoning*. Ed. by Alan Robinson and Andrei Voronkov. Amsterdam, The Netherlands, The Netherlands: Elsevier Science Publishers B. V., 2001, pp. 1853–1964.

[HHP87] Robert Harper, Furio Honsell, and Gordon D. Plotkin. "A Framework for Defining Logics". In: IEEE Computer Society, 1987, pp. 194–204.

[HLRR13] Stefan Hetzl, Tomer Libal, Martin Riener, and Mikheil Rukhaia. "Understanding Resolution Proofs through Herbrand's Theorem". In: *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*. Ed. by Didier Galmiche and Dominique Larchey-Wendling. Vol. 8123. Lecture Notes in Computer Science. Springer, 2013, pp. 157–171.

[JdM12] Dejan Jovanović and Leonardo de Moura. "Solving Non-linear Arithmetic". English. In: *International Joint Conference on Automated Reasoning (IJCAR)*. Ed. by Bernhard Gramlich, Dale Miller, and Uli Sattler. Vol. 7364. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 339–354.

[KBT+16] Guy Katz, Clark W. Barrett, Cesare Tinelli, Andrew Reynolds, and Liana Hadarean. "Lazy proofs for DPLL(T)-based SMT solvers". In: *Formal Methods In Computer-Aided Design (FMCAD)*. Ed. by Ruzica Piskac and Muralidhar Talupur. IEEE, 2016, pp. 93–100.

[KV13] Laura Kovács and Andrei Voronkov. "First-Order Theorem Proving and Vampire". English. In: *Computer Aided Verification (CAV)*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 1–35.

[LMO05] K. Rustan M. Leino, Madan Musuvathi, and Xinming Ou. "A Two-Tier Technique for Supporting Quantifiers in a Lazily Proof-Explicating Theorem Prover". In: *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. Ed. by Nicolas Halbwachs and Lenore D. Zuck. Vol. 3440. Lecture Notes in Computer Science. Springer, 2005, pp. 334–348.

[LP16] K. Rustan M. Leino and Clément Pit-Claudel. "Trigger Selection Strategies to Stabilize Program Verifiers". In: *Computer Aided Verification (CAV)*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9779. Lecture Notes in Computer Science. Springer, 2016, pp. 361–381.

[MBG06] Sean McLaughlin, Clark Barrett, and Yeting Ge. "Cooperating Theorem Provers: A Case Study Combining HOL-Light and CVC Lite". In: *Electr. Notes Theor. Comput. Sci.* 144.2 (2006), pp. 43–51.

[Mei00] Andreas Meier. "TRAMP: Transformation of machine-found proofs into natural deduction proofs at the assertion level (system description)". In: *Proc. Conference on Automated Deduction (CADE)*. Ed. by David McAllester. Vol. 1831. Lecture Notes in Computer Science. Springer, 2000, pp. 460–464.

[Mil15] Dale Miller. "Proof Checking and Logic Programming". In: *Logic-Based Program Synthesis and Transformation (LOPSTR)*. Ed. by Moreno Falaschi. Vol. 9527. Lecture Notes in Computer Science. Springer, 2015, pp. 3–17.

[Mos08] Michał Moskal. "Rocket-Fast Proof Checking for SMT Solvers". In: *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 486–500.

[MŁ06] Michał Moskal and Jakub Łopuszański. *Fast Quantifier Reasoning With Lazy Proof Explication*. Tech. rep. Institute of Computer Science, University of Wrocław, 2006.

[MŁK08] Michał Moskal, Jakub Łopuszański, and Joseph R. Kiniry. "E-matching for Fun and Profit". In: *Electron. Notes Theor. Comput. Sci.* 198.2 (May 2008), pp. 19–35.

[Nel80] Charles Gregory Nelson. "Techniques for Program Verification". PhD thesis. Stanford, CA, USA, 1980.

[NO79] Greg Nelson and Derek C. Oppen. "Simplification by Cooperating Decision Procedures". In: *ACM Trans. Program. Lang. Syst.* 1.2 (Oct. 1979), pp. 245–257.

[NO80] Greg Nelson and Derek C. Oppen. "Fast Decision Procedures Based on Congruence Closure". In: *J. ACM* 27.2 (1980), pp. 356–364.

[NO07] Robert Nieuwenhuis and Albert Oliveras. "Fast congruence closure and extensions". In: *Information and Computation* 205.4 (2007). Special Issue: 16th International Conference on Rewriting Techniques and Applications, pp. 557–580.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. "Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T)". In: *J. ACM* 53.6 (Nov. 2006), pp. 937–977.

[NR01] Robert Nieuwenhuis and Albert Rubio. "Paramodulation-Based Theorem Proving". In: *Handbook of automated reasoning*. Ed. by Alan Robinson and Andrei Voronkov. Vol. 1. 2001, pp. 371–443.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Vol. 2283. LNCS. Springer, 2002.

[NWCS01] Andreas Nonnengart, Weidenbach, Christoph, and Stadtwald. "Computing Small Clause Normal Forms". In: *Handbook of automated reasoning*. Ed. by Alan Robinson and Andrei Voronkov. Vol. 1. Elsevier and MIT Press, 2001, pp. 335–367.

[Pau83] Lawrence C. Paulson. "A Higher-Order Implementation of Rewriting". In: *Sci. Comput. Program.* 3.2 (1983), pp. 119–149.

[PS07] Lawrence C. Paulson and Kong Woei Susanto. "Source-Level Proof Reconstruction for Interactive Theorem Proving". In: *Theorem Proving in Higher Order Logics (TPHOLs)*. Ed. by Klaus Schneider and Jens Brandt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 232–245.

[PWZ14] Ruzica Piskac, Thomas Wies, and Damien Zufferey. "GRASShopper - Complete Heap Verification with Mixed Specifications". In: *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. Ed. by Erika Ábrahám and Klaus Havelund. Vol. 8413. Lecture Notes in Computer Science. Springer, 2014, pp. 124–139.

[Pug92] William Pugh. "The Omega Test: a fast and practical integer programming algorithm for dependence analysis". In: *Communications of the ACM* 8 (1992), pp. 4–13.

[RBSV16] Giles Reger, Nikolaj Bjorner, Martin Suda, and Andrei Voronkov. "AVATAR Modulo Theories". In: *Global Conference on Artificial Intelligence (GCAI)*. Ed. by Christoph Benzmüller, Geoff Sutcliffe, and Raul Rojas. Vol. 41. EPiC Series in Computing. EasyChair, 2016, pp. 39–52.

[Rey16] Andrew Reynolds. "Conflicts, Models and Heuristics for Quantifier Instantiation in SMT". In: *Vampire workshop*. Ed. by Laura Kovács and Andrei Voronkov. EPiC Series in Computing. EasyChair, 2016, pp. 1–15.

[RTdM14] Andrew Reynolds, Cesare Tinelli, and Leonardo Mendonça de Moura. "Finding conflicting instances of quantified formulas in SMT". In: *Formal Methods In Computer-Aided Design (FMCAD)*. IEEE, 2014, pp. 195–202.

[RTGK13] Andrew Reynolds, Cesare Tinelli, Amit Goel, and Sava Krstić. "Finite Model Finding in SMT". English. In: *Computer Aided Verification (CAV)*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 640–655.

[RTG+13] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters, and Clark Barrett. "Quantifier Instantiation Techniques for Finite Model Finding in SMT". In: *Proc. Conference on Automated Deduction (CADE)*. Ed. by Maria Paola Bonacina. Vol. 7898. Lecture Notes in Computer Science. Springer, 2013, pp. 377–391.

[Rüm08] Philipp Rümmer. "A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic". In: *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. Ed. by Iliano Cervesato, Helmut Veith, and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 274–289.

[Rüm12] Philipp Rümmer. "E-Matching with Free Variables". In: *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. Ed. by Nikolaj Bjørner and Andrei Voronkov. Vol. 7180. Lecture Notes in Computer Science. Springer, 2012, pp. 359–374.

[Sch13] Stephan Schulz. "System Description: E 1.8". English. In: *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. Ed. by Ken McMillan, Aart Middeldorp, and Andrei Voronkov. Vol. 8312. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 735–743.

[Seb07] Roberto Sebastiani. "Lazy Satisability Modulo Theories". In: *JSAT* 3.3-4 (2007), pp. 141–224.

[Sho84] Robert E. Shostak. "Deciding Combinations of Theories". In: *J. ACM* 31.1 (Jan. 1984), pp. 1–12.

[SP16] John Slaney and Bruno Woltzenlogel Paleo. "Conflict Resolution: a First-Order Resolution Calculus with Decision Literals and Conflict-Driven Clause Learning". In: *CoRR* abs/1602.04568 (2016).

[Stu09] Aaron Stump. "Proof Checking Technology for Satisfiability Modulo Theories". In: *Electr. Notes Theor. Comput. Sci.* 228 (2009), pp. 121–133.

[SZS04] Geoff Sutcliffe, Jürgen Zimmer, and Stephan Schulz. "TSTP Data-Exchange Formats for Automated Theorem Proving Tools". In: *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*. Ed. by Weixiong Zhang and Volker Sorge. Vol. 112. Frontiers in Artificial Intelligence and Applications. IOS Press, 2004, pp. 201–215.

[TBR00] Ashish Tiwari, Leo Bachmair, and Harald Ruess. "Rigid E-Unification Revisited". In: *Proc. Conference on Automated Deduction (CADE)*. Ed. by David McAllester. Vol. 1831. Lecture Notes in Computer Science. Springer, 2000, pp. 220–234.

[Tse83] G. S. Tseitin. "On the Complexity of Derivation in Propositional Calculus". In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Ed. by Jörg H. Siekmann and Graham Wrightson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 466–483.

[Vor14] Andrei Voronkov. "AVATAR: The Architecture for First-Order Theorem Provers". English. In: *Computer Aided Verification (CAV)*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 696–710.

[WDF+09] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. "SPASS Version 3.5". English. In: *Proc. Conference on Automated Deduction (CADE)*. Ed. by RenateA. Schmidt. Vol. 5663. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 140–145.

[ZWR16] Aleksandar Zeljic, Christoph M. Wintersteiger, and Philipp Rümmer. "Deciding Bit-Vector Formulas with mcSAT". In: *Theory and Applications of Satisfiability Testing (SAT)*. Ed. by Nadia Creignou and Daniel Le Berre. Vol. 9710. Lecture Notes in Computer Science. Springer, 2016, pp. 249–266.

[ZMSZ04] Jürgen Zimmer, Andreas Meier, Geoff Sutcliffe, and Yuan Zhan. "Integrated Proof Transformation Services". In: *Workshop on Computer-Supported Mathematical Theory Development*. Ed. by Christoph Benzmüller and Wolfgang Windsteiger. 2004.