UNIVERSITÉ DE LORRAINE | BIBLIOTHÈQUES UNIVERSITAIRES

AVERTISSEMENT
DROIT D'AUTEUR

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document. Toute contrefaçon, plagiat ou reproduction illicite encourt une poursuite pénale.

Contact bibliothèque : ddoc-theses-contact@univ-lorraine.fr
(Cette adresse ne permet pas de contacter les auteurs)

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10

UNIVERSITÉ DE LORRAINE
34 COURS LÉOPOLD — CS 25233
54052 NANCY CEDEX
TÉL : 03 54 50 54 00
CONTACT@UNIV-LORRAINE.FR
WWW.UNIV-LORRAINE.FR

Reconstructing SMT Proofs in Lambdapi

COLTELLACCI ALESSIO



UNIVERSITÉ DE LORRAINE

IAEM Lorraine Doctoral School

Reconstructing SMT Proofs in Lambdapi

THESIS

Defended on 26 January 2026

for the degree of

Doctor of the University of Lorraine
specialty: Computer Science

by

ALESSIO COLTELLACCI

Thesis Committee:

Chair:	Régine Laleau	University Paris-Est Créteil
Reviewers:	Laura Kovács	Vienna University of Technology
	Alexander Steen	University of Greifswald
Examiners:	Mathias Fleury	University of Freiburg
	Chantal Keller	LMF, University Paris-Saclay
Supervisors:	Stephan Merz	INRIA, University of Lorraine
	Gilles Dowek	INRIA, University of Paris-Saclay

LORIA — Lorraine Research Laboratory in Computer Science and its Applications — UMR 7503

Coltellacci Alessio: *Reconstructing SMT Proofs in Lambdapi,* Checking and Exporting SMT Proofs using Lambdapi, © 2026

SUPERVISORS:
Pr. Stephan Merz
Pr. Gilles Dowek

LOCATION:
France

TIME FRAME:
2026

What men really want is not knowledge but certainty.

— Bertrand Russell

Madness is the result not of uncertainty but certainty.

— Friedrich Nietzsche

亲爱的小猫，

此间所书，见证着我们神话留下的痕迹。这则神话始于东方，始于那座海上雄狮之岛——那狮子屹立于自己的海岬之上，置身于熠熠生辉的青色环礁之间。在那里，岁月恒久不变，日升日落从天地之初起便循着同样的时辰。那是东方之海珍宝交汇的十字路口。自从我们在那里相遇，我们便一同探索着这片宇宙，将我们的足迹一路延伸至“六边形之国”，延伸至英格兰那葱郁青翠的山丘——在那里，亚瑟王传奇是一种精神寄托。在这片宇宙之中，专属于我们二人的“通用语”得以诞生，而我们每一天都在共同地丰盈着它。

因此，这篇论文理应归功于你。在我们这段共同渡过的时光里，是你赋予了这个项目色彩和独特灵魂。你曾是它的见证者，也是它最坚强的后盾。当我的外婆离我而去的时候——我曾深爱她至无以复加——你以你的魔法，将那一天化作一个恬静的时刻，仿佛静静悬于时光之中，定格于英格兰的悬崖之上。你赠予我这世间所罕有的：让我专注的宁静，让我前行的信念。你懂得在我周围编织出那样一片静默与安详的空间，使我最好的一面得以从中浮现。正因如此，我要再次重复：这篇论文，应属于你。

小狗

PUBLICATIONS

[1] Alessio Coltellacci. "Reconstruction of TLAPS Proofs Solved by VeriT in Lambdapi." In: *Rigorous State-Based Methods - 9th International Conference, ABZ 2023, Nancy, France, 2023, Proceedings*. Ed. by Uwe Glässer, José Creissac Campos, Dominique Méry, and Philippe A. Palanque. Vol. 14010. Lecture Notes in Computer Science. Springer, 2023, pp. 375–377.

[2] Alessio Coltellacci, Bruno Andreotti, Haniel Barbosa, Gilles Dowek, and Stephan Merz. "Reconstruction of SMT proofs with Lambdapi." In: *Acta Informatica* 63.1 (2026), p. 8. DOI: 10.1007/S00236-025-00515-W. URL: https://doi.org/10.1007/s00236-025-00515-w.

[3] Alessio Coltellacci and Stephan Merz. "Checking Linear Integer Arithmetic Proofs in Lambdapi." In: *Frontiers of Combining Systems*. Ed. by René Thiemann and Christoph Weidenbach. Cham: Springer Nature Switzerland, 2026, pp. 367–385.

[4] Alessio Coltellacci, Stephan Merz, and Gilles Dowek. "Reconstruction of SMT Proofs with Lambdapi." In: *Proceedings of the 22nd International Workshop on Satisfiability Modulo Theories co-located with the 36th International Conference on Computer Aided Verification (CAV 2024), 2024*. Ed. by Giles Reger and Yoni Zohar. Vol. 3725. CEUR Workshop Proceedings. CEUR-WS.org, 2024, pp. 13–23.

ABSTRACT

Modern automated theorem provers are increasingly used to establish the correctness of critical computer systems. Among them, Satisfiability Modulo Theories (SMT) solvers play a central role thanks to their expressive power and efficiency in handling complex logical reasoning tasks. However, their size and complexity make full formal verification of their implementations impractical, raising a fundamental question of trust:

Can we trust the correctness of their results?

A promising answer lies in *proof logging*, where solvers produce proof traces that proof checkers can independently check. The Alethe format has recently emerged as a unified framework for representing such SMT proofs. Several SMT solvers have adopted the Alethe format. However, Alethe lacks a certified proof checker, limiting its adoption and trustworthiness.

In this thesis, we propose a reconstruction framework that translates Alethe proof traces into the Lambdapi proof assistant, an implementation of the $\lambda\Pi/\equiv$-calculus modulo rewriting. Lambdapi is a foundational proof assistant based on dependent type theory and rewriting rules, intended to serve as a pivot for exchanging proofs between interactive proof assistants. We present a modular encoding of SMT logic and Alethe rules in Lambdapi, covering core theories such as first-order reasoning and linear arithmetic.

We detail the reconstruction process, which requires techniques such as proof by reflection and the elaboration of solver-specific simplifications. Finally, we evaluate our implementation on benchmark examples and demonstrate that Alethe proofs generated by modern SMT solvers can be efficiently and reliably reconstructed in Lambdapi, paving the way toward portable and independently verifiable SMT proofs.

RÉSUMÉ

Les démonstrateurs automatiques de théorème modernes sont de plus en plus plébiscités pour établir la correction de systèmes informatiques critiques. Parmi eux, les solveurs de satisfiabilité modulo des théories (SMT) occupent une place centrale, en raison de leur grande expressivité et de leur efficacité dans le traitement de raisonnements logiques complexes. Cependant, l'ampleur de leur base de code, conjuguée à la sophistication de leur architecture logicielle, rend toute vérification formelle exhaustive impraticable, soulevant ainsi une question fondamentale de confiance:

Dans quelle mesure peut-on accorder foi à la correction des résultats produits par ces solveurs ?

Une réponse prometteuse à cette problématique réside dans le recours à la journalisation de preuves (*proof logging*), approche consistant à faire produire par les solveurs des traces de preuve susceptibles d'être vérifiées de manière indépendante par des vérificateurs externes. Le format Alethe est récemment apparu comme un standard pour représenter de telles preuves SMT. Plusieurs solveurs SMT ont adopté le format Alethe. Néanmoins, l'absence actuelle d'un vérificateur de preuves certifié pour Alethe en limite encore l'adoption.

Dans cette thèse, nous proposons un outil permettant de traduire les preuves SMT produites au format Alethe vers l'assistant de preuve Lambdapi, une implémentation du λΠ-calcul modulo théorie. Lambdapi est un assistant de preuve fondationnel, basé sur la théorie des types dépendants et sur des règles de réécriture, conçu pour servir de pivot pour l'échange de preuves entre assistants de preuve interactifs. Nous présentons un encodage modulaire de la logique SMT et des règles Alethe dans Lambdapi, couvrant des théories fondamentales telles que le raisonnement du premier ordre et l'arithmétique linéaire.

Nous détaillons ensuite le processus de reconstruction, qui mobilise des techniques comme la preuve par réflexion et l'élaboration de simplifications propres aux solveurs. Enfin, nous évaluons notre implémentation sur des benchmarks et montrons que les preuves Alethe générées par des solveurs SMT modernes peuvent être reconstruites de manière efficace et fiable dans Lambdapi, ouvrant ainsi la voie à des preuves SMT à la fois vérifiées et portables.

RÉSUMÉ ÉTENDU

CONTEXTE ET MOTIVATION

Les solveurs SMT (*Satisfiabilité modulo théories*) [12, 28] se sont imposés comme des outils centraux de la vérification formelle moderne. Ces solveurs sont utilisés comme outils pour la démonstration automatique de théorèmes intégrant le raisonnement propositionnel, enrichi par des théories telles que l'arithmétique linéaire, les fonctions non interprétées ou encore les bit-vectors. Leur grande efficacité en fait des composants essentiels de nombreux logiciels industriels et académiques, par exemple pour la modélisation et la vérification de systèmes critiques, la vérification de programmes et de matériels, l'analyse statique, ou encore l'intégration dans des assistants de preuve [4, 71].

Cependant, cette efficacité s'accompagne d'une question fondamentale de confiance dans les résultats produits. Les solveurs SMT sont des logiciels complexes, optimisés et en constante évolution. Ils sont en outre souvent écrits en C/C++ pour des raisons de performance, ce qui complique fortement leur vérification formelle et leur formalisation. Il est donc difficile d'en certifier entièrement l'implémentation, et l'expérience montre que des bogues peuvent entraîner des verdicts incorrects [29]. Dans des contextes critiques, où leur résultat entraîne une décision de sûreté, il devient indispensable de renforcer notre confiance en ces outils.

L'approche prometteuse du *proof logging* [13] consiste à certifier les résultats du solveur plutôt que son implémentation. Cette approche consiste en ce que le solveur produise une trace de preuve, appelée *certificat*, qui justifie le verdict. Un outil indépendant, plus simple que le solveur, vérifie ensuite ce certificat. Cette séparation des rôles est attrayante car la vérification d'une preuve est conceptuellement plus simple et peut s'appuyer sur un noyau de confiance réduit.

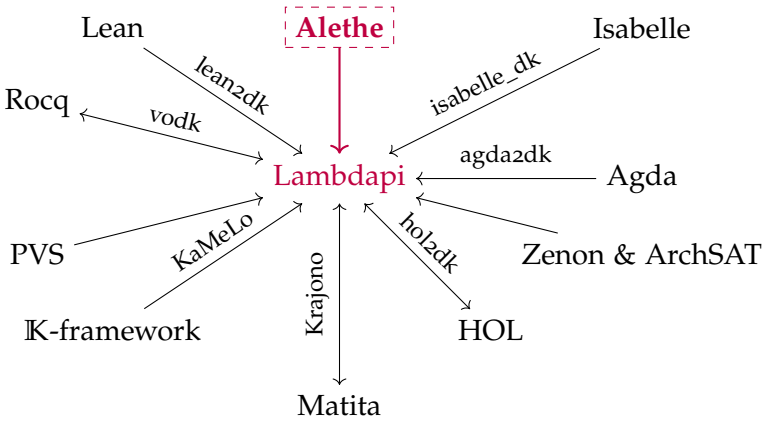


Image 1: Lambdapi, un langage assembleur pour les systèmes de preuve.

Le format de preuve Alethe [90] a émergé comme un format de preuve standard pour représenter des preuves d'insatisfiabilité SMT. Ces preuves justifient le verdict `unsat` d'un solveur en montrant que le problème d'entrée ne possède aucun modèle. Alethe repose sur SMT-LIB, le langage standard pour exprimer des problèmes SMT [18]. Les preuves sont organisées en une suite d'étapes, et Alethe propose un catalogue de règles de raisonnement couvrant la résolution, les lemmes de théorie, les simplifications, les quantificateurs et la skolémisation. Néanmoins, Alethe ne dispose pas encore d'un vérificateur certifié largement adopté, ce qui limite son rôle en tant que format d'échange réellement fiable [3]. Des travaux de reconstruction existent néanmoins dans des assistants de preuve, par exemple dans Isabelle/HOL [60, 88].

Cette thèse répond à ce besoin en développant un cadre de *reconstruction* de preuves Alethe dans Lambdapi [37]. Lambdapi est un assistant de preuve fondé sur le λΠ-calcul modulo théories, conçu comme un langage pivot pour échanger des preuves entre systèmes. Comme l'illustre la Figure 1, de nombreux assistants et outils peuvent exporter ou importer des preuves via Lambdapi, qui joue alors le rôle de "langage assembleur" pour les systèmes de preuve. Reconstruire une preuve Alethe dans Lambdapi revient à produire un terme de preuve dont la validité est vérifiée par le noyau de Lambdapi, ce qui fournit une garantie forte. L'objectif est ainsi de rendre les preuves Alethe portables et vérifiables indépendamment du solveur.

ALETHE, PREUVES SMT ET ENJEUX DE VÉRIFICATION

Le format de traces de preuve Alethe [11] pour les solveurs SMT se compose de deux parties : un langage de traces fondé sur SMT-LIB, et une collection de règles de preuve. Les traces constituent des témoins d'insatisfiabilité d'un ensemble de contraintes. Elles prennent la forme de séquences $a_1 \dots a_m\ t_1 \dots t_n$ où les a_i correspondent aux contraintes du problème SMT initial réfuté, chaque t_i est une clause inférée à partir des éléments précédents de la séquence, et t_n est $\bot$ i.e. la clause vide. Dans la suite, nous désignons le problème SMT-LIB comme le *problème d'entrée*.

```
(set-logic UF)
(declare-sort U 0)
(declare-fun a () U)
(declare-fun b () U)
(declare-fun p (U) Bool)
(assert (p a))
(assert (= a b))
(assert (not (p b)))
(check-sat)
(get-proof)
```

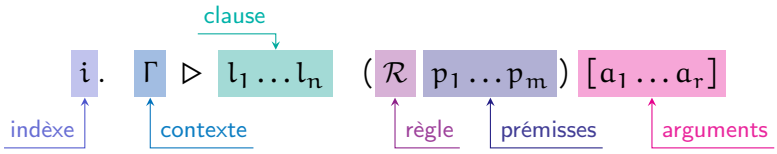
↯

```
(assume a0 (p a))
(assume a1 (= a b))
(assume a2 (not (p b)))
(step t1 (cl (not (= (p a) (p b))) (not (p a)) (p b)) :rule equiv_pos2)
```

```
5 (step t2 (cl (= (p a) (p b))) :rule cong :premises (a1))
6 (step t3 (cl (p b)) :rule resolution :premises (t1 t2 a0))
7 (step t4 (cl) :rule resolution :premises (a2 t3))
```

Code 1: Un problème SMT et sa preuve Alethe trouvée par cvc5.

L'exemple de la Code 1 illustre ce principe : à partir du problème d'entrée, le solveur produit une trace Alethe composée d'étapes `assume` et `step`, aboutissant à la clause vide `(cl)`. Dans ce problème d'entrée, on suppose d'abord que $p(a)$ est vrai et que $a = b$. Intuitivement, l'égalité $a = b$ permet de transporter la propriété p de a vers b, donc d'obtenir $p(b)$. Or on ajoute aussi l'hypothèse contraire $\neg p(b)$: on cherche donc à montrer que cet ensemble d'hypothèses est contradictoire. Les solveurs SMT procèdent par réfutation : ils établissent cette contradiction, c'est-à-dire l'*insatisfiabilité*. L'insatisfiabilité obtenue certifie alors que, dans le problème initial, $p(a)$ et $a = b$ entraînent bien $p(b)$; la trace Alethe en donne la justification étape par étape. Pour analyser et reconstruire de telles traces, nous décrivons maintenant (Équation (1)) la structure générale d'une étape Alethe:



(1)

Pour analyser et reconstruire de telles traces, nous décrivons maintenant la structure générale d'une étape Alethe, résumée par Équation (1). Une étape est constituée d'un indice $i \in \mathbb{I}$, où $\mathbb{I}$ est un ensemble infini dénombrable d'indices (par exemple `a0`, `t1`), et d'une clause de formules $l_1, \dots, l_n$ représentant une disjonction n-aire. Les étapes qui ne sont pas des hypothèses sont justifiées par une règle de preuve $\mathcal{R}$ qui dépend d'un ensemble éventuellement vide de prémisses $\{p_1 \dots p_m\} \subseteq \mathbb{I}$, ne référant que des étapes antérieures, de sorte que la preuve forme un graphe orienté acyclique. Une règle peut aussi dépendre d'une liste d'arguments $[a_1 \dots a_r]$, où chaque argument a_i est soit un terme, soit une paire (x_i, t_i), où x_i est une variable et t_i un terme. L'interprétation des arguments dépend de la règle. Le contexte Γ d'une étape est une liste $c_1 \dots c_l$, où chaque élément c_j est soit une variable, soit un couple variable-terme noté $x_j \mapsto t_j$. Ainsi, les étapes dont le contexte est non vide contiennent des variables x_j qui apparaissent dans l_i et seront substituées par les t_j. Les règles de preuve $\mathcal{R}$ incluent des lemmes de théorie et d'autres règles de logique comme la règle `resolution`, qui correspond à l'hyper-résolution sur des clauses, et consiste à composer plusieurs résolutions binaires pour dériver une clause à partir de plusieurs prémisses.

Ces éléments suffisent pour appréhender la preuve du Code 1, qui comporte ici sept étapes. Elle commence par trois premières étapes `assume` (`a0`, `a1`, `a2`) qui réaffirment les assertions du problème d'entrée. Dans la syntaxe concrète, les étapes `assume` disposent d'une commande dédiée **`assume`**, afin de les distinguer clairement des étapes ordinaires, qui utilisent la commande **`step`**. Cette commande est du sucre syntaxique pour une commande `step` utilisant la règle Alethe

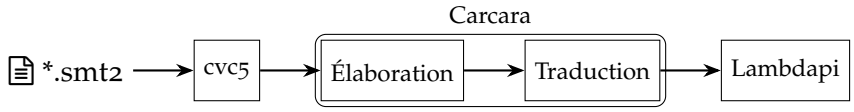


Image 2: Architecture du processus de reconstruction

`hole` qui admet une clause. L'étape `t1` introduit une tautologie de la forme $\neg(\varphi_1 \approx \varphi_2) \vee \neg\varphi_1 \vee \varphi_2$, justifiée par la règle `equiv_pos2`. L'étape `t2` établit $p(a) \approx p(b)$ par congruence à partir de l'hypothèse `a1` et matéralisée par la règle `cong`. L'étape `t3` applique ensuite la règle `resolution` aux prémisses `t1`, `t2` et `a0` pour dériver $p(b)$. Enfin, l'étape `t4` conclut la preuve en produisant la clause vide i.e. $\bot$, concrètement notée `(cl)`. Remarquons que les contextes Γ de toutes les étapes sont vides dans cet exemple.

Preuve à forte granularité

Les preuves Alethe sont exprimés sous un raisonnement à pas large entrainant des difficultés lorsqu'on cherche à les vérifiers. Par exemple, la règle `resolution` utilisé dans le Code 1 n'indique pas les pivots, ainsi que l'ordre des résolutions pour dériver la clause finale. L'ordre des résolutions binaires effectués durant l'hyper résolutions pourrer dériver differente clause sémantiquement equivalente mais syntaxiquement différente. De plus, le format ne fixe pas la sémantique des clauses et introduits aussi des transformations implicites au long de la preuve comme par exemple le réordonnement des littéraux dans une clause. Ainsi, le caractère simplicite de certaines règles Alethe, qui résument en une étape un calcul ou une simplification non triviale doivent être détaillé en raisonnement à pas plus petit afin d'être reconstruite dans un assistant à la preuve. D'autre part, les SMT solveurs tendent à travailler avec de large expression et peuvent produire de très longue preuve qui ainsi augmente en pratique la complexité spatiale pour vérifier une preuve.

ÉLABORATION DES TRACES : AFFINER POUR MIEUX VÉRIFIER

Afin de palier à certain problèmes décrit plus haut, nous introduisons une étape de preprocessing de la preuve avant de la reconstruire. Des outils comme Carcara [3] et RARE [60, 78] transforment les preuves Alethe produites par les solveurs en preuves à petit pas plus adaptées à la reconstruction. L'élaboration peut notamment: éliminer des pas redondants, expliciter des réordonnancements implicites, décomposer des règles trop grossières en sous-étapes plus élémentaires, ou remplacer des simplifications complexes par des séquences compatibles avec les bibliothèques de preuve disponibles. Néanmoins, une preuve élaborée augmente le taux de reconstruction réussie au détriment d'une complexité spatiale et temporel qui augmente i.e. la preuve aura plus d'étapes est sera donc plus difficile à valider.

Comme le présente la Image 2, nous tirons partie du module interne d'élaboration des preuves Alethe par Carcara afin ensuite de les traduires en Lambdapi pour les vérifiers. Nous illustrons ceci avec l'élaboration de la preuve Code 1. Carcara analyse la résolution est indique les pivots dans les arguments $[a_1 \dots a_r]$ de l'étape, de plus, il ordonne les premises $\{p_1 \dots p_m\}$.

Ainsi, comme on peut le constater dans le Code 2 ci-dessous, les étapes `t3` et `t4` sont maintenant détaillés avec les pivots et l'ordre des résolutions facilitatant la reconstruction dans un assistant à la preuve comme Lambdapi.

```
(step t3 (cl (p b)) :rule resolution :premises (t1 t2 a0)
    :args ((= (p a) (p b)) false (p a) false))
(step t4 (cl) :rule resolution :premises (a2 t3)
  :args ((p b) false)})
```

Code 2: Elaboration des étapes de résolutions de Code 1 par Carcara

Dans ces travaux nous avons ajouté un module de traduction dans Carcara afin de pouvoir traduire une preuve Alethe élaborée en une preuve Lambdapi que nous pouvons par la suite vérifier avec notre encodage de la logique SMT dans Lambdapi que nous allons présenter dans la section suivante.

LAMBDAPI COMME NOYAU DE CONFIANCE ET LANGAGE PIVOT

Lambdapi est une implémentation du λΠ-calcul modulo théories noté λΠ/≡ [37], une extension du *Edinburgh Logical Framework* λΠ [53], un λ-calcul simplement typé avec types dépendants. Le λΠ/≡ ajoute la possibilité de définir des règles de réécriture d'ordre supérieur. Sa syntaxe est donnée par la grammaire suivante:

Sortes	$s ::= \texttt{TYPE} \mid \texttt{KIND}$
Termes	$t, v, A, B, C ::= c \mid x \mid s \mid \Pi\, x : A,\, B \mid \lambda\, x : A,\, t \mid t\ v$
Contextes	$\Gamma ::= \langle\rangle \mid \Gamma, x : A$
Signatures	$\Sigma ::= \langle\rangle \mid \Sigma, c : C \mid \Sigma, c := t : C \mid \Sigma, t \hookrightarrow v$

où c est une constante, et x une variable. Les ensembles des variables et des constantes sont disjoints. Les *sortes* sont des constantes utilisées pour vérifier qu'un type est bien formé (voir [53, §2.1]). Le terme $\Pi\, x : A.\, B$ est le produit dépendant, et l'on écrit $A \to B$ lorsque x n'apparaît pas libre dans B ; $\lambda\, x : A.\, t$ est une abstraction et $t\ v$ une application. Un *contexte local* Γ est une suite finie de déclarations $x : A$ introduisant des variables et leurs types. Une *signature* Σ, représentant le contexte global, est une suite finie d'hypothèses $c : C$ (la constante c est de type C), de définitions $c := t : C$ et de règles de réécriture $t \hookrightarrow v$ telles que $t = c\ v_1 \dots v_n$ pour une constante c.

La relation $\hookrightarrow_{\beta\Sigma}$ est engendrée par la β-réduction et les règles de réécriture de Σ. On note $\hookrightarrow^*_{\beta\Sigma}$ sa clôture réflexive et transitive, et $\equiv_{\beta\Sigma}$ la clôture réflexive de $\hookrightarrow_{\beta\Sigma}$. La relation $\hookrightarrow_{\beta\Sigma}$ doit être confluente, c'est-à-dire que, si $t \hookrightarrow^*_{\beta\Sigma} v_1$ et $t \hookrightarrow^*_{\beta\Sigma} v_2$, alors il existe un terme w tel que

$$\frac{\Gamma \vdash_\Sigma B : u \qquad \Gamma \vdash_\Sigma t : A \qquad A \equiv_{\beta\Sigma} B}{\Gamma \vdash_\Sigma t : B}\ (\text{Conv})$$

Image 3: Règle de typage pour la conversion entre deux types.

$v_1 \hookrightarrow^*_{\beta\Sigma} w$ et $v_2 \hookrightarrow^*_{\beta\Sigma} w$; et elle doit préserver le typage : si $\Gamma \vdash_\Sigma t : A$ et $t \hookrightarrow_{\beta\Sigma} v$, alors $\Gamma \vdash_\Sigma v : A$ [21].

Un jugement de typage en Lambdapi est de la forme $\Gamma \vdash_\Sigma t : A$ et affirme que le terme t a le type A dans le contexte Γ et la signature Σ. Les règles de typage sont analogues à celles de $\lambda\Pi$ [53, §2], à ceci près que la règle Conv (Image 3) identifie les types modulo $\equiv_{\beta\Sigma}$ et non seulement modulo la β-réduction.

Encodage de la logique SMT en Lambdapi

Nous rassemblons ici un aperçu des types, des connecteurs logiques et des règles de réécriture qui constituent la signature Σ utilisée dans notre encodage de la logique SMT.

Définition (Aperçu de l'encodage)**.**

```
Set : TYPE                          Prop : TYPE
El : Set → TYPE                     Prf : Prop → TYPE
⇝ : Set → Set → Set                 ∨ : Prop → Prop → Prop
El (x ⇝ y) ↪ El x → El y            ¬ : Prop → Prop
Clause : TYPE                       𝓕 : Clause → Prop
■ : Clause                          𝓕 ■ ↪ ⊥
∀ : Prop → Clause → Clause          𝓕 x ∀ y ↪ x ∨ (𝓕 y)
++ : Clause → Clause → Clause       Prf•(c : Clause) ≔ Prf(𝓕 c)
■ ++ x ↪ x
(x ∀ y) ++ z ↪ x ∀ (y ++ z)
```

La première partie introduit deux univers *à la Tarski*, `Set` et `Prop`, ainsi que leurs fonctions de décodage *à la Gödel* `El` et `Prf`. Un terme x : `Set` (resp. p : `Prop`) est vu comme un `TYPE` via `El x` (resp. `Prf p`). Le constructeur $\rightsquigarrow$ joue le rôle d'un type de flèches, avec la règle de réécriture `El (x ⇝ y) ↪ El x → El y` réécrivant $\rightsquigarrow$ en une métaflèche. L'opérateur $\vee$ représente une disjonction binaire et utilise une notation infixe, tandis que $\neg$ représente la négation.

La seconde partie présente l'encodage des clauses. Le type `Clause` a deux constructeurs, le premier ■ représentant une clause vide, $\forall$ ajoute un littéral en tête à une clause existante, et l'interprétation $\mathcal{F}$ traduit une clause en une proposition. L'opérateur ++ modélise la concaténation de clauses via réécriture, et nous abrégeons enfin, pour toute clause c, une preuve de $\mathcal{F}\,c$ par `Prf•(c)` $\coloneqq$ `Prf` $(\mathcal{F}\,c)$.

Nous introduisons d'abord les connecteurs d'équivalence $(\Leftrightarrow)$ et d'égalité $(=)$. Le caractère classique du raisonnement des solveurs SMT est capturé par deux axiomes : le tiers exclu et l'extensionalité des propositions.

$$\Longleftrightarrow : \texttt{Prop} \to \texttt{Prop} \to \texttt{Prop} \qquad \text{(infixe)}$$
$$= : \Pi\,(a : \texttt{Set}) \to \texttt{El}\,a \to \texttt{El}\,a \to \texttt{Prop} \qquad \text{(infixe)}$$
$$\texttt{tiers_exclu} : \Pi[p : \texttt{Prop}], \texttt{Prf}(p \vee \neg p)$$
$$\texttt{prop_ext} : \Pi[p\,q : \texttt{Prop}], \texttt{Prf}(p \Longleftrightarrow q) \to \texttt{Prf}(p = q)$$

Fonctions de traduction de Alethe vers Lambdapi

Nous présentons ici les fonctions de traduction. Elles permettent de passer d'une trace Alethe, fondée sur la signature SMT-LIB, à des objets bien typés en Lambdapi.

La signature SMT-LIB est décrite dans [18, §5.2.1]. Les détails de la traduction peuvent être trouvés dans Chapitre 7. Afin d'éviter toute confusion avec la signature Lambdapi Σ, nous notons $\Theta^{\mathcal{S}}$ l'ensemble des sortes SMT-LIB, $\Theta^{\mathcal{F}}$ l'ensemble des symboles de fonctions, et $\Theta^{\mathcal{X}}$ l'ensemble des variables.

Alethe ne prend pas en charge les sortes `Array` et `String`. De plus, notre reconstruction ne prend pas encore en charge `Bitvector` et `Real`. Notre traduction repose sur les fonctions suivantes :

- $\mathcal{D}$ traduit les déclarations de sortes et de fonctions de $\Theta^{\mathcal{S}}$ et $\Theta^{\mathcal{F}}$ en constantes de la signature Σ,
- $\mathcal{S}$ associe à chaque sorte SMT un type de Σ,
- $\mathcal{E}$ traduit les expressions SMT en termes de $\lambda\Pi/\equiv$,
- $\mathcal{C}$ traduit une liste de commandes i.e. étapes $c_1 \dots c_n$ de la forme $i.\ \Gamma \rhd\ \varphi\ (\mathcal{R}\ P)[A]$ en jugements de typage $\Gamma \vdash_\Sigma i := M : N$.

Exemple. Nous illustrons l'utilisation de ces quatre fonctions pour la traduction d'Alethe vers Lambdapi, à partir des définitions du problème d'entrée du Code 1.

```
1 symbol U : Set;
2 symbol a : El U;
3 symbol b : El U;
4 symbol p : El U → Prop;
```

La fonction $\mathcal{D}$ introduit alors dans Σ des constantes correspondant à ces symboles :

$$\texttt{U} : \texttt{Set} \qquad \texttt{a} : \texttt{El}\,\texttt{U} \qquad \texttt{b} : \texttt{El}\,\texttt{U} \qquad \texttt{p} : \texttt{El}\,\texttt{U} \to \texttt{Prop}.$$

La fonction $\mathcal{S}$ associe par exemple $\mathcal{S}(\texttt{U}) = \texttt{El}\,\texttt{U}$ et $\mathcal{S}(\texttt{Bool}) = \texttt{Prop}$. Enfin, la fonction $\mathcal{E}$ traduit des expressions en termes de $\lambda\Pi/\equiv$, par exemple $\mathcal{E}(\texttt{not (p b)})$ est traduit en $\neg(p\,b)$.

RECONSTRUCTION DES PREUVES DU PREMIER ORDRE

Nous donnons ici un bref aperçu de la stratégie de reconstruction adoptée dans nos travaux. Les détails figurent au Chapitre 9 pour la logique du premier ordre et au Chapitre 10 pour l'arithmétique.

Pour les règles de la logique du premier ordre, la reconstruction suit une stratégie en deux niveaux. D'abord, les règles tautologiques sont reconstruites par des lemmes génériques sur les clauses, permettant de factoriser les preuves et de réduire la duplication. Par exemple, le Code 3 montre la reconstruction de la preuve du Code 1. L'étape `t1` peut-être reconstruit directement en Lambdapi en encodant le lemme générique:

$$\texttt{equiv_pos2} : \Pi[a : \texttt{Prop}]\Pi[b : \texttt{Prop}], \texttt{Prf}^{\bullet}(\neg(a = b) \vee \neg a \vee b \vee \blacksquare)$$

et similairement pour la congruence. Ensuite, les règles de déduction comme la `resolution` utilisés aux étapes `t3` et `t4` sont reconstruites par un script de preuve où nous définissons un lemme générique pour la résolution binaire (Théorème 3). Une hyper-résolution est ainsi reconstruite par l'application en chaine de résolution binaire via le langage de tactique de Lambdapi.

```
1  symbol p_2 ≔ (p b);
2  symbol p_4 ≔ (p a);
3  symbol p_6 ≔ (p a = p b);
4
5  symbol a2 : Prf• (¬ (p b) ∨ ■);
6  symbol a1 : Prf• (a = b ∨ ■);
7  symbol a0 : Prf• (p a ∨ ■);ls
8
9  opaque symbol t1 : Prf• (¬ p_6 ∨ ¬ p_4 ∨ p_2 ∨ ■) ≔
10 begin
11 apply equiv_pos2;
12 end;
13
14 opaque symbol t2 : Prf• (p_6 ∨ ■) ≔
15 begin
16 apply ∨ i1;
17 apply feq p (headcl a1);
18 end;
19
20 opaque symbol t3 : Prf• (p_2 ∨ ■) ≔
21 begin
22 have t1_t2 : Prf• (¬ p_4  ∨ p_2 ∨ ■) {
23     apply disj_resolutionN1 0 0 t1 t2 ⊤ᵢ ⊤ᵢ (eq_refl _);
24 };
25 have t1_t2_a0 : Prf• (p_2 ∨ ■) {
26     apply disj_resolutionN1 0 0 t1_t2 a0 ⊤ᵢ ⊤ᵢ (eq_refl _);
27 };
28 refine t1_t2_a0;
29 end;
30
31 opaque symbol t4 : Prf• ■ ≔
32 begin
33 have a2_t3 : Prf• ■ {
34     apply disj_resolutionN1 0 0 a2 t3 ⊤ᵢ ⊤ᵢ (eq_refl _);
35 };
36 refine a2_t3;
37 end;
```

Code 3: Traduction de la preuve Code 1

L'arithmétique linéaire est un domaine où les solveurs SMT recourent à des algorithmes sophistiqués. Les preuves Alethe contiennent souvent des étapes qui impliquent des calculs arithmétiques ou des normalisations. Les vérifier pas à pas sur le plan logique serait trop coûteux. La thèse adopte alors une approche par *réflexion*. L'idée est de représenter les expressions arithmétiques sous une forme *réifiée* i.e. une algèbre interne, de définir une procédure de normalisation

Tableau 1: Résultats cvc5 et Carcara (format Succès–Erreur–Temps).

Nom	Logique	Échantillons	cvc5	Élaboration
UltimateAutomizer	LIA	153	153 - 0 - 0	146 - 7 - 0
sledgehammer	UFLIA	1,521	1351 - 0 - 170	1279 - 72 - 1
20220307-SMPT	QF_LIA	1,568	1522 - 0 - 46	1520 - 26 - 0
20170428-Barrett	UF	1,585	1512 - 0 - 73	1491 - 22 - 0
sledgehammer	UF	1,403	1269 - 0 - 134	1206 - 68 - 0

sur ces objets, puis de prouver, dans Lambdapi, un théorème de correction indiquant que la normalisation préserve la sémantique. Ainsi, au lieu de reconstruire un long raisonnement arithmétique, on reconstruit la preuve que la procédure de normalisation est correcte, puis on l'applique.

CORRECTION DE LA TRADUCTION

La thèse établit un argument de correction montrant que la traduction préserve la conséquence sémantique. Intuitivement, si une étape Alethe est valide au sens de la sémantique Alethe/SMT-LIB, alors la preuve construite dans Lambdapi démontre une proposition correspondante dans l'encodage.

Théorème (Correction de la reconstruction)**.** Pour tout préfixe valide d'une trace de preuve Alethe $P = [c_1, \ldots, c_i]$, si

$$c_i.\Gamma \rhd \varphi_1 \ldots \varphi_m\,(\mathcal{R}\,P)[A]$$

alors

$$\vdash_\Sigma c_i : \mathtt{Prf}^\bullet(\mathcal{E}(\varphi_1) \vee \cdots \vee \mathcal{E}(\varphi_m) \vee \blacksquare) \coloneqq M_i.$$

où M_i est un terme de preuve de la clause traduite.

EVALUATION

Nous évaluons notre chaîne complète (Image 2) pour la reconstruction de preuves SMT en Lambdapi à l'aide des tests de performance fournis par la SMT-COMP [17]. Nous utilisons des échantillons des familles SMT-LIB couvrant exactement les fragments pris en charge: UF, QF_UF, LIA, UFLIA et QF_LIA. Nous présentons ici un court extrait de nos résultats; les détails de la configuration et l'ensemble des mesures sont présentés aux Chapitre 12 et Partie A.5. Nous nous concentrons en particulier sur le corpus *Sledgehammer*, qui reflète un cas d'usage central: la décharge automatique de sous-buts dans un assistant de preuve via un solveur SMT, puis la validation indépendante du certificat produit. Un échantillon représentatif de chaque logique prise en charge est présenté dans ces tableaux. Nous reportons les résultats au format Succès–Erreur–Temps Écoulé (S–E–T). Une limite de temps de 30 secondes est donnée pour cvc5 et Carcara, et de 20 secondes pour Lambdapi.

Tableau 2: Traduction et vérification Lambdapi (Succès–Erreur–Temps).

Nom	Logique	Traduction	Vérification
UltimateAutomizer	LIA	90 - 51 - 0	71 - 19 - 0
sledgehammer	UFLIA	1081 - 52 - 0	897 - 170 - 14
20220307-SMPT	QF_LIA	1397 - 7 - 0	702 - 691 - 5
20170428-Barrett	UF	1419 - 0 - 0	1284 - 121 - 14
sledgehammer	UF	1099 - 7 - 0	959 - 117 - 23

Tableau 3: Temps de vérification de Lambdapi (ms).

Nom	Moyenne	Min	Q1	Médiane	Q3	Max
UltimateAutomizer	238	128	176	188	268	1049
sledgehammer	665	114	216	348	641	Temps expiré
20220307-SMPT	170	97	117	121	132	4339
20170428-Barrett	389	125	190	227	329	20323
sledgehammer	684	113	208	355	759	Temps expiré

Le Tableau 1 montre que, dans la grande majorité des cas, cvc5 produit un certificat de preuve et que Carcara parvient ensuite à l'élaborer; ce qui permet sa traduction et sa vérification. Le Tableau 2 résume les échantillons qui ont pu être élaborés, puis traduits, et ensuite vérifiés par Lambdapi, et le Tableau 3 fournit des statistiques sur les temps de vérification. Sur le corpus *Sledgehammer*, nous obtenons un taux de réussite de vérification de près de 90 %. Nous examinons les goulots d'étranglement dans la conclusion de Chapitre 12.

CONCLUSION ET PERSPECTIVES

Cette thèse contribue à un écosystème SMT plus fiable et plus interopérable en proposant un cadre de reconstruction qui traduit des traces Alethe en preuves vérifiables par le noyau de Lambdapi, au moyen d'un encodage modulaire de la logique SMT et de techniques de reconstruction par réflexion. L'évaluation sur des familles SMT-LIB représentatives montre que la chaîne proposée permet de vérifier efficacement un grand nombre de preuves. En reconstruisant des preuves Alethe dans Lambdapi, on obtient des certificats vérifiables, indépendants et potentiellement exportables vers d'autres assistants. Nos expérimentations ont également permis d'identifier des anomalies ensuite corrigées, notamment dans la génération de certificats de cvc5. Au-delà de ces expérimentations, nous avons utilisé ces travaux pour vérifier des preuves issues de l'assistant de preuve TLAPS, que nous décrivons au Chapitre 12. Notre outil a ainsi permis de mettre en évidence des erreurs dans certaines preuves de TLAPS, dues à un bogue dans l'encodage SMT de la théorie des ensembles.

Plusieurs perspectives s'ouvrent naturellement. Étendre la couverture à d'autres théories importantes (e.g. bit-vectors) renforcerait encore l'universalité du cadre. Approfondir l'intégration avec d'autres assistants (export vers Rocq, Lean, etc.) contribuerait à faire d'Alethe un véritable format d'échange de preuves, au service d'une vérification formelle plus transparente et plus digne de confiance.

ACKNOWLEDGMENTS

This thesis would not have been possible without the invaluable guidance of Professor Frédéric Blanqui, who introduced me to Lambdapi and implemented several modifications to the tool that were essential for the reconstruction of SMT proofs. Working with him on Lambdapi has also instilled in me a strong desire to contribute to its long-term development. I am also profoundly grateful to him for welcoming me into the EuroProofNet research network, which offered me many opportunities for collaboration and for presenting my work.

I would like to express my deepest appreciation to Professor Haniel Barbosa for sharing his deep expertise in SMT solvers and SMT proof formats , and for the strong interest he has consistently shown in my work, which I greatly appreciated. I am equally indebted to his student, Bruno Andreotti, the main developer of Carcara, for his constant availability, generosity in our exchanges, and for his unfailing reactivity to my many requests.

I wish to extend my sincere thanks to Professor Jean-Paul Bodeveix, who has supported and followed my academic path since my Master's studies. It has been a pleasure and an honour to collaborate with him in the context of the ANR ICSPA project.

I would also like to thank the members of the ICSPA project for their support and for the many fruitful discussions we shared. I am also particularly grateful to Mathias Fleury, Hanna Lachnitt, Chantal Keller, and Hans-Jörg Schurr, with whom I had the pleasure of working and exchanging ideas. I would also like to thank the members of the VerDis team, especially my four "thesis siblings": Thomas Bagrel, Ghilain Bergeron, Sarah Depernet, and Vincent Trélat, for their support, discussions, and companionship. I am especially grateful to Sophie Tourret for including me in the Carma project, which greatly facilitated the development of this work in collaboration with Professor Haniel Barbosa and Bruno Andreotti. I also wish to thank the members of the Deducteam, in particular Theo Winterhalter, Ciaran Dune, Melanie Tapproge, Thomas Traversié, Rishikesh Vaishnav, Nicolas Margulies and Bruno Barras, for the stimulating scientific environment they helped create.

I am deeply indebted to my family, particularly to my parents, for their unwavering support throughout these years. I would also like to warmly thank my former colleague Didier Plaindoux for his encouragement over the course of my studies and research. Finally, I would like to thank Xavier Van de Woestyne for creating the image used on the cover of this thesis, which gives a distinctive visual identity to this work.

I would also like to express my sincere gratitude to the ANR ICSPA project directed by Professor Catherine Dubois for funding my PhD.

Its financial support made this research possible and provided the conditions necessary for me to carry out this work in a stimulating and collaborative scientific environment.

CONTENTS

I CONTEXT 1

1 INTRODUCTION 3
- 1.1 Thesis outline 5

2 LAMBDAPI 9
- 2.1 The λΠcalculus modulo 9
- 2.2 Signature rules 9
- 2.3 Rewriting 10
  - 2.3.1 β-reduction 10
  - 2.3.2 Σ-reduction 11
- 2.4 Typing system 11
- 2.5 Prelude 15
- 2.6 Constructive connectives, quantifiers 16
  - 2.6.1 Inductive data structure 17
- 2.7 Language extensions 19
  - 2.7.1 Modifiers on symbols 19
  - 2.7.2 Metatactics 21
  - 2.7.3 Implicit Coercions 22

3 SATISFIABILITY MODULO THEORIES 25
- 3.1 SMT Solvers 25
- 3.2 Terms and Sorts 25
- 3.3 semantics 27
- 3.4 Theories 29
  - 3.4.1 The Core theory 30
  - 3.4.2 Uninterpreted Symbols 30
  - 3.4.3 The theory of linear arithmetic 30
  - 3.4.4 Bitvectors 30
- 3.5 Solving SMT Problems 33
- 3.6 SMT proof 36

4 ALETHE: A NEW STANDARD FOR PROOF TRACE REPRESENTATION 39
- 4.1 The Alethe Proof Format 39
- 4.2 The Alethe Language 39
  - 4.2.1 Many-Sorted First-Order Logic 40
  - 4.2.2 Steps 40
  - 4.2.3 Subproofs and Lemmas 41
- 4.3 The concrete syntax of Alethe 43
- 4.4 Overview of Alethe Rules 46
- 4.5 Checking Alethe Proofs 51
  - 4.5.1 Abstract procedure 51
  - 4.5.2 Challenges in Proof Verification 53

5 ELABORATION OF ALETHE PROOFS 55
- 5.1 Elaboration with Carcara 55
  - 5.1.1 Eliminating clause reordering steps 55
  - 5.1.2 Elaboration of resolution steps 56
  - 5.1.3 Elaboration of `lia_generic` steps 57

5.2 Elaboration of simplifications with RARE 58
6 RELATED WORKS 61
6.1 The Eunoia proof format 61
6.2 Proof reconstruction for hammers 62
6.3 SMT integration in proof assistants 63
6.4 Proof Checkers for SMT Solvers 64
6.5 The Unverified Backend Problem 64

II CONTRIBUTIONS 69
7 ENCODING THE LOGIC OF SMT IN LAMBDAPI 71
7.1 A Prelude Encoding for Alethe 71
7.2 Representing Linear Integer Arithmetic (**LIA**) 72
7.3 Representing Linear Arithmetic (**LA**) 75
7.4 SMT-LIB helper operators 77
7.5 Encoding clauses 78
8 RECONSTRUCTING SMT PROOFS 81
9 RECONSTRUCTING FOL PROOFS 83
9.1 Tautologous rules and simple deduction 83
9.2 Simplification rules 89
9.3 Quantifiers 91
9.4 Skolemization 91
9.5 Alethe subproofs 93
9.6 Proof by reflection 94
9.7 N-ary rules 96
9.8 The `evaluate` cvc5 rule 98
9.9 The `contraction` rule 103
10 RECONSTRUCTING LINEAR ARITHMETIC PROOFS 109
10.1 Reconstruction of Linear Integer Arithmetic 109
10.2 General scheme of the reflective reconstruction 109
10.3 Reflection with inner normalization 110
10.3.1 Representation of integers 110
10.3.2 AC inner normalization 111
10.4 Reflection with outer normalization 115
10.4.1 Reflecting integers 115
10.4.2 Outer normalisation procedure 117
10.4.3 Correctness of Reified Normalization 118
10.5 Reconstruction of Linear Real Arithmetic 121
11 TRANSLATION CORRECTNESS 123
12 EVALUATION 127
12.1 Benchmark Suite and Evaluation Settings 127
12.2 Benchmark settings 127
12.3 Parallel reconstruction and checking 129
12.4 Evaluation over SMT-LIB benchmarks 130
12.5 Reconstruction of TLA$^+$ proofs 135
12.6 Evaluation Summary 138
13 CONCLUSION 141
13.1 contributions 141
13.2 Future Perspectives 142
13.3 Summary 145

III APPENDIX 149

A APPENDIX 151

A.1 Confluence of the rewriting rules of integers and positive binary number 151

A.2 Arithmetic definitions 152

A.3 Supported Alethe rules 155

A.4 Translation example with more Alethe features 156

A.5 Complete benchmark results 161

BIBLIOGRAPHY 171

LIST OF FIGURES

Figure 1 Lambdapi as proof interchange 5
Figure 2 Typing rules of λΠ/ ≡ 12
Figure 3 The main loop of an SMT solver. 33
Figure 4 Example proof output. 44
Figure 5 The proof grammar. 44
Figure 6 A proof snippet before and after elimination of `reordering` steps 56
Figure 7 Resolution-step elaboration 57
Figure 8 Overview of the approach 58
Figure 9 Cantor theorem in TLA$^+$ 66
Figure 10 Proof obligation of step ⟨1⟩6. 66
Figure 11 TLAPS verification pipeline 68
Figure 12 Representative Alethe translations 84
Figure 13 Resolution rule 84
Figure 14 Translation of `not_simplify` 90
Figure 15 Translating the `forall_inst` command. 91
Figure 16 Translating the `sko_forall` command. 92
Figure 17 Translation of subproof and bind 94
Figure 18 Translating the `pos_and` command. 98
Figure 19 Normalization by reflection 110
Figure 20 Pipeline of the benchmarks 127
Figure 21 Parallel translation and checking. 129
Figure 22 Checking times across SMTLIB samples 134
Figure 23 Checking times across TLAPS samples 138
Figure 24 Verifying TLAPS proof with Lambdapi. 145
Figure 25 List of Alethe UF rules supported in our reconstruction (alphabetical order). 155
Figure 26 List of LIA/LA rules supported in our reconstruction (alphabetical order). 156
Figure 27 List of RARE rules supported in our reconstruction (alphabetical order). 156
Figure 28 cvc5 times. 165
Figure 29 Carcara elaboration times. 167
Figure 30 Translation times for small problems 169
Figure 31 Translation times for large problems 170

LIST OF TABLES

Table 1 Function symbols of the real and integers theory. 31
Table 2 SMT-LIB Bit-Vector (BV) theory function symbols (sizes n, m, k, i, j are numerals). 32

Table 3 Linear arithmetic rules in Alethe 49
Table 4 Specification of `distinct_elim` 78
Table 5 N-ary operators rules 96
Table 6 Summary of benchmark. The format S – F – T indicates respectively the number of success (S), failed (F) and timeout (T) samples. 130
Table 7 Translation and checking results. The format S – F – T indicates respectively the number of success (S), failed (F) and timeout (T) samples. 132
Table 8 Translation and checking results. The format S – F – T indicates respectively the number of success (S), failed (F) and timeout (T) samples. 133
Table 9 Lambdapi checking times 135
Table 10 Checking times with AC 136
Table 11 Summary of benchmark. The format S – F – T indicates respectively the number of success (S), failed (F) and timeout (T) samples. 136
Table 12 Translation and checking results. The format S – F – T indicates respectively the number of success (S), failed (F) and timeout (T) samples. 137
Table 13 Lambdapi checking times (in ms). 137
Table 14 Summary of benchmark. 162
Table 15 cvc5 and Alethe elaboration results. 163
Table 16 cvc5 time measurements 164
Table 17 Carcara elaboration times 166
Table 18 Translation results. 168

LIST OF LISTINGS

Listing 1 A Simple SMT Problem 4
Listing 2 Alethe Proof of Listing 1 Found by cvc5. 5
Listing 3 Concrete syntax of Example 3 16
Listing 4 Example of builtins metatactics 21
Listing 5 Running example and Alethe proof 40
Listing 6 Example of a Linear Arithmetic Input Problem 48
Listing 7 Alethe Proof of Listing 6 found by cvc5 48
Listing 8 Elaboration of `lia_generic` 57
Listing 9 An Eunoia proof example 61
Listing 10 Proof obligation of step ⟨1⟩6. in Cantor theorem 67
Listing 11 Trace from encoded in Lambdapi. 89
Listing 12 Alethe subproof example. 93
Listing 13 Proof trace using cvc5 evaluate 99

Listing 14 Usage of Tauto 103
Listing 15 Lambdapi proof of List. 7 114
Listing 16 LIA reconstruction example 120
Listing 17 TRS rules for Z and P 151
Listing 18 SMT Input Problem 157
Listing 19 Alethe proof of the input example 157
Listing 20 Lambdapi Translation of Listing 19 158

ACRONYMS

SMT Satisfiability Modulo Theories

ATP Automated Theorem Proving

SAT Boolean Satisfiability Problem

AC Associative Commutative

Part I

CONTEXT

1 INTRODUCTION

Powerful modern automated theorem provers are widely used now to prove security and safety properties of computer systems. These tools are capable of handling complex logical reasoning tasks with a degree of automation and efficiency that complements or extends manual mathematical proof development. Their ability to automatically generate and check formal proofs has made them indispensable for ensuring the reliability and safety of both hardware and software systems, where correctness is paramount.

The Satisfiability Modulo Theories (SMT) solvers [12, 28] rank among the most advanced automated reasoning tools currently available and have become indispensable in a wide variety of formal reasoning tasks. These include program verification and interactive theorem proving. Their efficiency and expressive power enable them to handle complex logical proofs and verifications with speed. At the same time, the growing reliance on SMT solvers raises a fundamental question of trust: *can we be confident that the results produced by these highly complex SMT solvers are correct?*

Despite their central role, SMT solvers are large software systems with complex architectures [12], often comprising hundreds of thousands of lines of code. This complexity makes it extremely difficult to guarantee their correctness. The annual SMT-COMP competition [17, 101], regularly uncovers cases where different solvers disagree on the satisfiability status of the same benchmark. State-of-the-art SMT solvers have been found to have bugs [29] due in part to error-prone optimizations, despite the best efforts of developers. Such discrepancies highlight the practical risk of relying blindly on solver outputs in safety-critical applications.

A natural response to this problem is to attempt the certification of the solver itself. However, formally verifying such a large and complex codebase is highly impractical [83, §2.2.1]. Simplifying a solver's design to ease certification would necessarily come at the expense of performance, which is a key factor in their adoption. Moreover, certification tends to "freeze" the implementation: once a system is verified, the integration of new features, heuristics, or optimizations becomes prohibitively costly, since each change would require a new certification effort. Consequently, directly certifying solvers in a late lifecycle of development might not scale with the pace of their development.

A first alternative is to develop a certified solver in a proof assistant environment such as the IsaSAT solver [47, 49] that is a fully verified Boolean Satisfiability Problem (SAT) solver developed in Isabelle/HOL [77]. This alternative offers strong guarantees about the correctness of the implementation itself, since the solver is extracted from a machine-checked formalization. Moreover, it has been demon-

strated to achieve competitive performance in SAT solving [48]. In this respect, IsaSAT advances the state of the art by providing a verified implementation of SAT solving. However, the restriction to SAT limits its applicability in domains where SMT theories reasoning is required.

A second alternative line of research is *proof logging* [68], where the approach is to certify the *results* rather than the solver implementation. Proof logging is a technique for automatically reviewing the reasoning steps of a proof engine by a separate tool. This is achieved by requiring the solver to produce a proof, or certificate, of its output. A proof trace or, more generally, certificate formally records the logical reasoning that leads to a solution, enabling independent verification by a separate proof checker. Essentially, the task of proof checking is conceptually simpler and typically much faster than solving itself. Moreover, a well designed trace format allows traces to be generated and checked with low performance overhead. This decouples the trust in solver results from the correctness of the solver's implementation, thereby providing a more practical path toward trustworthy automated reasoning. It is then the checker of the proof trace that becomes the critical link for establishing confidence in the correctness of a verdict, and establishing soundness of a proof checker is much easier than verifying an SMT solver. DRAT proofs [54] is a standard for verifying unsatisfiability proofs emitted by modern SAT solvers. The independent checker rupee [81] verifies DRAT proofs produced by modern SAT solvers. An attractive way of building and verifying a trace checker is to have a proof assistant interpret a trace and produce a proof that is accepted by the kernel of the proof assistant. This method is known as *proof reconstruction* [26]. For example, this approach has been adopted in several proof assistants that employ *hammers* for discharging goals to Automated Theorem Proving (ATP) systems, including SMT solvers. Examples include Sledgehammer [20] for Isabelle/HOL and CoqHammer [39, 40] for Coq. The hammer translates the conjecture and facts supplied by the user or harvested from the context to the input language of the back-end, invokes it, and, in the case of success, attempts to reconstruct the proof in the logic of the proof assistant, based on a trace of the proof found by the back-end.

Alethe [87, 90] is an established SMT proof format generated by the solvers veriT [28] and cvc5 [12]. As the example Listing 2 shows, the format is close to SMT-LIB and facilitates proof production. However, it lacks a standalone checker, which harms its usability and hinders its adoption. Moreover, the coarse-grained steps can be too expensive to check and lead to verification failures.

```
1 (set-logic UF)
2 (declare-sort U 0)
3 (declare-fun a () U)
4 (declare-fun b () U)
5 (declare-fun p (U) Bool)
6 (assert (p a))
7 (assert (= a b))
8 (assert (not (p b)))
9 (check-sat)
```

```
10 (get-proof)
```

Listing 1: A Simple SMT Problem

```
1 (assume a0 (p a))
2 (assume a1 (= a b))
3 (assume a2 (not (p b)))
4 (step t1 (cl (not (= (p a) (p b))) (not (p a)) (p b)) :rule equiv_pos2)
5 (step t2 (cl (= (p a) (p b))) :rule cong :premises (a1))
6 (step t3 (cl (p b)) :rule resolution :premises (t1 t2 a0))
7 (step t4 (cl) :rule resolution :premises (a2 t3))
```

Listing 2: Alethe Proof of Listing 1 Found by cvc5.

This thesis addresses these challenges by introducing a translation framework from Alethe proofs to Lambdapi [37], a proof assistant for the λΠ-calculus modulo rewriting (λΠ/ ≡). The λΠ/ ≡-calculus combines dependent types, which enable precise encodings of object logic, with modulo theory which extends reasoning by allowing rewrite rules for defining theories and deductions. In this way, the λΠ/ ≡-calculus provides a meta-logic capable of expressing object logic, axioms, inference rules, and proofs of various theorem provers. Lambdapi also supports exporting proofs to other systems (Figure 1), providing a bidirectional connection between different proof environments. This has been one of our principal reasons for selecting Lambdapi, since it enables the conversion and sharing of Alethe proofs with other logical systems such as Rocq and Isabelle/HOL.

In this work, we identify common challenges and develop general encoding strategies. Based on these strategies, we present a modular encoding in Lambdapi of selected Alethe inference rules. We then empirically evaluate the effectiveness of this approach through the automated generation of proof certificates verifiable within the Lambdapi proof assistant.

1.1 THESIS OUTLINE

This chapter has introduced the challenges of ensuring trustworthiness in SMT solvers, highlighting the need for verifiable proofs to address the complexity and potential incorrectness of SMT solvers.

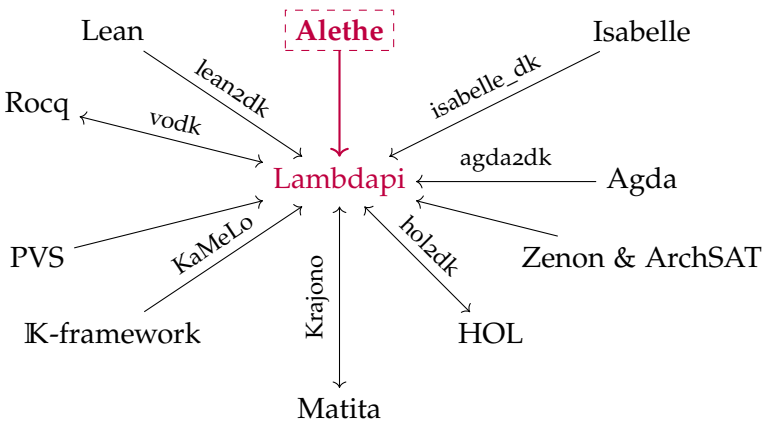


Figure 1: Lambdapi, an assembly language for proof systems.

While SMT solvers are widely used in formal methods, their outputs are often taken for granted, despite the risks of errors in large, optimised codebases. The goal of this thesis is to provide a robust framework for certifying SMT proofs by translating them into a dependently typed setting, making them both trustworthy and portable across different proof assistants. The thesis is structured as follows:

- In Chapter 2, we introduce Lambdapi, the foundational framework for our work. We present the λΠ-calculus modulo rewriting, its typing system, and key features, including rewrite rules and inductive types. This chapter establishes the formal basis for encoding SMT proofs. It demonstrates why Lambdapi is uniquely suited for this task due to its interoperability with other proof assistants and its support for user-defined rewriting.
- In Chapter 3, we provide background on SMT, covering the SMT-LIB standard, the semantics of first-order logic with theories, and the architecture of modern SMT solvers. We discuss the role of proof production in ensuring solver correctness and motivate the need for a standardised, verifiable proof format like Alethe.
- In Chapter 4, we dive into the Alethe proof format, detailing its design, syntax, and logical rule system including resolution, quantifier instantiation, and theory specific rules (e.g., for linear arithmetic). We explain how Alethe represents proofs as sequences of steps. This chapter also highlights the challenges posed by coarse-grained proof steps and the lack of a standalone checker, which our work addresses.
- In Chapter 5, we describe how preprocessing tools, such as Carcara [3] and RARE [78], refine and elaborate proofs before translation, eliminating redundant steps and preparing them for reconstruction in Lambdapi.
- In Chapter 6, we survey related work, comparing Alethe with other proof formats (e.g., Eunoia) and discussing existing approaches to integrating SMT solvers with proof assistants. We position our work within the broader landscape of proof certification for SMT solvers.
- In Chapter 7, we lay the groundwork for our encoding by defining how SMT logic can be represented in Lambdapi. We develop a prelude that captures SMT-LIB's sorts, terms, and formulas, and show how to encode theories like Linear Integer Arithmetic (LIA) and Linear Arithmetic (LA). This chapter bridges the gap between SMT's many-sorted first-order logic and Lambdapi's dependent types.
- In Chapter 8, 9 and 10, we present the core of our reconstruction methodology. Chapter 9 focuses on reconstructing first-order proofs, including tautologies, resolution, and quantifier

handling, while Chapter 10 extends this to linear arithmetic proofs. We introduce reflective techniques for normalising arithmetic expressions and prove the correctness of our translations, ensuring that reconstructed proofs faithfully represent the original SMT reasoning.

- In Chapter 11, we formalise the correctness of our translation, proving that the encoding preserves semantic entailment. This chapter provides the theoretical guarantees that underpin our approach.
- In Chapter 12, we shift to practical evaluation, presenting benchmarks and case studies that demonstrate the efficiency and scalability of our method. We demonstrate how our toolchain validates proofs from real-world applications, including TLAPS proofs [36], and discuss optimisations for parallel checking.
- Finally, in Chapter 13, we conclude by summarising our contributions and discussing limitations and future directions. We highlight the potential for extending our work to additional theories (e.g., bit-vectors) and integrating with more proof assistants, ultimately aiming to establish Alethe as a standard for proof exchange in formal methods.

2 LAMBDAPI

2.1 THE λΠCALCULUS MODULO

The λΠ-calculus modulo rewriting (λΠ/ ≡) extends the Edinburgh Logical Framework (LF) [53] by allowing user-defined rewrite rules at both the term and type levels. In this framework, all terms and types are identified modulo the congruence relation generated by standard β-reduction together with the specified rewrite rules. λΠ/ ≡ is implemented in the Lambdapi proof language [37]. Lambdapi, a successor of the Dedukti proof language [5, 84], is an implementation of λΠ/ ≡; Lambdapi differs primarily by offering an integrated proof tactic language. However, it can still import and export Dedukti theories to maintain compatibility. Lambdapi, similarly to Dedukti, is designed to facilitate the exchange of proofs between systems. Lambdapi/Dedukti was used to translate [95] the Matita arithmetic library to several systems, including Coq and PVS, export [23] the HOL Light standard library to Coq, and verify Event-B proofs [52].

2.2 SIGNATURE RULES

We start by defining the basic elements of the λΠ/ ≡ calculus and giving their syntax. The terms of the λΠ-calculus modulo are the same as in λΠ-calculus (LF). The Terms of λΠ/ ≡ are divided into three levels: Objects (denoted by M and N), Types (denoted by A and B), and Kinds (denoted by K). The syntax of λΠ/ ≡ is given by the following grammar:

Objects	$M, N ::= c \mid x \mid \lambda x : A, M \mid M\ N$
Types	$A, B ::= a \mid \Pi x : A, B \mid A\ M$
Kinds	$K ::= \texttt{TYPE} \mid \Pi x : A, K$
Terms	$t, u ::= M \mid A \mid K \mid \texttt{KIND}$

where c and a are constants, and x is a variable (ranging over disjoint sets). Dependent products $\Pi x : A, B$ (respectively $\Pi x : A, K$) are simply written $A \to B$ when x does not occur in B (respectively K), $\lambda x : A, M$ is an abstraction, and $M\ N$ (respectively $A\ M$) is an application.

Definition 2.2.1 (Substitutions)**.** A substitution σ is a function written $[x_1 \leftarrow N_1, \ldots, x_n \leftarrow N_n]$, from the variables to the terms with a finite domain. We write $t\sigma$ the term t where the variables are replaced by their image by σ.

Contexts Γ are finite sequences used to specify the type of the free variables. Signatures (denoted Σ) serve as the *global context* and are

used to declare the constants and rewrite rules considered by the users.

$$\begin{array}{ll} \text{Contexts} & \Gamma ::= \langle\rangle \mid \Gamma, x : A \\ \text{Signature} & \Sigma ::= \langle\rangle \mid \Sigma, c : A \mid \Sigma, a : K \mid \Sigma, M \hookrightarrow N \\ & \qquad \mid \Sigma, A \hookrightarrow B \mid c \coloneqq A : M \end{array}$$

A *signature* is a finite sequence of *assumptions* $c : A$ (respectively $a : K$), indicating that constant c is of type C (respectively kind K), *definitions* $c \coloneqq A : M$, indicating that c has the value M and type A. We write $\langle\rangle$ when contexts or signatures are empty. Rewrite rules are pairs $M \hookrightarrow N$ (respectively $A \hookrightarrow B$) , where the head symbol of M (respectively A) is a constant and where free variables of N (respectively B) occur in M (respectively A). Rules must also preserve typing (subject-reduction property) as detailed in Section 2.4, that is, if an instance of a left-hand side has some type, then the corresponding instance of the right-hand side should have the same type. A symbol with a definition cannot appear at the head of M (or A) in a rewrite rule.

Definition 2.2.2 (Delta reduction)**.** Let Σ be a signature containing constant definitions of the form $c \coloneqq t$. We write

$$c \quad \to_\delta \quad t$$

to denote *delta reduction*, that is, the replacement of the constant c by its defining term t.

In Lambdapi, delta reduction is applied implicitly during computation. This means that whenever a defined constant appears in a term, it is automatically unfolded to its definition without requiring an explicit reduction step.

Example 1. Suppose the context contains the definition

$$\mathsf{plus_one} \coloneqq \lambda n.\, n + 1.$$

Then we have the delta reduction

$$\mathsf{plus_one}\ 2 \quad \to_\delta \quad (\lambda n.\, n + 1)\ 2 \quad \to_\beta \quad 2 + 1.$$

2.3 REWRITING

In the $\lambda\Pi/\equiv$-calculus we distinguish two kinds of rewriting.

2.3.1 β-*reduction*

The first kind of rewriting is β-reduction, which is defined as usual.

Definition 2.3.1 (β-reduction)**.** The β-reduction relation $\longrightarrow_\beta$ is the smallest relation on terms containing $(\lambda x : A, u)v \longrightarrow_\beta u[x \leftarrow v]$ for any A, u, v with type A and closed by subterm reduction, meaning that if $t \longrightarrow_\beta t'$, then the reduction also holds inside any larger term context containing t.

Notation. We write $\longrightarrow_\beta^*$ for the reflexive and transitive closure of $\longrightarrow_\beta$ and $\equiv_\beta$ for the congruence generated by $\longrightarrow_\beta$.

2.3.2 Σ-*reduction*

The second kind of rewriting is Σ-reduction, the relation generated by the rewrite rules of a signature Σ.

Definition 2.3.2 (Σ-reduction)**.** Let Σ be a global context. The Σ-reduction relation $\longrightarrow_\Sigma$ is the smallest relation on terms containing $u \longrightarrow_\Sigma v$ for each rule $u \hookrightarrow v \in \Sigma$ and closed by substitution and subterm reduction.

Notation. We write:

- $\longrightarrow_\Sigma^*$ the reflexive and transitive closure of $\longrightarrow_\Sigma$;
- $\equiv_\Sigma$ the congruence generated by $\longrightarrow_\Sigma$;
- $\longrightarrow_{\beta\Sigma}$ for $\longrightarrow_\beta \cup \longrightarrow_\Sigma$;
- $\longrightarrow_{\beta\Sigma}^*$ the reflexive and transitive closure of $\longrightarrow_{\beta\Sigma}$;
- $\equiv_{\beta\Sigma}$ for the equivalence relation generated by $\longrightarrow_{\beta\Sigma}$.

Lambdapi implements untyped rewriting because it will be inefficient to check the well-typedness of the substitution at each reduction step.

Definition 2.3.3 (Linearity)**.** We also define the following properties:

- A term is linear if no variable occurs twice in it;
- A rewrite rule $M \hookrightarrow N$ is left-linear if M is linear.

Example 2 (Non-linear rule)**.** The rule $c\ x\ x\ y \hookrightarrow u$ is non-left linear because the variable x occurs twice on the left of the arrow.

2.4 TYPING SYSTEM

We now give the typing rules of the λΠ/≡-calculus in Figure 2. A Lambdapi typing judgment $\Gamma \vdash_\Sigma t : A$ asserts that term t has type A in the context Γ and the signature Σ.

Definition 2.4.1 (Well-Typed Term)**.** We say that a term t has a type A in a signature Σ and a context Γ if the judgment $\Gamma \vdash_\Sigma t : A$ is derivable by the inference rules of Figure 2. We say that a term is well-typed if such an A exists.

Definition 2.4.2 (Well-Formed Local Context)**.** A local context Γ is well-formed with respect to a global context Σ if the judgment $\vdash_\Sigma \Gamma$ is derivable by the inference rules of Figure 2.

Remark 1. The typing rules shown in Figure 2 are similar to those of λΠ-calculus [53, §2], except for the additional rule (Conv) that identifies types modulo $\equiv_{\beta\Sigma}$ instead of just modulo $\equiv_\beta$.

Contexts:

$$\frac{}{\vdash_\Sigma \langle\,\rangle}\ \text{(Empty)} \qquad \frac{\vdash_\Sigma \Gamma \quad \Gamma \vdash_\Sigma A : s}{\vdash_\Sigma \Gamma, x : A}\ \text{(Decl)}\ x \notin \Gamma$$

Terms:

$$\frac{\vdash_\Sigma \Gamma \quad \Gamma \vdash_\Sigma A : s}{\Gamma \vdash_\Sigma c : A}\ \text{(Const)}$$

$$\frac{\vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma \texttt{TYPE} : \texttt{KIND}}\ \text{(Sort)} \qquad \frac{\vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma x : A}\ \text{(Var)}\ x : A \in \Gamma$$

$$\frac{\Gamma \vdash_\Sigma t : A}{\Gamma \vdash_\Sigma c : A \coloneqq t}\ \text{(Delta)}$$

$$\frac{\Gamma, \vdash_\Sigma A : \texttt{TYPE} \quad \Gamma, x : A \vdash_\Sigma B : s \quad \Gamma, x : A \vdash_\Sigma t : B}{\Gamma \vdash_\Sigma \lambda x : A, t : \Pi x : A, B}\ \text{(Abs)}$$

$$\frac{\Gamma \vdash_\Sigma A : \texttt{TYPE} \quad \Gamma, x : A \vdash_\Sigma B : s}{\Gamma \vdash_\Sigma \Pi x : A, B : s}\ \text{(Prod)}$$

$$\frac{\Gamma \vdash t : \Pi x : A, B \quad \Gamma \vdash u : A}{\Gamma \vdash_\Sigma t\ u : B[u \leftarrow x]}\ \text{(App)}$$

$$\frac{\Gamma, \vdash_\Sigma B : u \quad \Gamma \vdash_\Sigma t : A \quad A \equiv_{\beta\Sigma} B}{\Gamma \vdash_\Sigma t : B}\ \text{(Conv)}$$

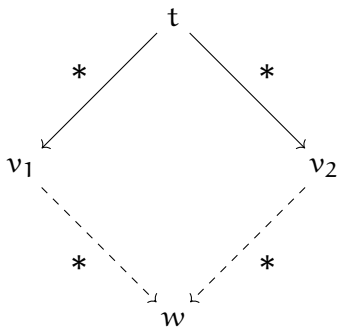
Figure 2: Typing rules of $\lambda\Pi/\equiv$

The main difference between the λΠ-calculus and the λΠ/ ≡-calculus is the presence of the rewrite rules in global contexts. We need to take this into account when typing global contexts. A key feature of any type system is the preservation of typing by reduction: the subject reduction property.

Definition 2.4.3 (Subject Reduction)**.** Let $\longrightarrow_r$ be a binary relation on terms. We say that a global context Σ satisfies the *subject reduction* property for $\longrightarrow_r$ if the following holds: for any well-formed local context Γ with Σ, for any terms t_1 and t_2 such that $t_1 \longrightarrow_r t_2$, and for any term T, if $\Sigma, \Gamma \vdash t_1 : T$ then $\Sigma, \Gamma \vdash t_2 : T$.

In the λΠ/ ≡calculus, allowing arbitrary rewrite rules in the context could breaks properties such as subject reduction for relation $\longrightarrow_{\beta\Sigma}$. Lambdapi requires user rules in $\longrightarrow_{\beta\Sigma}$ for the encoded theories to be *confluent* (Definition 2.4.4) and typing-preserving, even if it does not automatically check that confluence holds.

Definition 2.4.4 (Confluent $\longrightarrow_{\beta\Sigma}$)**.** The relation $\longrightarrow_{\beta\Sigma}$ must be confluent, i.e., whenever $t \longrightarrow^*_{\beta\Sigma} v_1$ and $t \longrightarrow^*_{\beta\Sigma} v_2$, there exists a term w such that $v_1 \longrightarrow^*_{\beta\Sigma} w$ and $v_2 \longrightarrow^*_{\beta\Sigma} w$.



Confluence Property

In other words, confluence is a property of rewriting systems, describing the condition where terms in such a system can be rewritten in more than one way to yield the same result.

Remark 2. External tools, such as CSI [102], can be used to verify the confluence of a set of rewriting rules with a Signature. Lambdapi can export its rules and the signature Σ in a format accepted by CSI.

One approach to prove the confluence property that we will now describe is to check that the critical pairs (Definition 2.4.6) of the rewrite rules in Σ are joinable.

Definition 2.4.5 (Position and Subterm)**.** Let t be a term. We denote $\mathtt{Post}(t)$ the set of positions in t that is a sequence of integers and the subterm $t_{|p}$ of t at position $p \in \mathtt{Post}(t)$ are defined inductively as follows:

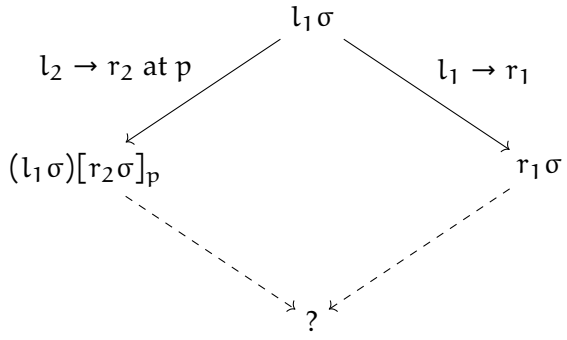
- if t is a variable, then $\mathtt{Post}(t) = \{\epsilon\}$ and $t_{|\epsilon} = t$;
- if $t = f\ t_1 \ldots t_n$, then

$$\mathrm{Post}(t) = \{\epsilon\} \cup \{1.q \mid q \in \mathrm{Post}(t_1)\} \cup \cdots \cup \{n.q \mid q \in \mathrm{Post}(n_1)\},$$

and $t_{|\epsilon} = t$,

and $t_{|i.q} = t_{i|q}$

Definition 2.4.6 (Critical pair)**.** Let $l_i \hookrightarrow r_i$ (for $i = 1, 2$) be two rewrite rules. Suppose that they do not share any variable (variables can be renamed if needed). If there exists $p \in \mathrm{Post}(l_1)$ such that $l_{1|p}$ is not a variable and σ a unifier of $l_{1|p}, l_2$ then we have the following reduction $l_1\sigma \hookrightarrow (l_1\sigma)[l_2\sigma]_p$. We say that the pair $((l_1\sigma)[l_2\sigma]_p, r_1\sigma)$ is a critical pair and that the two rewrite rules overlap.



Definition 2.4.7 (Joinable)**.** Two term $x, y \in \Sigma$ are *joinable* (written $x \downarrow y$), if there exists z such that $x \hookrightarrow^* z$ and $y \hookrightarrow^* z$.

To guarantee the property of confluence of Lambdapi's rewrite system with user-defined rules we leverage local confluence. This property acts as a stepping stone toward full confluence via Newman's lemma (Lemma 1), provided the system is also terminating.

Definition 2.4.8 (Local confluence)**.** A constant $a \in \Sigma$ is *locally confluent* if for all constants $b, c \in \Sigma$ with the rewrite rules $a \hookrightarrow b$ and $a \hookrightarrow b$ we have $b \downarrow c$. The Σ is locally confluent if all its elements are locally confluent.

Theorem 1 (Critical Pair Theorem)**.** The set of rewrite rules R over the signature Σ (and its set of variables $\mathcal{V}$) is locally confluent if and only if its critical pairs are joinable.

Lemma 1 (Newman's Lemma)**.** If the set of rewriting of a signature Σ is *terminating* (i.e., there is no infinite sequence $x_0 \hookrightarrow x_1 \hookrightarrow x_2 \hookrightarrow \cdots$) and *locally confluent* (i.e., for all $s, u, v \in \Sigma$, if $s \hookrightarrow u$ and $s \hookrightarrow v$ then there exists $c \in \Sigma$ such that $u \hookrightarrow^* c$ and $v \hookrightarrow^* c$), then it is confluent (i.e., for all $s, u, v \in \Sigma$, if $s \hookrightarrow^* u$ and $s \hookrightarrow^* v$ then there exists $w \in \Sigma$ such that $u \hookrightarrow^* w$ and $v \hookrightarrow^* w$).

Remark 3. Similar to Confluence, Lambdapi does not verify that a set of rules terminates. However, external tools such as AProVe [51] can be used to check the termination of a set of rewriting rules with a signature. Lambdapi can export its definitions and Signature into a format accepted by the tool AProVe.

Combining Theorem 1 with Lemma 1 we get the following result:

Corollary 1.1. The set of rewrite rules $\mathcal{R}$ over the signature Σ is confluent if and only if its critical pairs are joinable.

A complete proof of subjection reduction for $\longrightarrow_\beta$ and $\longrightarrow_\Sigma$ can be found in [84, §2.6.4].

2.5 PRELUDE

The Lambdapi standard library includes the following definitions, which we will reuse and contribute to enrich.

Definition 2.5.1 (Prelude Encoding)**.** The signature Σ of our encoding contains the following definitions and rewrite rules that we use to encode Alethe proofs:

`Set : TYPE`	`Prop : TYPE`
`El : Set → TYPE`	`Prf : Prop → TYPE`
`⇝ : Set → Set → Set` (infix)	`o : Set`
`El (x ⇝ y) ↪ (El x → El y)`	`El o ↪ Prop`
`= : Π[a : Set], El a → El a → Prop`	(infix)

The notions of term, proposition, and proof are not primitive in $\lambda\Pi/\equiv$. We first define a notion analogous to the predicate logic notion of term, to express the objects the theory speaks about, such as the natural numbers. The constants `Set` and `Prop` in Definition 2.5.1 are type universes "à la Tarski" [67, §Universes] in $\lambda\Pi/\equiv$. The type `Set` will be used to represent the universe of *small types*, i.e. a subclass of types for which we can define equality. Since elements of type `Set` are not types themselves, we also introduce a function `El : Set → TYPE` that embeds the terms of type `Set` into terms of type `TYPE`. Equality (=) over small types is parameterized over types `El a` for the type parameter `a : Set`.

Just as $\lambda\Pi/\equiv$ does not contain a primitive notion for expressing the objects of the theory, it does not contain a primitive notion of proposition. The constant `Prop` represents the universe of propositions. Predicate symbols are represented as constants of type `Set → ⋯ → Prop`. Similar to `Set`, elements of type `Prop` are not types themselves; they are mapped to types by the function `Prf : Prop → TYPE`. Drawing an analogy with the Curry-de-Brujin-Howard isomorphism [92], this function embeds propositions into types by mapping each proposition A to the type `Prf` A, which represents its proofs. However, the analogy is only partial in two important ways. First, the propositions themselves are not the types of their proofs: if t is a proof of A, then it does not have the type A, but the type `Prf` A. Second, the embedding is not surjective—meaning not all types correspond to proofs. For instance, neither `Set` nor `Prop` is itself a type of proof.

Consequently, a *step* in an Alethe proof trace is represented as an inhabitant of type `Prf` p, for a suitable proposition p.

Example 3 (Multiplication group)**.** The encoding of the multiplication group with a multiplication symbol $\times$, an inverse operation inv and a neutral element 1.

$$\begin{array}{lll} \iota : \mathtt{TYPE} & \times : \iota \to \iota \to \iota & \mathrm{inv} : \iota \to \iota \\ 1 : \iota & x \times 1 \hookrightarrow x & x \times (\mathrm{inv}\ x) \hookrightarrow 1 \\ \mathrm{inv}\ (\mathrm{inv}\ x) \hookrightarrow x & 1 \times x \hookrightarrow x & (\mathrm{inv}\ x) \times x \hookrightarrow 1 \\ \mathrm{inv}\ 1 \hookrightarrow 1 & (x \times y) \times z \hookrightarrow x \times (y \times z) & \end{array}$$

```
1  symbol ι : TYPE;
2  symbol × : ι → ι → ι;
3  symbol 1 : ι;
4  symbol inv : ι → ι;
5
6  rule $x × 1 ↪ $x;
7  rule 1 × $x ↪ $x;
8  rule ($x × $y) × $z ↪ $x × ($y × $z);
9  rule $x × (inv $x) ↪ 1;
10 rule (inv $x) × $x ↪ 1;
11 rule inv 1 ↪ 1;
12 rule inv (inv $x) ↪ $x;
```

Listing 3: Concrete syntax of Example 3

The Listing 3 above shows the concrete syntax for the group specification. In Lambdapi's concrete syntax, constants are declared using the **`symbol`** keyword, while variables in rewrite rules are prefixed with `$` to distinguish them from constants during pattern matching.

2.6 CONSTRUCTIVE CONNECTIVES, QUANTIFIERS

In our framework, we adopt the standard set of logical operators and axioms from intuitionistic logic.

Definition 2.6.1 (Constructive connectives, quantifiers)**.**

$$\top : \mathtt{Prop} \tag{1}$$

$$\bot : \mathtt{Prop} \tag{2}$$

$$\wedge : \mathtt{Prop} \to \mathtt{Prop} \to \mathtt{Prop} \quad \text{(written infix)} \tag{3}$$

$$\vee : \mathtt{Prop} \to \mathtt{Prop} \to \mathtt{Prop} \quad \text{(written infix)} \tag{4}$$

$$\Rightarrow : \mathtt{Prop} \to \mathtt{Prop} \to \mathtt{Prop} \quad \text{(written infix)} \tag{5}$$

$$\mathtt{Prf}\,(a \Rightarrow b) \hookrightarrow \mathtt{Prf}\,a \to \mathtt{Prf}\,b \tag{6}$$

$$\neg a \coloneqq a \Rightarrow \bot \tag{7}$$

$$\forall : \Pi[a : \mathtt{Set}], (\mathtt{El}\,a \to \mathtt{Prop}) \to \mathtt{Prop} \tag{8}$$

$$\mathtt{Prf}(\forall\, p) \hookrightarrow \Pi\, x, \mathtt{Prf}(p\ x) \tag{9}$$

$$\exists : \Pi\,[a : \mathtt{Set}], (\mathtt{El}\,a \to \mathtt{Prop}) \to \mathtt{Prop} \tag{10}$$

According to the Curry-de Bruijn-Howard correspondence, a proof of $A \Rightarrow B$ should have the type $\mathtt{Prf}\,(A \Rightarrow B)$. However, such a proof actually has type $\mathtt{Prf}\,A \to \mathtt{Prf}\,B$. This means that the types $\mathtt{Prf}\,(A \Rightarrow B)$ and $\mathtt{Prf}\,A \to \mathtt{Prf}\,B$ must be identified. To achieve this, we introduce the rewriting rule Equation (6), allowing $\mathtt{Prf}\,(A \Rightarrow B)$ to be rewritten as $\mathtt{Prf}\,A \to \mathtt{Prf}\,B$.

We cannot assign the type $\Pi X : \mathtt{TYPE}, (X \to \mathtt{Prop}) \to \mathtt{Prop}$ to the universal quantifier, because in $\lambda\Pi/\equiv$ there is no mechanism for quantifying over variables of type TYPE. Instead, we assign the type $\Pi x : \mathtt{Set}, (\mathtt{El}\, x \to \mathtt{Prop}) \to \mathtt{Prop}$ to the generic universal quantifier $(\forall)$ Equation (8). Furthermore, the previously defined term $o : Set$, together with the rewrite rule $\mathtt{El}\, o \hookrightarrow \mathtt{Prop}$ enables quantification over propositions, rendering the universal quantifier impredicative.

Example 4 (quantification over Prop with o)**.** The eliminator for the bottom $(\bot)$ can be formalized as:

$$\bot_e : \mathtt{Prf}\,(\bot \Rightarrow \forall (p : \mathtt{El}\, o), p)$$

Example 5 (Product type)**.** The product type, which pairs together values from two types, is constructed as follows:

Non dependent product type:

$\times : \mathrm{Set} \to \mathrm{Set} \to \mathrm{Set}$ *(written infix)*

Pairing symbol:

$(,)[a\ b] : \mathtt{El}\, a \to \mathtt{El}\, b \to \mathtt{El}(a \times b)$

Projections (written postfix):

$_1 : \mathtt{El}\,(a \times b) \to \mathtt{El}\, a$ with rewrite rule $(x, _)_1 \hookrightarrow x$

$_2 : \mathtt{El}\,(a \times b) \to \mathtt{El}\, b$ with rewrite rule $(_, y)_2 \hookrightarrow y$

The pairing constructor is denoted by $(,)$ (a comma-like symbol), which takes two terms $x : \mathtt{El}\, a$ and $y : \mathtt{El}\, b$ and produces a term $x, y : \mathtt{El}(a \times b)$. Two projection symbols $__1$ and $__2$ extract the first and second components of a pair, respectively.

2.6.1 *Inductive data structure*

To illustrate how inductive types are expressed in Lambdapi, we present below some data structures. While these examples are is primarily intended to demonstrate the syntax and capabilities of the system, some of these structures will later be employed in our encoding in Chapter 7. Lambdapi extends the $\lambda\Pi/\equiv$-calculus modulo by providing partial support for inductive types. An inductive type is declared by specifying its name, followed by a list of constructors C_i, each describing one possible way of building an element of the type. The general form is:

$$\begin{array}{l}(a_1 : A_1 \ldots a_1 : A_n)\ T : I_1 \to \ldots I_n \to \mathtt{TYPE} \\ \mid C_1 : \cdots : T \ldots \\ \mid C_2 : \cdots : T \ldots \\ \vdots\end{array}$$

The (possibly empty) set $(a_1 : A_1 \ldots a_n : A_n)$ represents the parameters, while the (possibly empty) set $I_1, \ldots, I_n$ represents the

indices. This support is described as partial because Lambdapi currently does not enforce the positivity constraint on inductive definitions [35, §2.2]. From any inductive type declaration, Lambdapi automatically derives the corresponding induction principle along with its associated inference rules. A current limitation exists forcing inductive type to be of kind `TYPE`, so for example it is not possible to create inductive predicates (`Prop`).

Definition 2.6.2 (Natural)**.** The Peano numbers are defined as:

```
ℕ : TYPE
| 0 : ℕ
| +1 : ℕ → ℕ   (written postfix)

nat : Set
El nat ↪ ℕ
```

We make use of Lambdapi's `builtin` mechanism to enable decimal notation for numeric values, allowing us to write, e.g., 2 instead of ((0 +1) +1).

Definition 2.6.3 (Boolean)**.** The booleans are defined as:

```
𝔹 : TYPE
|true : 𝔹
|false : 𝔹

bool : Set
El bool ↪ 𝔹
```

Definition 2.6.4 (List)**.** The inductive type for lists is defined as follows:

```
(a : Set) 𝕃 : TYPE
| □ : 𝕃 a
| :: : El a → 𝕃 a → 𝕃 a   (written infix)

list : Set → Set
El (list a) ↪ 𝕃 a
```

where the constructor □ represents the empty list and :: represents the cons operation that prepends an element to a list. For example, the list containing elements x, y, and z (in that order) is written as x :: y :: z :: □.

Definition 2.6.5 (Dependent pair)**.** The type ∫ parameterized by a predicate p on a type `El` a, represents a dependent pair consisting of an element x : `El` a and a proof that p x holds.

$$\int : \forall a : \texttt{Set}, (\texttt{El}\ a \to \texttt{Prop}) : \texttt{TYPE}$$
$$| \ \texttt{exist} : \Pi(x : \texttt{El}\ a), \texttt{Prf}(p\ x) \to \int\ p$$

$$\texttt{int}\ [a : \texttt{Set}] : (\texttt{El}\ a \to \texttt{Prop}) \to \texttt{Set}$$
$$\texttt{El}\ \texttt{int}\ p \hookrightarrow \int\ p$$

$$s_1\ [a : \texttt{Set}][p : (\texttt{El}\ a \to \texttt{Prop})](e : \int\ a\ p) : \texttt{El}\ a$$
$$(\texttt{exist}\ x\ _)\ s_1 \hookrightarrow x$$
$$s_2\ [a : \texttt{Set}][p : (\texttt{El}\ a \to \texttt{Prop})](e : \int\ a\ p) : \texttt{Prf}(p\ (e\ s_1))$$
$$(\texttt{exist}\ _\ p)\ s_2 \hookrightarrow p$$

The projections from this dependent pair are defined through the symbols s_1 and s_2, corresponding respectively to the first and second components of the pair. The rewrite rule for s_1 extracts the value x from a term of the form `exist` x _, while the rule for s_2 retrieves the proof that $(p\ x)$ holds.

Definition 2.6.6 (Dependent vector)**.** The type `Vec` $a\ n$ of vectors of elements of a of length n. We define the constant `Vec` $a\ n$ inductively for $n : \mathbb{N}$ as:

$$(a : \texttt{Set})\texttt{Vec} : \mathbb{N} \to \texttt{TYPE}$$
$$| \ \square_V : \texttt{Vec}\,a\,0$$
$$| \ ::_V\ [n : \mathbb{N}] : \texttt{El}\ a \to \texttt{Vec}\ a\ n \to \texttt{Vec}\ a\ (n+1) \quad \text{(infix)}$$
$$\texttt{vec} : \texttt{Set} \to \mathbb{N} \to \texttt{Set}$$
$$\texttt{El}\ (\texttt{vec}\ a\ n) \hookrightarrow \texttt{Vec}\ a\ n$$

2.7 LANGUAGE EXTENSIONS

Lambdapi extends the language accepted by the $\lambda\Pi/\equiv$-calculus to make it easier to use. For example, this lets the user to define implicit coercions, programmable tactics, modifiers on symbols that qualify the visibility, reduction, and compilation behaviour of declared symbols.

2.7.1 *Modifiers on symbols*

Lambdapi modifiers are keywords that precede a constant declaration to provide the system with additional information on its properties and behavior. We will present the modifiers **`sequential`** and **`associative commutative`** that will be use in the contributions of this thesis.

2.7.1.1 *Sequential*

The modifier **`sequential`** alters the pattern matching algorithm. By default, the order of rule declarations is not taken into account. This modifier tells Lambdapi to apply rules defining a sequential symbol in the order they have been declared. However, this rule can break important properties such as confluence or *stable by substitution*.

Lemma 2 (Stability by substitution)**.** Suppose we have a typing judgment

$$\Gamma, x : A \vdash t : B$$

and also a derivation

$$\Gamma \vdash u : A,$$

then replacing x by u in t preserves typing:

$$\Gamma \vdash t[x \leftarrow u] : B[x \leftarrow u].$$

Example 6. Consider the constant `eq : Prop → Prop → 𝔹` defined **`sequential`** and the following non-linear set of rewrites rules:

```
eq x x ↪ true                eq x y ↪ false
```

Then it is not stable by substitution given the following substitution $\vdash \texttt{eq}[x \leftarrow y]$ which breaks Lemma 2.

2.7.1.2 *Associative commutative*

Lambdapi handles Associative Commutative (AC) symbols via the modifiers **`associative commutative`**, normalizing terms to a canonical form using a built-in ordering ([24], [22, §5]).

Definition 2.7.1 (builtin ordering relation)**.** Let T be a type with constructors $\epsilon : T$, $C_1 : A \to T$, $C_2 : A \to B \to T$, and $C_n : \mathcal{T}_1 \to \cdots \to \mathcal{T}_n \to T$ for some arbitrary types $A, B, \mathcal{T}_1, \ldots, \mathcal{T}_n$ representing constructors of different finite arities.

The $\preceq$ builtin total order on T-terms is defined as follows. Terms are ordered such that

$$\epsilon < C_1(a) \preceq C_1(a') < C_2(c, b) < C_n(t_1, \ldots, t_n)$$

for any terms $a \preceq a'$ of appropriate types, any $c : A$, and any $b : B$.

- For C_1 constructor terms, $C_1(a_1) \preceq C_1(a_2)$ if and only if $a_1 \preceq a_2$.
- For C_2 constructor terms, $C_2(c_1, b_1) \preceq C_2(c_2, b_2)$ if and only if either $c_1 < c_2$, or $c_1 = c_2$ and $b_1 \preceq b_2$.
- For C_n constructor terms, $C_n(t_1^1, \ldots, t_n^1) \preceq C_n(t_1^2, \ldots, t_n^2)$ if and only if there exists some $k \in \{1, \ldots, n\}$ such that $t_i^1 = t_i^2$ for all $i < k$ and $t_k^1 < t_k^2$, or $t_i^1 = t_i^2$ for all $i \in \{1, \ldots, n\}$.

Now, consider a binary constructor $\oplus : T \to T \to T$ declared **`associative commutative`**. The transformation to canonical form implemented in Lambdapi ensures that $\oplus$ expressions such as:

$$(C_2\ c_1\ x_1) \oplus (C_1\ k_1) \oplus (C_2\ c_2\ x_2) \oplus \cdots \oplus (C_2\ c_n\ x_n)$$

will be normalized following the builtin relation of Definition 2.7.1.

Definition 2.7.2. Let $\twoheadrightarrow^{AC}$ be the relation mapping every term t to its unique AC-canonical form denoted $[t]$.

Definition 2.7.3 (AC equivalence)**.** Two terms t and u are AC-equivalent (written $t \simeq_{AC} u$) if and only if their AC-canonical forms are equal: $[t] = [u]$.

Example 7. Consider the inductive type $\mathbb{M}$ below with 3 constructors of three different arities from 0 to 2, and the natural order on $\mathbb{N}$ terms:

$$\begin{aligned}
&\mathbb{M} : \texttt{TYPE}\\
&\mid \epsilon : \mathbb{M}\\
&\mid P_1 : \mathbb{N} \to \mathbb{M}\\
&\mid P_2 : \mathbb{N} \to \mathbb{N} \to \mathbb{M}\\
&\mid \oplus : \mathbb{M} \to \mathbb{M} \to \mathbb{M} \quad \text{(written infix)}
\end{aligned}$$

We define $\oplus$ as **`associative commutative`**. Therefore, the term

$$P_2(1,2) \oplus P_1(1) \oplus P_2(1,1) \oplus P_1(2) \oplus \epsilon$$

is normalized to:

$$\epsilon \oplus P_1(1) \oplus P_1(2) \oplus P_2(1,1) \oplus P_2(1,2)$$

Remark 4. The properties of *local confluence* and *termination* need to take AC-canonicalization into account to prove that they still hold.

2.7.2 *Metatactics*

In addition to primitive tactics [1], Lambdapi provides *metatactics*[2], a small language of tactic combinators that is interpreted by the tactic `eval`. A metatactic is an ordinary term of type `Tactic : TYPE` that rewrites to an expression over a collection of builtins and is then executed as a primitive tactic on the current proof state. This design lets us program tactic behaviour with rewriting and higher-level composition, which is particularly convenient for robust, automation-friendly scripts in our reconstruction setting.

```
1 symbol Tactic : TYPE;
2 symbol tactic : Set;
3 rule El tactic ↪ Tactic;
4
5 symbol and : Tactic → Tactic → Tactic;
```

1 https://lambdapi.readthedocs.io/en/latest/tactics.html
2 https://lambdapi.readthedocs.io/en/latest/tacticals.html

```
6  builtin "and" ≔ and;
7
8  symbol #apply [p] : Prf p → Tactic;
9  builtin "apply" ≔ #apply;
10
11 symbol #fail : Tactic;
12 builtin "fail" ≔ #fail;
13
14 symbol #orelse : Tactic → Tactic → Tactic;
15 builtin "orelse" ≔ #orelse;
```

Listing 4: Example of builtins metatactics

Concretely, `eval` expects a term that reduces to a composition of builtin tactics like the one from Listing 4 , and the control combinators `#repeat`, `#and`, etc. The combinator `#and` sequences two tactics , `orelse` encodes left-biased choice, `#repeat` iterates a tactic as long as it makes progress.

Example 8 (The `do` metatactic.)**.** The metatactic `do` below provides a simple example of how iterative control structures can be defined directly in Lambdapi without extending the core Lambdapi tactics.

```
nothing ≔ #try #fail

do : ℕ → Tactic → Tactic
do 0 t ↪ nothing
do (n +1) t ↪ t #and (do n t)
```

It acts as a counted loop over tactics: given a natural number and a tactic, it sequentially executes the tactic that many times. Because `do` is defined purely by rewrite rules rather than as a primitive command, it demonstrates the reflective nature of metatactics—proof automation patterns can be expressed and combined within the logical framework itself, using existing tactic combinators such as `#and`.

2.7.3 *Implicit Coercions*

Lambdapi can be instructed to insert function applications into terms whenever needed for typability. Such functions are known as *coercions* , which we denote as:

```
coerce A B x ↪ f x,
```

meaning that a term $x : A$ is coerced by f into a term of type B.

Example 9. Consider the types $\mathbb{B}$ and $\mathbb{N}$ (Definitions 2.6.2 and 2.6.3), and a function:

```
bool2nat : 𝔹 → ℕ
bool2nat true ↪ 1
bool2nat false ↪ 0

coerce 𝔹 ℕ x ↪ bool2nat x
```

which maps `true` to 1 and `false` to 0. By declaring this function as a coercion from booleans to natural numbers, Lambdapi can implicitly apply it whenever a term of type $\mathbb{B}$ is used in a context expecting a $\mathbb{N}$. In other words, the coercion mechanism makes it possible to write mixed expressions such as $x + b$, where $x : \mathbb{N}$ and $b : \mathbb{B}$, and have Lambdapi automatically interpret it as $x +$ (`bool2nat` b).

SUMMARY

This chapter presented the λΠ-calculus modulo rewriting as implemented in Lambdapi, highlighting how user rewrite rules at both the term and type levels extend LF. We presented concrete data structures and the modifiers that will be used throughout the next chapters for our encoding.

3 SATISFIABILITY MODULO THEORIES

3.1 SMT SOLVERS

Satisfiability Modulo Theories (SMT) [15] extend SAT solving [45, 72], combining propositional search with reasoning in background theories. These theories formalize some fundamental data types of computer science, including integers and rationals, floating-point numbers, strings, arrays, sequences, records, and finite sets. By incorporating the essential algebraic properties of such types as built-in reasoning modules, SMT solvers can handle them natively rather than treating them as uninterpreted. SMT solvers are widely used in software and hardware verification [75], constraint solving [8, 100], and are integrated into interactive theorem prover for automated theorem proving [20, 40, 46, 65, 79].

SMT solvers take as input a logical formula and determine whether it is *satisfiable*, that is, whether there exists an assignment to the formula's variables that makes it true. In this sense, they function as constraint solvers. In many applications, however, the goal is not merely to establish that a property, expressed as a formula F, holds in some instance, but rather that it holds universally. Formally, this amounts to proving that F is *valid*. To verify validity with an SMT solver, this can be done by checking if the *negation* of the formula, $\neg\mathsf{F}$, is false in all cases i.e. *unsatisfiable*. A proof of the *unsatisfiability* of $\neg\mathsf{F}$ provides evidence that F is valid. Thus, SMT solvers provide not only a means of solving constraints but also a method for establishing theorems, since valid formulas are commonly referred to as theorems.

Modern *hammer* tools, such as CoqHammer [40] for Rocq [93] or Sledgehammer [20] for Isabelle/HOL [77], and the SMT integration in Lean [71], use of SMT solvers to automate proof search. Their purpose is to bridge interactive theorem provers with powerful automated reasoning backends. Given a conjecture, the hammer translates the goal and relevant lemmas into a form suitable for external solvers, including SMT solvers, and then attempts to discharge the proof obligation automatically. If the SMT solver succeeds in showing that the translated problem is valid, the hammer reconstructs or certifies a corresponding proof inside the interactive prover. In this way, SMT solvers act as critical engines for proof automation, providing fast search capabilities that complement the expressive but more manual environments of interactive theorem proving.

3.2 TERMS AND SORTS

The SMT-LIB initiative [18] defines a standardized input language and logical framework for SMT solvers, specifies a set of common

background theories, and maintains an extensive collection of benchmarks. The SMT-LIB logic is a many-sorted first-order logic with equality.

- The set $\mathcal{S}$ contains the sort symbols denoted by σ. The set contains the symbol **`Bool`** for Booleans.
- The countably infinite set $\mathcal{X}$ contains the variables denoted by x and y.
- The set $\mathcal{F}$ contains the function symbols. This set contains the symbols $\approx$, $\vee$, $\wedge$, and $\neg$.
- The two-element set $\mathcal{B}$ contains the symbols for the **`Bool`**.

Definition 3.2.1 (Terms)**.** The following grammar rule describes the construction of terms t

$$t ::= x \mid f\ t^* \mid f^{\sigma} t^* \mid \exists (x : \sigma)^+ . t \mid \forall (x : \sigma)^+ . t \mid \texttt{let}\ (x = t)^+\ \texttt{in}\ t$$

The notation $()^*$ denotes a possibly empty and finite list, and $()^+$ a finite list with at least one member.

A sort designates the logical type associated with terms, variables, and functions, thereby defining the domain of discourse for a given theory. Sorts provide the formal structure necessary to ensure that formulas are well-formed, preventing ill-formed operations across incompatible domains. Since the SMT-LIB logic is many-sorted, every term has an associated sort σ. To avoid ambiguous sorts, there is a syntax to concretize the sort of a function symbol f^{σ}. Third, the term language also encompasses terms of Boolean sort, so that formulas are represented as Boolean terms, while still being distinguished by their sorts. Moreover, symbols such as equality and the Boolean connectives are not treated as primitive syntax but are instead provided by the special *theory Core*.

Well-sortedness in SMT-LIB is defined with respect to a *signature*, which specifies the available sorts and function symbols from which terms can be constructed. The concrete signature depends on the selected background theories, but it can also be extended by the user with uninterpreted sorts and symbols through dedicated commands. While SMT-LIB in general supports parametric sorts and sort terms, these features are not used in our setting.

A formula without free variables is called a *sentence*. A term without any variables is called a *ground term*. A formula that is a ground term is a *ground formula*.

Notation. Since the symbol Σ is already used for Lambdapi signatures, and SMT-LIB also employs Σ for its own signature, we instead denote SMT signature by Θ.

Definition 3.2.2 (Signature)**.** A signature is a tuple:

$$\Theta = (\Theta^{\mathcal{S}}, \Theta^{\mathcal{F}}, \Theta^{\mathcal{X}}, V, R)$$

- a finite set $\Theta^{\mathcal{S}}$ of sort symbols that contains the symbol `Bool`;
- The countably infinite set $\Theta^{\mathcal{X}}$ contains the variables;
- a set $\Theta^{\mathcal{F}}$ $\Theta^{\mathcal{F}}$ of function symbols;
- a partial mapping V from $\Theta^{\mathcal{X}}$ to $\Theta^{\mathcal{S}}$ that assigns sorts to some variables,
- and a ranking relation $R \subseteq \Theta^{\mathcal{F}} \times (\Theta^{\mathcal{S}})^+$. The relation must be left-total: for all $f \in \Theta^{\mathcal{F}}$ there is a $\eta \in (\Theta^{\mathcal{S}})^+$ such that $f\ R\ \eta$. The set $(\Theta^{\mathcal{S}})^+$ is the set of non-empty finite lists over the sort symbols in $\Theta^{\mathcal{S}}$.

A rank denotes the expected sort of the symbol's arguments and the return sort. The lists $(\Theta^{\mathcal{S}})^+$ are the ranks. From left to right, a rank denotes the expected sort of the symbol's arguments and of the result. Since the SMT-LIB logic allows ad-hoc polymorphism (overloading), R is not left-unique. A function symbol together with a sort is called a ranked function symbol. A ranked function is written as $f : \sigma_1 \dots \sigma_n\sigma \in \Theta^{\mathcal{F}} \times (\Theta^{\mathcal{S}})^+$.

Notation. We write $f : \sigma_1 \dots \sigma_n\sigma \in \Theta$ if $f\ R\ \sigma_1 \dots \sigma_n\sigma_1$ and $x \in \Theta$ if $V(x) = \sigma$.

Definition 3.2.3 (Well-sorted Term)**.** A term t generated by the grammar rules in Definition 3.2.2 is *well-sorted* for a signature Θ if t has sort σ. Let $\Theta[x_1 : \sigma_1, \dots, x_n : \sigma_n]$ denote the signature for which the function V maps x_i to σ_i for $i \in 1, \dots, n$ and otherwise corresponds to V of Θ. The term t has sort σ if a judgment $\Theta \vdash t : \sigma$ is derivable by the following sorting rules:

$$\frac{}{\Theta \vdash x : \sigma} \text{ if } x : \sigma \in \Theta$$

$$\frac{\Theta \vdash t_1 : \sigma_1 \quad \dots \quad \Theta \vdash t_k : \sigma_k}{\Theta \vdash f\, t_1 \cdots t_k : \sigma} \text{ if } \begin{cases} f : \sigma_1 \cdots \sigma_k\sigma \in \Theta, \\ f : \sigma_1 \cdots \sigma_k\sigma' \notin \Theta \\ \quad \text{for all } \sigma' \neq \sigma \end{cases}$$

$$\frac{\Theta \vdash t_1 : \sigma_1 \quad \dots \quad \Theta \vdash t_k : \sigma_k}{\Theta \vdash f t_1 \cdots t_k : \sigma} \text{ (if } f : \sigma_1 \cdots \sigma_k\sigma \in \Theta)$$

$$\frac{\Theta[x_1 : \sigma_1, \dots, x_{k+1} : \sigma_{k+1}] \vdash t : \texttt{Bool}}{\Theta \vdash Q\, x_1 : \sigma_1 \cdots x_{k+1} : \sigma_{k+1}.t : \texttt{Bool}} \text{ (if } Q \in \{\forall, \exists\})$$

$$\frac{\begin{array}{c}\Theta \vdash t_1 : \sigma_1 \dots \Theta \vdash t_{k+1} : \sigma_{k+1} \\ \Theta[x_1 : \sigma_1, \dots, x_{k+1} : \sigma_{k+1}] \vdash t : \sigma\end{array}}{\Theta \vdash (\texttt{let}\ x_1 = t_1 \cdots x_{k+1} = t_{k+1}\ \texttt{in}\ t) : \sigma} \text{ (x_i all distinct)}$$

3.3 SEMANTICS

To define the semantics of the SMT-LIB logic, we use the notion of Θ-structure.

Definition 3.3.1 (Θ-structure)**.** A Θ-*structure* is a tuple $\mathcal{A} = (U, I)$ of a set U, the *universe*, and a mapping I.

- Each sort $\sigma \in \Theta^S$ is mapped to a non-empty set $I(\sigma) \subseteq U$, the *domain* of σ. The mapping is such that $U = \bigcup_{\sigma \in \Theta^S} I(\sigma)$.
- For each sort $\sigma_1 \dots \sigma_n \sigma$, $I(\sigma_1 \dots \sigma_n \sigma)$ is the set of total functions from $I(\sigma_1) \times \cdots \times I(\sigma_n)$ to $I(\sigma)$.
- $I(\mathsf{Bool}) = \{\mathsf{true}, \mathsf{false}\}$, where true and false are distinct and both are elements of U.
- Each ranked function symbol $f : \sigma$ (i.e. constant) is mapped such that $I(f : \sigma) \in I(\sigma)$.
- Each ranked function symbol $f : \sigma_1 \dots \sigma_n \sigma \in \Theta$ is mapped to a total function in $I(\sigma_1 \dots \sigma_n \sigma)$.

A Θ-interpretation $\mathcal{I}$ is a pair $(\mathcal{A}, \nu)$ of a Θ-structure $\mathcal{A}$ and a valuation ν that assigns meaning to variables, given a $\mathcal{A}$ structure and a variable assignment, so that we can decide whether a formula is true or false. This mapping function is denoted $[\![_]\!]^{\mathcal{I}}$.

Definition 3.3.2 (Θ-interpretation)**.** Let Θ be a signature and $\mathcal{I} = (\mathcal{A}, \nu)$ a Θ-interpretation. For every well-sorted term t with sort σ, the mapping $[\![t]\!]^{\mathcal{I}}$ is defined recursively as follows:

$$
\begin{aligned}
&[\![x]\!]^{\mathcal{I}} = \nu(x : \sigma) \\
&[\![\hat{f}\ t_1 \dots t_n]\!]^{\mathcal{I}} = \mathcal{A}(f : \sigma_1 \dots \sigma_n \sigma)(a_1, \dots, a_n)
\end{aligned}
$$

where

$$
\begin{cases}
\mathcal{I} = (\mathcal{A}, \nu) \text{ with signature } \Theta, \\
\hat{f} = f \text{ or } \hat{f} = f^{\sigma}, \\
\Theta \vdash t_i : \sigma_i \text{ and } a_i = [\![t_i]\!]^{\mathcal{I}}, \quad i = 1, \dots, n.
\end{cases}
$$

$$
[\![\text{let } x_1 = t_1 \cdots x_n = t_n \text{ in } t]\!]^{\mathcal{I}} = [\![t]\!]^{\mathcal{I}'}
$$

where

$$
\begin{cases}
\mathcal{I} \text{ has signature } \Theta, \\
\Theta \vdash t_i : \sigma_i \text{ and } a_i = [\![t_i]\!]^{\mathcal{I}}, \quad i = 1, \dots, n, \\
\mathcal{I}' = \mathcal{I}[x_1 : \sigma_1 \mapsto a_1, \dots, x_n : \sigma_n \mapsto a_n].
\end{cases}
$$

$$
[\![\exists x_1 : \sigma_1 \cdots x_n : \sigma_n . t]\!]^{\mathcal{I}} = \text{true iff } [\![t]\!]^{\mathcal{I}'} = \text{true for some } \mathcal{I}'
$$

where

$$
\begin{cases}
\mathcal{I} = ((U, I), \nu), \\
(a_1, \dots, a_n) \in I(\sigma_1) \times \cdots \times I(\sigma_n), \\
\mathcal{I}' = \mathcal{I}[x_1 : \sigma_1 \mapsto a_1, \dots, x_n : \sigma_n \mapsto a_n].
\end{cases}
$$

$$[\![\forall x_1 : \sigma_1 \cdots x_n : \sigma_n . t]\!]^{\mathcal{I}} = \text{true} \text{ iff } [\![t]\!]^{\mathcal{I}'} = \text{true for all } \mathcal{I}'$$

where

$$\begin{cases} \mathcal{I} = ((U, I), \nu), \\ (a_1, \ldots, a_n) \in I(\sigma_1) \times \cdots \times I(\sigma_n), \\ \mathcal{I}' = \mathcal{I}[x_1 : \sigma_1 \mapsto a_1, \ldots, x_n : \sigma_n \mapsto a_n]. \end{cases}$$

The notions of satisfiability and falsifiability are defined in the standard way. A Θ-interpretation satisfies a formula φ if $[\![\varphi]\!]^{\mathcal{A}} = \texttt{true}$. Conversely, it falsifies φ if $[\![\varphi]\!]^{\mathcal{A}} = \texttt{false}$. A formula φ is *satisfiable* if there exists a Θ-interpretation that satisfies it, and *unsatisfiable* if no such interpretation exists.

The notion of a *model* applies only to sentences, i.e., formulas without free variables. In this case, the valuation ν is irrelevant. A model of a sentence φ is a Θ-structure $\mathcal{A}$ such that every (equivalently, at least one) Θ-interpretation $(\mathcal{A}, \nu)$ satisfies φ. As usual, $\models$ denotes semantic entailment. A set of sentences Γ entails a sentence ψ if every model of Γ is also a model of ψ. This is written $\Gamma \models \psi$, and if Γ contains a single sentence φ, it is abbreviated $\varphi \models \psi$.

3.4 THEORIES

As discussed above, well-sortedness in SMT-LIB is defined relative to a signature. The SMT-LIB language further permits the combination of theories, provided their signatures are compatible. Two signatures are compatible if they share the same sort symbols and assign identical sorts to all variables. The combination of two compatible theories is then defined as the class of Θ-structures whose projections onto the signatures of the component theories are models of those theories.

A *logic* in SMT-LIB consists of a specific signature Θ, a theory $\mathcal{T}$, and a restriction on the set of admissible sentences. These restrictions are typically syntactic and serve to exclude ill-formed expressions. For instance, in the logic of linear arithmetic, scalar multiplication by concrete coefficients is admitted, allowing terms of the form `c`, `(* c x)`, or `(* x c)` with `x` a free constant and `c` a numeral or its negation, whereas products of two variables are not.

Following common usage in the SMT community, we will not always maintain a strict distinction between the terms "logic" and "theory".

Definition 3.4.1 (Models)**.** The *models* of a Θ-theory $\mathcal{T}$ are the Θ-structures that belong to $\mathcal{T}$.

Defining theories this way gives us the flexibility to declare theories as either specific models, or as the Θ-structures that satisfies a set of sentences which are called axioms.

Definition 3.4.2 (Satisfiability in a theory)**.** A Θ-sequence is satisfiable in a theory $\mathcal{T}$ if and only if it is satisfied by one of the models of $\mathcal{T}$.

A set Γ of Θ-sequences $\mathcal{T}$-entails a Θ-sequence φ denoted $\Gamma \models_{\mathcal{T}} \varphi$ if and only if every model of $\mathcal{T}$ that satisfies all sentences in Γ satisfies φ as well.

In what follows, we briefly introduce some of the logics that will be used in the subsequent chapters.

3.4.1 *The Core theory*

This theory is used by every SMT-LIB logic. It contains the usual Boolean connectives $\wedge, \vee, \neg, \rightarrow$, `xor`, `distinct`, `ite`. Equality, denoted by $\approx$ (= in SMT-LIB), is also part of this theory. The *Core theory* does not contain a dedicated "if and only if" connective. Instead equality with Boolean sorts is used.

Remark 5. Since SMT-LIB is restricted to ASCII characters, it uses the strings `and, or, not, =>, xor` for boolean connectives.

3.4.2 *Uninterpreted Symbols*

The logic **UF** (for uninterpreted functions) allows the declaration of new function symbols. The prefix **QF** (for quantifier-free) may be applied to any logic to indicate that quantifiers are disallowed. The simplest extension of the core theory is therefore **QF_UF**, denoting quantifier-free formulas over uninterpreted functions. This logic excludes quantifiers and introduces no additional symbols beyond those of the core theory.

3.4.3 *The theory of linear arithmetic*

Linear arithmetic in SMT-LIB is specified by the theories of reals (**LA**) and integers (**LIA**), together with syntactic constraints ensuring that arithmetic expressions remain linear.

The signature of the theory of reals and integers contains two additional sorts: `Int` and `Real` , along with the function symbols listed in Table 1. Most of these functions are overloaded to operate uniformly on both integers and reals. Their interpretation is standard, with the exception of division. Since SMT-LIB does not support partial functions, terms such as $(/\ x\ 0.0)$ are well-defined , although the standard leaves their interpretation unconstrained. Consequently, for each real constant v, the term $(\approx\ v\ (/\ t\ 0.0))$ admits a model. The same consideration applies to integer division (`div`).

3.4.4 *Bitvectors*

Bitvectors in SMT-LIB are defined by the theory **BV**. Its signature introduces the sort `(BV n)` for every fixed width $n \geq 1$, together with the function symbols listed in Table 2. The semantics of these operations follow standard fixed-width, two's complement bitvector arithmetic, including modular addition, subtraction, multiplication, and

Symbol	String	Rank
$-$	`-`	Int Int
		Real Real
$-$	`-`	Int Int Int
		Real Real Real
`+`	`+`	Int Int Int
		Real Real Real
$\times$	`*`	Int Int Int
		Real Real Real
`div`	`div`	Int Int Int
`/`	`/`	Real Real Real
`mod`	`mod`	Int Int Int
`abs`	`abs`	Int Int
$\leq$	`<=`	Int Int Bool
		Real Real Bool
`<`	`<`	Int Int Bool
		Real Real Bool
$\geq$	`>=`	Int Int Bool
		Real Real Bool
`>`	`>`	Int Int Bool
		Real Real Bool
`to_real`	`to_real`	Int Real
`to_int`	`to_int`	Real Int
`is_int`	`is_int`	Real Bool
`divisible` n	`(_ divisible n )`	Int Bool

Table 1: Function symbols of the real and integers theory.

Symbol	String	Rank
~	bvnot	BV n BV n
!	bvneg	BV n BV n
&&	bvand	BV n BV n BV n
||	bvor	BV n BV n BV n
⊕	bvxor	BV n BV n BV n
↑	bvnand	BV n BV n BV n
↓	bvnor	BV n BV n BV n
⊙	bvxnor	BV n BV n BV n
+	bvadd	BV n BV n BV n
−	bvsub	BV n BV n BV n
×	bvmul	BV n BV n BV n
$/_u$	bvudiv	BV n BV n BV n
$/_s$	bvsdiv	BV n BV n BV n
mod_u	bvurem	BV n BV n BV n
mod_s	bvsrem	BV n BV n BV n
mod_s	bvsmod	BV n BV n BV n
≪	bvshl	BV n BV n BV n
$≫_u$	bvlshr	BV n BV n BV n
$≫_s$	bvashr	BV n BV n BV n
∘	concat	BV n BV m BV (n+m)
—	extract i j	BV n BV (i−j+1)
—	repeat k	BV n BV (k · n)
—	zero_extend k	BV n BV (n+k)
—	sign_extend k	BV n BV (n+k)
—	rotate_left k	BV n BV n
—	rotate_right k	BV n BV n
$<_u$	bvult	BV n BV n Bool
$≤_u$	bvule	BV n BV n Bool
$>_u$	bvugt	BV n BV n Bool
$≥_u$	bvuge	BV n BV n Bool
$<_s$	bvslt	BV n BV n Bool
$≤_s$	bvsle	BV n BV n Bool
$>_s$	bvsgt	BV n BV n Bool
$≥_s$	bvsge	BV n BV n Bool
bvcomp	bvcomp	BV n BV n BV 1
int2bv	((_ int2bv n))	Int BV n
bv2int	bv2int	BV n Int

Table 2: SMT-LIB Bit-Vector (BV) theory function symbols (sizes n, m, k, i, j are numerals).

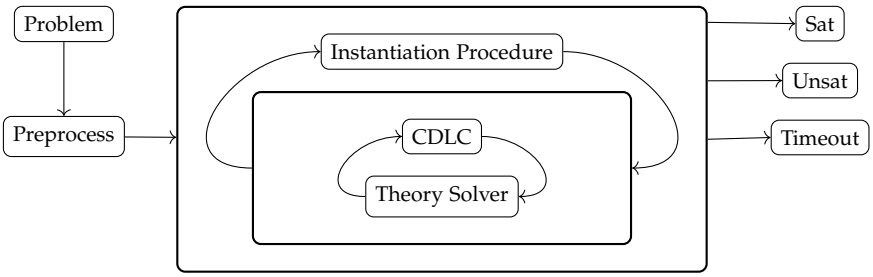


Figure 3: The main loop of an SMT solver.

division, as well as bitwise logical and shift operators. All operations are total, since results are wrapped modulo 2^n.

3.5 SOLVING SMT PROBLEMS

The core idea of SMT solving is to combine propositional reasoning with theory reasoning. The standard method for propositional reasoning is *conflict-driven clause learning* (CDCL) [76]. The Figure 3 illustrates the operation of a CDCL($\mathcal{T}$)-based SMT solver on a problem with quantifiers. The solver first preprocesses the input. Then two procedures cooperate to refute the problem: the ground solver, i.e. the combination of the propositional CDCL engine with the theory solvers restricted to quantifier-free formulas, either refutes it at the ground level or produces a ground model. For quantifier-free problems this is sufficient, whereas in the presence of quantifiers the instantiation procedure is applied to the ground model, either producing new ground lemmas or establishing satisfiability. A *ground model* is an assignment to the variables of a quantifier-free formula that satisfies all ground literals with respect to the background theories. Ground models are obtained from the combination of the CDCL engine with the theory solvers, prior to any reasoning about quantifiers.

Given an input sentence (i.e., a closed term of sort `Bool`), an SMT solver applies several preprocessing steps before entering the solving phase. This produces an equisatisfiable problem F in clause normal form. To optimize the ground solver, ground subformulas are fully clausified. Quantified formulas, however, are treated differently: they are usually neither converted into prenex form nor clausified, and strong quantifiers [7] are typically not fully Skolemized. Preserving the original structure of quantified formulas is crucial for certain instantiation techniques.

Preprocessing also includes lightweight rewriting of formulas. For example, the term $(\bot \to \varphi_1) \to \varphi_2$ is simplified to φ_2. Skolemization is another preprocessing step, but its application is largely implementation dependent. The abstract CDCL($\mathcal{T}$) calculus itself only addresses ground reasoning. In practice, solvers such as cvc5 and veriT, in their default configurations, Skolemize only outermost strong quantifiers.

The inner loop of the SMT solver, shown in Figure 3, attempts to construct a model of the quantifier-free part of the problem. The

CDCL engine first searches for a propositional model of the abstracted problem. The theory solvers then verify whether the literals in this propositional model are consistent with their respective background theories. If they are, the literals form a ground model.

If the SMT solver finds that no propositional model exists (i.e., the abstraction is propositionally unsatisfiable), the original problem is *unsatisfiable* and solving terminates. Otherwise, the propositional model is passed to the theory solvers. Their operation depends on the background theories involved, and their calculi may differ substantially. When multiple theories are present, their solvers must also be combined. In all cases, the theory solver evaluates the candidate model: if it is consistent with the theories, the literals constitute a ground model of the quantifier-free fragment of the sentence; otherwise, contradictions are reported back.

Remark 6. This style of SMT solving is known as the *lazy* approach. By contrast, the *eager* approach encodes the entire input problem directly into propositional logic, ensuring that satisfiability is preserved by the translation. The resulting propositional formula can then be handed to any standard SAT solver.

Algorithm 1 CDCL: algorithmic view

```
1:  procedure SAT(C)
2:      while ⊤ do
3:          if Propagate() then
4:              if ¬Decide() then
5:                  return SAT
6:              end if
7:              continue
8:          end if
9:          if level = 0 then
10:             return UNSAT
11:         end if
12:         Analyse()
13:         Backtrack()
14:     end while
15: end procedure
```

Propagate: find unit clauses repeatedly and push literals on the stack. Returns $\bot$ iff unsatisfied clause

Decide: chooses one non-assigned literal, push on stack. Returns $\bot$ iff no literal

Analyse: analyse the conflict from propagate, create conflict clause, add it in the set of clauses

Backtrack: backtrack (eliminate literals from stack) until conflict clause is unit

The Algorithm 1 outlines the high-level workflow of CDCL, which iteratively:

1. decide a truth value to an unassigned variable,
2. perform *unit propagation*,
3. On conflict, derive a clause to block this and related conflicts, then
4. backjumps non-chronologically using conflict analysis to skip irrelevant decisions.

where *unit propagation* automatically assigns the only remaining literal in a clause to satisfy it (e.g., if $(\neg x \vee y)$ has $x = \texttt{true}$, then y must be true). This repeats until no such literals exist or a conflict is found.

Example 10 (CDCL execution)**.** To illustrate the procedure, consider the trace generated from the clause set below:

$$
\begin{aligned}
C_1 &: \quad x_1 \vee x_2 \\
C_2 &: \quad \neg x_2 \vee \neg x_3 \\
C_3 &: \quad \neg x_2 \vee \neg x_4 \vee \neg x_5 \\
C_4 &: \quad x_3 \vee x_5 \vee x_6 \\
C_5 &: \quad x_7 \vee \neg x_6 \vee \neg x_8 \\
C_6 &: \quad \neg x_4 \vee x_8 \vee x_9 \\
C_7 &: \quad x_{10} \vee \neg x_9 \vee x_{11} \\
C_8 &: \quad \neg x_{11} \vee x_8 \vee \neg x_{12} \\
C_9 &: \quad x_{12} \vee \neg x_{13} \\
C_{10} &: \quad x_7 \vee x_{12} \vee x_{14} \\
C_{11} &: \quad \neg x_6 \vee x_{12} \vee x_{15} \\
C_{12} &: \quad x_{13} \vee \neg x_{14} \vee \neg x_{16} \\
C_{13} &: \quad \neg x_{15} \vee \neg x_{14} \vee x_{16}
\end{aligned}
$$

We have the trace:

- No unit clause: decide $\neg x_1$ — $\neg x_1$
- C_1: propagate x_2 — x_2/C_1
- C_2: propagate $\neg x_3$ — $\neg x_3/C_2$
- No unit clause: decide x_4 — x_4
- C_3: propagate $\neg x_5$ — $\neg x_5/C_3$
- C_4: propagate x_6 — x_6/C_4
- No unit clause: decide $\neg x_7$ — $\neg x_7$
- C_5: propagate $\neg x_8$ — $\neg x_8/C_5$
- C_6: propagate x_9 — x_9/C_6
- No unit clause: decide $\neg x_{10}$ — $\neg x_{10}$
- C_7: propagate x_{11} — x_{11}/C_7
- C_8: propagate $\neg x_{12}$ — $\neg x_{12}/C_8$
- C_9: propagate $\neg x_{13}$ — $\neg x_{13}/C_9$
- C_{10}: propagate x_{14} — x_{14}/C_{10}
- C_{11}: propagate x_{15} — x_{15}/C_{11}
- C_{12}: propagate $\neg x_{16}$ — $\neg x_{16}/C_{12}$
- C_{13}: propagate x_{16} — x_{16}/C_{13}
- Conflict

The solver first decides $\neg x_1$, which by unit propagation yields x_2 from clause C_1 and subsequently $\neg x_3$ from clause C_2. A new decision x_4 is then made; clause C_3 forces $\neg x_5$, and clause C_4 forces x_6. Since no further unit clauses are available, the solver decides $\neg x_7$, which triggers $\neg x_8$ from C_5 and x_9 from C_6. Next, the decision $\neg x_{10}$ initiates a series of propagations: x_{11} from C_7, $\neg x_{12}$ from C_8, $\neg x_{13}$ from C_9, x_{14} from C_{10}, and x_{15} from C_{11}. Finally, clause C_{12} propagates $\neg x_{16}$, while clause C_{13} simultaneously requires x_{16}, producing a direct contradiction. This conflict demonstrates that the assignment trail cannot be extended further, and the explored branch is therefore unsatisfiable.

3.6 SMT PROOF

Since SMT applications crucially rely on correct results, ensuring solver correctness becomes paramount. Ensuring the correctness of SMT solvers is a significant challenge, as these tools have large and highly optimized code bases that evolve rapidly [12]. Rather than attempting to certify the entire solver implementation, a promising approach is to instrument SMT solvers to produce *proofs* of their results [14].

Proof production drastically reduces the size of the trusted code base, since only the proof checker must be trusted, not the entire solver. Beyond this fundamental advantage, proof infrastructure provides multiple benefits: it enforces modularity and clarity in the solver's code, often uncovering bugs in the process; it serves as a

powerful debugging aid, where failed or incorrect proof generation highlights faults in the solver; and it offers a form of test oracle, since an incorrect proof can be detected by an independent checker [55].

When a solver determines satisfiability, it may return a model, while in the unsatisfiable case, a proof of inconsistency provides a verifiable guarantee of correctness. However, producing well-formed proofs is itself non-trivial: errors may arise from faulty proof steps, cycles in proof graphs, or reliance on unsupported assumptions. Efficiency is another key concern, as proof generation must not significantly degrade solver performance.

A common technique is *lazy proof generation*, where the solver constructs a high-level proof sketch during the search, delaying the expansion of detailed lemmas until after unsatisfiability is established. This reduces implementation effort, minimizes performance overhead, and ensures that only relevant proof material is expanded. Lazy proof generation is a key feature of the DRAT [54] proof format used by SAT solvers, and several ways of expanding (also known as elaborating) DRAT proofs have been proposed [38, 57, 62]. This line of work also inspired investigations in the SMT setting [56], where lazy proof generation was developed for CVC4, the predecessor of cvc5 [16]. A modern proof format for SMT, Alethe [87, 90], will be introduced in Chapter 4.

Despite progress, current proof-producing SMT solvers still face shortcomings: some produce proofs that are too shallow, requiring complex external checking; others leave performance-critical components outside of proof production; and many rely on monolithic proof formats, limiting interoperability. Nevertheless, proof production remains the most promising avenue for ensuring SMT solver correctness without certifying their entire, complex code base [91].

4 ALETHE: A NEW STANDARD FOR PROOF TRACE REPRESENTATION

4.1 THE ALETHE PROOF FORMAT

Alethe [87, 90] is a modern proof format for SMT, designed to facilitate integration with proof assistants. It has been co-developed with proof checkers in Isabelle/HOL [88] and currently supports proofs in the theories of uninterpreted functions (**UF**), linear arithmetic (**LA**), and linear integer arithmetic (**LIA**), while support for bit-vectors (**BV**) is under active development. As a result, for SMT fragments covered by Alethe, solver results can be verified inside proof assistants to prove correctness of the solver.

The format is intended as a flexible and uniform representation of *unsatisfiability* proofs produced by SMT solvers as described in Section 3.6. Its design combines a natural-deduction style structure with a set of rules operating on first-order clauses. The language of Alethe is based on SMT-LIB [18], and therefore inherits its many-sorted first-order logic as foundation. With the exception of clauses for propositional reasoning, there is no dedicated syntax for any theory.

Beyond serving as an interchange format, Alethe aims to keep proofs accessible to both humans and automated tools. Proofs are intended to be readable by humans, easily checked, and readily consumable by other systems such as interactive theorem provers or stand-alone proof checkers [3]. The format is already adopted in practice, being supported by the SMT solvers veriT [28] and cvc5 [12], and ongoing development for the Constrained Horn Clauses solver Golem [25].

4.2 THE ALETHE LANGUAGE

The Alethe proof trace format [11] for SMT solvers comprises two parts: the trace language based on SMT-LIB and a collection of proof rules. Traces witness proofs of unsatisfiability of a set of constraints. They are sequences $a_1 \dots a_m \; t_1 \dots t_n$ where the a_i corresponds to the constraints of the original SMT problem being refuted, each t_i is a clause inferred from previous elements of the sequence, and t_n is $\bot$ (the empty clause). In the following, we designate the SMT-LIB problem as the *input problem*.

```
1 (set-logic UF)
2 (declare-sort U 0)
3 (declare-fun a () U)
4 (declare-fun b () U)
5 (declare-fun p (U) Bool)
6 (assert (p a))
7 (assert (= a b))
8 (assert (not (p b)))
```

```
(check-sat)
(get-proof)
```

↯

```
(assume a0 (p a))
(assume a1 (= a b))
(assume a2 (not (p b)))
(step t1 (cl (not (= (p a) (p b))) (not (p a)) (p b)) :rule equiv_pos2)
(step t2 (cl (= (p a) (p b))) :rule cong :premises (a1))
(step t3 (cl (p b)) :rule resolution :premises (t1 t2 a0))
(step t4 (cl) :rule resolution :premises (a2 t3))
```

Listing 5: Running example: an SMT problem and its Alethe proof found by cvc5.

We will use the input problem shown in the top part of Listing 5 with its Alethe proof (found by cvc5) in the bottom part as a running example (for **UF** logic) to provide an overview of Alethe concepts and to illustrate our embedding in Lambdapi. The *input problem* is given as a script consisting of commands that interact with the SMT solver. For example, in Listing 5, **`assert`** introduces an assertion, **`check-sat`** triggers the solving process, and **`get-proof`** requests the solver to output a proof. In the following, we introduce the concepts needed to understand the proof produced by cvc5 in Listing 5.

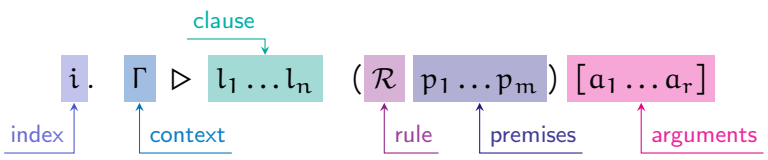
4.2.1 *Many-Sorted First-Order Logic*

An Alethe proof trace inherits the declarations of its input problem. All symbols (sorts, functions, assertions, etc.) declared or defined in the input problem remain declared or defined, respectively. Furthermore, the syntax for terms, sorts, and annotations uses the syntactic rules (Definition 3.2.1) and the signature context (Definition 3.2.2) defined in Chapter 3. The set of sorts present in a proof depends on the chosen SMT-LIB logic or theory, in addition to any sorts introduced by the user. However, the sort **`Bool`** is always included.

Alethe extends SMT-LIB with Hilbert's `choice` operator. A term of the form `(choice x (P x))` denotes a witness v such that $P(v)$ holds if such a witness exists; if not, the term denotes an arbitrary value. We will use the notation $\epsilon x, P\ x$ to denotes `(choice x (P x))`. To ensure coherence, Alethe enforces functionality of the choice operator with respect to logical equivalence: for any formulas P and Q with free variable x, if $\forall x.\, P \simeq Q$, then it must also hold that $\epsilon\, x.\, P \simeq \epsilon\, x.\, Q$.

4.2.2 *Steps*

An Alethe proof is represented as an indexed sequence of steps. In order to reflect the concrete syntax of Alethe proofs, we denote proof steps in the abstract notation by the following form:



(1)

A step Equation (1) consists of an index $i \in \mathbb{I}$ where $\mathbb{I}$ is a countable infinite set of indices (e.g. `a0`, `t1`), and a clause of formulae $l_1, \ldots, l_n$ representing an n-ary disjunction. Steps that are not assumptions are justified by a proof rule $\mathcal{R}$ that depends on a possibly empty set of premises $\{p_1 \ldots p_m\} \subseteq \mathbb{I}$ that only references earlier steps such that the proof forms a directed acyclic graph. A rule might also depend on a list of arguments $[a_1 \ldots a_r]$ where each argument a_i is either a term or a pair (x_i, t_i) where x_i is a variable and t_i is a term. The interpretation of the arguments is rule specific. The context Γ of a step is a list $c_1 \ldots c_l$ where each element c_j is either a variable or a variable-term tuple denoted $x_j \mapsto t_j$. Therefore, steps with a non-empty context contain variables x_j that appear in l_i and will be substituted by t_j. Proof rules $\mathcal{R}$ include theory lemmas and `resolution`, which corresponds to hyper-resolution on ground first-order clauses.

We now have the key components to explain the guiding proof in the bottom part of Listing 5 that consists of seven steps. The proof starts with three `assume` steps `a0`, `a1`, `a2` that restate the assertions from the input problem. In the concrete syntax, assume steps have a dedicated command **`assume`** to clearly distinguish them from normal steps that use the **`step`** command. Step `t1` introduces a tautology of the form $\neg(\varphi_1 \approx \varphi_2) \vee \neg\varphi_1 \vee \varphi_2$, justified by the rule `equiv_pos2`. Steps `t2`, `t3`, `t4` use earlier premises that correspond to previous steps. Step `t2` proves $p(a) \approx p(b)$ by congruence (rule `cong`) from the assumption `a1`. Step `t3` applies the `resolution` rule of propositional logic to the premises `t1, t2, a0` to derive $p(b)$. Finally, the step `t4` concludes the proof by generating the empty clause $\bot$, concretely denoted as `(cl)` in Listing 5. Notice that the contexts Γ of all steps are empty in this proof.

4.2.3 *Subproofs and Lemmas*

Alethe uses subproofs to prove lemmas and to create and manipulate the context. To prove lemmas, a subproof can introduce local assumptions. The `subproof` use at step `2` in Example 11 discharges the local assumptions. As the example shows, the abstract notation denotes subproofs by a frame around the steps in the subproof. A subproof step cannot use a premise from a subproof nested within the current subproof. Subproofs are also used to manipulate the context Γ.

Example 11. This example shows a refutation of the formula $(2+2) \simeq 5$. The proof uses a subproof to prove the lemma $((2+2) \simeq 5) \Rightarrow 4 \simeq 5$.

Step 2 can be viewed as a lemma, and the steps 2.0 – 2.2 are the intermediate steps to prove it.

1.	▷	$(2+2) \approx 5$		assume
2.0	▷	$(2+2) \approx 5$	assume	
2.1	▷	$(2+2) \approx 4$	sum_simplify	
2.2	▷	$4 \approx 5$	(trans 2, 3)	
2.	▷	$\neg((2+2) \approx 5), 4 \approx 5$		subproof
3.	▷	$(4 \approx 5) \approx \bot$		eq_simplify
4.	▷	$\neg((4 \approx 5) \approx \bot), \neg(4 \approx 5), \bot$		equiv_pos2
5.	▷	$\bot$		(resolution 1, 2, 3, 4)

4.2.3.1 *Contexts*

Alethe contexts are a general mechanism to write substitutions and to change them by attaching new elements. We recall that a context is a (possibly empty) list $c_1, \ldots, c_n$, where each element c_i is either:

1. a variable x_i, called a *fixing* of x_i, or
2. a pair $x_i \mapsto t_i$, representing a substitution of x_i by the term t_i.

Every non-empty Γ induces a substitution $\mathtt{subst}(\Gamma)$. The induced substitution is always *capture-avoiding*, i.e., it avoids binding free variables. We use the notation $\bar{c}$ for $c_1, \ldots, c_{n-1}$.

If Γ ends with a fixing, the variable shadows any earlier substitution for it:

$$\mathtt{subst}([\bar{c}, x_n]) \text{ is } \mathtt{subst}([\bar{c}]), \text{ but } x_n \text{ maps to } x_n.$$

If Γ ends with a mapping, the substitution is extended accordingly:

$$\mathtt{subst}([\bar{c}, x_n \mapsto t_n]) = \mathtt{subst}([\bar{c}]) \circ \{x_n \mapsto t_n\},$$

where $\circ$ denotes substitution composition. The application of a context Γ to a term t is written $\mathtt{subst}(\Gamma)(t)$.

Example 12.

$$\mathtt{subst}([x \mapsto 1, x \mapsto (g\ x)])(x) = g(1)$$
$$\mathtt{subst}([x \mapsto 1, x, x \mapsto (g\ x)])(x) = g(x)$$

Contexts are used to express proofs about the preprocessing of terms. The conclusions of proof rules that use contexts always have the form

i. $\Gamma \quad \triangleright \quad t \approx u \quad \mathtt{rule}...$

where the term t is the original term, and u is the term after preprocessing. Tautologies with contexts correspond to judgments $\vDash_{\mathcal{T}} \mathtt{subst}(\Gamma)(t) \approx u$. Formally, the context is translated to λ-abstractions and applications [11, §3.1].

Example 13. Consider a subproof with the context $\Gamma = [y, x \mapsto y]$, used to rename a bound variable:

1.	$\triangleright$	$\forall x, (P\ x)$	`assume`
2.	$\triangleright$	$\neg(\forall y, (P\ y))$	`assume`
3.0	$y, x \mapsto y \triangleright$	$x \simeq y$	`refl`
3.1	$y, x \mapsto y \triangleright$	$(P\ x) \simeq (P\ y)$	`cong 3.0`
3.	$\triangleright$	$\forall x, (P\ x) \simeq \forall y, (P\ y)$	`bind`
4.	$\triangleright$	$\neg(\forall x, (P\ x) \simeq \forall y, (P\ y)),$ $\neg(\forall x, P\ x), (\forall y, P\ y)$	`equiv_pos2`
5.	$\triangleright$	$\bot$	`(resolution 1,2,3,4)`

Remark 7. This kind of subproof in Example 13 is needed in Alethe to justify variable renaming under binders. It is not relevant in Lambdapi, since Lambdapi relies on α-equivalence when comparing terms with binders, and therefore treats $\Pi x.\, P(x)$ and $\Pi y.\, P(y)$ as definitionally equal without requiring an explicit proof step.

4.3 THE CONCRETE SYNTAX OF ALETHE

We recall that the concrete ASCII representation of the Alethe proofs is based on the SMT-LIB standard. Figure 4 shows an example proof as printed by cvc5. The proof is truncated with for readability. The format follows the SMT-LIB standard when possible. Alethe mirrors this structure. Therefore, beside the SMT-LIB logic and term language, it also uses commands to structure the proof. An Alethe proof is a list of commands.

The symbolic names introduced by the `:named` annotation also stay valid and can be used in the script. For the purpose of the proof rules, terms are treated as if proxy names introduced by :named annotations have been expanded and annotations have been removed. For example, the term `or (! (P a) :named foo) foo` and `or (P a) (P a)` are considered to be syntactically equal.

The Figure 5 shows the grammar of the proof text. It is based on the SMT-LIB grammar, as defined in the SMT-LIB standard [18, Appendix B] The non-terminals ⟨`attribute`⟩, ⟨`function_def`⟩ ⟨`sorted_var`⟩, ⟨`symbol`⟩ and ⟨`term`⟩ are as defined in the standard. The non-terminal ⟨`proof_term`⟩ corresponds to the ⟨`term`⟩ non-terminal of SMT-LIB, but is extended with the additional production for the `choice` (i.e. ϵ) binder.

Alethe proofs are a list of commands. Both commands assume and step require an index as the first argument to later refer back to it. This index is an arbitrary SMT-LIB symbol. It must be unique for each assume and step command. To simplify proof checking, the `assume` command must use the name assigned to the asserted formula if there is any. For example, if the input problem contains `(assert (! (not A) :named Goal))`, then the proof must refer to this assertion as `(assume Goal (not a))`.

```
(assume a0 (not (! (p a) :named @p_1)))
(assume a1 (forall ((x1 U)) (or (not (= x1 a)) (p x1))))
(assume a2 (forall ((x1 U) (x2 U) (x3 U))
  (or (not (= x2 (f x3))) (or (not (= x3 a)) (q x1 x2 x3)))))
(step t0 (cl
  (not (! (= (forall ((x1 U)) (or (not (= x1 a)) (p x1))) @p_1) :named @p_2))
  (not (forall ((x1 U)) (or (not (= x1 a)) (p x1)))) @p_1)
  :rule equiv_pos2)
(anchor :step t1 :args ((x1 U) (:= (x1 U) x1)))
(step t1.t0 (cl (= (= x1 a) (= a x1))) :rule rare_rewrite :args ("eq-symm" x1 a))
(step t1.t1 (cl (= (not (= x1 a)) (not (= a x1)))) :rule cong :premises (t1.t0))
(step t1.t2 (cl (= (! (p x1) :named @p_5) @p_5)) :rule refl)
(step t1.t3 (cl (=
  (or (not (= x1 a)) @p_5)
  (or (not (= a x1)) @p_5)))
  :rule cong :premises (t1.t1 t1.t2))
(step t1 (cl (=
  (forall ((x1 U)) (or (not (= x1 a)) (p x1)))
  (forall ((x1 U)) (or (not (= a x1)) (p x1))))) :rule bind)
(step t2 (cl (=
  (forall ((x1 U)) (or (not (= a x1)) (p x1)))
  (! (or (! (not (= a a)) :named @p_3) @p_1) :named @p_4)))
  :rule hole)
(step t3 (cl (= (= a a) true)) :rule rare_rewrite :args ("eq-refl" a))
...
(step t14 (cl) :rule resolution :premises (t13 a0) :args (@p_1 true))
```

Figure 4: Example proof output.

$$
\begin{aligned}
\langle \texttt{proof} \rangle &= \langle \texttt{proof_command} \rangle^{*} \\
\langle \texttt{proof_command} \rangle &= \texttt{(assume } \langle \texttt{symbol} \rangle \langle \texttt{proof_term} \rangle \texttt{)} \\
&\mid \texttt{(step } \langle \texttt{symbol} \rangle \langle \texttt{clause} \rangle \texttt{ :rule } \langle \texttt{symbol} \rangle \\
&\quad \langle \texttt{premises_annotation} \rangle^{?} \\
&\quad \langle \texttt{context_annotation} \rangle^{?} \langle \texttt{attribute} \rangle^{*} \texttt{)} \\
&\mid \texttt{(anchor :step } \langle \texttt{symbol} \rangle \\
&\quad \langle \texttt{args_annotation} \rangle^{?} \langle \texttt{attribute} \rangle^{*} \texttt{)} \\
&\mid \texttt{(define-fun } \langle \texttt{function_def} \rangle \texttt{)} \\
\langle \texttt{clause} \rangle &= \texttt{(cl } \langle \texttt{proof_term} \rangle^{*} \texttt{)} \\
\langle \texttt{proof_term} \rangle &= \langle \texttt{term} \rangle \text{ extended with} \\
&\quad \texttt{(choice (} \langle \texttt{sorted_var} \rangle \texttt{) } \langle \texttt{proof_term} \rangle \texttt{)} \\
\langle \texttt{premises_annotation} \rangle &= \texttt{:premises (} \langle \texttt{symbol} \rangle^{+} \texttt{)} \\
\langle \texttt{args_annotation} \rangle &= \texttt{:args (} \langle \texttt{step_arg} \rangle^{+} \texttt{)} \\
\langle \texttt{step_arg} \rangle &= \langle \texttt{symbol} \rangle \mid \texttt{(} \langle \texttt{symbol} \rangle \langle \texttt{proof_term} \rangle \texttt{)} \\
\langle \texttt{context_annotation} \rangle &= \texttt{:args (} \langle \texttt{context_assignment} \rangle^{+} \texttt{)} \\
\langle \texttt{context_assignment} \rangle &= \texttt{(} \langle \texttt{sorted_var} \rangle \texttt{)} \\
&\mid \texttt{(:=} \langle \texttt{symbol} \rangle \langle \texttt{proof_term} \rangle \texttt{)}
\end{aligned}
$$

Figure 5: The proof grammar.

The second argument in `step` is always a clause. To express disjunctions in SMT-LIB the `or` operator is used. This operator, however, needs at least two arguments and cannot represent unary or empty clauses. To circumvent this, Alethe introduces a new `cl` operator. It corresponds to the standard or function extended to one argument, where it is equal to the identity, and zero arguments, where it is equal to `false`.

SUBPROOFS The abstract notation in Section 4.2.3 marks subproofs with a vertical line. To map this notation to a list of commands, Alethe uses the `anchor` command. This command indicates the start of a subproof. A subproof is concluded by a matching `step` command. This step must use a *concluding rule* (such as `subproof`, `bind`, and so forth). The `:step` annotation of the `anchor` command is used to indicate the step command that will end the subproof. The clause of this step command is the conclusion of the subproof. If the subproof uses a context, the `:args` annotation of the anchor command indicates the arguments added to the context for this subproof. The annotation provides the sort of fixed variables.

In the Figure 4, a subproof starts at the `anchor` command at line 6. It ends with the `bind` steps that finish the proof of proving equality between the two term `(forall ((x1 U)) (or (not (= x1 a)) (p x1)))` and `(forall ((x1 U)) (or (not (= a x1)) (p x1)))`. The context for the subproof `t1` contains a *fixing* argument `((x1 U) (:= (x1 U) x1))`.

SHARING AND SKOLEM TERMS Usually, SMT solvers store terms internally in an efficient manner. A term data structure with perfect sharing ensures that every term is stored in memory precisely once. When printing the proof, this compact storage is unfolded. This leads to a blowup of the proof size. Alethe can optionally use sharing to print common subterms only once. This is realized using the standard naming mechanism of SMT-LIB. A term `t` is annotated with a name `n` by writing `(! t :named n )` where `n` is a symbol. After a term is annotated with a name, the name can be used in place of the term. This is a purely syntactical replacement. Alethe continues to use the names already used in the input problem. Hence, terms that already have a name in the input problem can be replaced by that name and new names introduced in the proof must not use names already used in the input problem.

To simplify reconstruction, Alethe can optionally define Skolem constants as functions. In this case, the proof contains a list of `define-fun` commands that define shorthand 0-ary functions for the `(choice ...)` terms needed. Without this option, no `define-fun` commands are issued, and the constants are expanded.

4.4 OVERVIEW OF ALETHE RULES

Together with the language, the Alethe format also includes a set of proof rules. The Alethe proof format [11, §5] gives the complete list of all proof rules. Historically, the proof rules correspond to the rules that the solver veriT can emit. Alethe provide different types of proof rules that we will describe:

TAUTOLOGOUS RULES AND SIMPLE DEDUCTION. These rules introduce tautologies. One example is the `equiv_pos2` rule used in Listing 5 that introduces the tautology clause:

$$\neg(\varphi_1 \approx \varphi_2), \neg\varphi_1, \varphi_2 \text{ equivalent to } \varphi_1 \approx \varphi_2 \Rightarrow \varphi_1 \Rightarrow \varphi_2$$

Other rules derive their conclusion from a single premise such as `cong` (congruence). Those rules are primarily used to simplify Boolean connectives during preprocessing. For example, the `implies` rule eliminates an implication: From $\varphi_1 \to \varphi_2$, it deduces $\neg\varphi_1, \varphi_2$.

SIMPLIFICATION RULES. These rules represent typical operator level simplifications. For example, the rule `or_simplify` below simplifies `or` expression:

$$\text{i.} \quad \Gamma \rhd \quad l_1 \vee \cdots \vee l_n \approx \psi \quad \texttt{or_simplify} \tag{11}$$

where ψ is the transformed term. The possible transformations are:

(1) $\bot \vee \cdots \vee \bot \Rightarrow \bot$

(2) $l_1 \vee \cdots \vee l_n \Rightarrow l'_1 \vee \cdots \vee l'_m$ where the right-hand side has some $\bot$ literals removed.

(3) $l_1 \vee \cdots \vee l_n \Rightarrow l'_1 \vee \cdots \vee l'_m$ where the right-hand side has some repeated literals removed.

(4) $l_1 \vee \cdots \vee \top \vee \cdots \vee l_n \Rightarrow \top$

(5) $l_1 \vee \cdots \vee l_i \vee \cdots \vee l_j \vee \cdots \vee l_n \Rightarrow \top$ where $l_i = \neg^{2p} x$, $l_j = \neg^{2q+1} x$.

RESOLUTION. CDCL($\mathcal{T}$)-based SMT solvers, and especially their SAT solvers, are fundamentally based on resolution of clauses. Hence, Alethe also has dedicated resolution proof rule:

$$\begin{array}{lll} i_1.\ \rhd & l_1^1, \ldots, l_{k^1}^1 & (\ldots) \\ i_n.\ \rhd & l_1^n, \ldots, l_{k^n}^n & (\ldots) \\ & \vdots & \\ j.\ \rhd & l_{s_1}^{r_1}, \ldots, l_{s_m}^{r_m} \quad (\texttt{resolution}\ i_1 \ldots i_n)[] & \end{array}$$

where $l_{s_1}^{r_1}, \ldots, l_{s_m}^{r_m}$ are from l_j^i and are the result of a chain of binary predicate resolution steps on the clauses of i_1 to i_n. A binary predicate resolution can be formally defined as:

$$\frac{l_1^1, \ldots, l_n^1, p \qquad l_1^2, \ldots, l_n^2, \neg p}{l_1^1, \ldots, l_n^1, l_1^2, \ldots l_n^2} \text{ (Resolution)}$$

It is possible that $m = 0$, i.e. that the result is the empty clause. When performing resolution steps, the rule implicitly merges repeated negations at the start of the formulas l_j^i. For example, the formulas $\neg\neg\neg P$ and $\neg P$ can serve as pivots during resolution. The first formula is interpreted as $\neg P$ and the second as just P for the purpose of performing resolution steps.

SMT solvers do not enforce a strict clausal normal form, ordinary disjunction is also used. Therefore Alethe keep clauses and disjunctions distinct to simplifies rule checking. For example, in the case of resolution there is a clear distinction between unit clauses where the sole literal starts with a disjunction and non-unit clauses. The syntax for clauses uses the `cl` operator, while disjunctions use the standard **`or`** operator. The `or` *rule* is responsible for converting disjunctions into clauses.

Alethe proofs use a generalized propositional resolution rule with the name `resolution` or `th_resolution`. Both names denote the same rule. The difference only serves to distinguish whether the rule was introduced by the SAT solver or by a theory solver. The resolution step is purely propositional; there is no notion of a unifier. The resolution rules also implicitly simplifies repeated negations at the head of literals.

QUANTIFIER INSTANTIATION. To express quantifier instantiation, the rule `forall_inst` is used. It produces a formula of the form $(\neg \forall \bar{x}_n.\, \varphi) \vee \varphi[\bar{t}_n]$, where φ is a term containing the free variables $\bar{x}_n$, and for each i the ground term t_i is a new term with the same sort as x_i.

$$i.\ \Gamma \quad \rhd (\neg \forall \bar{x},\, \varphi) \vee \varphi[\bar{t}] \quad (\texttt{forall_inst})$$

The argument of a `forall_inst` step, for a quantification of the form $\forall x_1, \ldots, x_n.\varphi$, is the list $t_1, \ldots, t_n$, where t_i denotes the substitute of variable x_i. While this information can be recovered from the term, providing it explicitly helps reconstruction because the implicit reordering of equalities obscures which terms have been used as instances.

Skolemization is handled by two rules. The rule `sko_ex` is used to express the Skolemization of an existentially quantified variable. The conclusion of a step that applies this rule is an equality. On the left-hand side of the equality stands a formula whose outermost symbol is an existential quantifier over some variable x. On the right-hand side, the variable x is replaced by the corresponding Skolem term. To justify this replacement, the `sko_ex` rule requires one premise. The premise is evaluated under a context that maps the existentially quantified variable to the Skolem term that witnesses its existence:

$$\begin{array}{lll} i.\Gamma, x_1 \mapsto \epsilon_1, \ldots, x_n \mapsto \epsilon_n & \rhd \varphi \approx \psi & (\ldots) \\ j.\Gamma & \rhd \exists x_1 \ldots x_n, \varphi \approx \psi & (\texttt{sko_ex}) \end{array}$$

where ϵ_i stands for $\epsilon\, x_i, (\exists x_{i+1}, \ldots, x_n, \varphi)$.

Analogously, the rule `sko_forall` is used to express the Skolemization for universal quantification. The structure of the rule is the same as that of `sko_ex`: the conclusion is again an equality, with the left-hand side being a formula starting with a universal quantifier over some variable x. In the formula on the right-hand side, the variable x is replaced by the appropriate Skolem choice term, which is typically constructed from the negation of the body of the universal quantification. As in the case of `sko_ex`, the `sko_forall` rule employs a single premise. The premise is evaluated under a context that maps the universally quantified variable to its associated Skolem choice term:

$$\frac{i.\Gamma, x_i \mapsto (\neg\epsilon\, x_i, \neg\phi) \quad \triangleright \varphi \approx \psi \qquad (\ldots)}{j.\Gamma \qquad \triangleright \forall x_1 \ldots x_n, \varphi \approx \psi \qquad (\texttt{sko_forall})} \quad \text{with } i \in 1 \ldots n$$

Example 14. Here the term $\neg p(\epsilon\, x, \neg p(x))$ is skolemized. The `refl` rule expresses a simple tautology on the equality (reflexivity in this case), cong is functional congruence, and `sko_forall` works like `sko_ex`, except that the choice term is $\epsilon\, x, \neg\varphi$.

$$\begin{array}{lr}
1.x \mapsto (\epsilon x,\ \neg(p\ x))\ \triangleright\ x \approx \epsilon x.\, \neg(p\ x) & \texttt{refl} \\
2.x \mapsto (\epsilon x,\ \neg(p\ x))\ \triangleright\ (p\ x) \approx p(\epsilon x.\, \neg(p\ x)) & \texttt{cong}\ 1 \\
3.\ \triangleright\ (\forall x,\ (p\ x)) \approx p(\epsilon x.\, \neg(p\ x)) & \texttt{sko_forall}\ 2 \\
4.\ \triangleright\ \neg(\forall x,\ (p\ x)) \approx \neg(p(\epsilon x.\, \neg(p\ x))) & \texttt{cong}\ 3
\end{array}$$

LINEAR ARITHMETIC. Proofs in linear arithmetic rely on a set of basic rules, summarized in Table 3. For example, the rule `la_totality` asserts a totality of the ordering $\leq$. and the main rules `la_generic` and `lia_generic`, which describe an algorithm. A step of the rule `la_generic` represents a tautological clause of linear inequalities, and can be verified by demonstrating that the conjunction of the negated inequalities is unsatisfiable. While the `la_generic rule` is primarily intended for **LRA** logic, it is also applied in **LIA** proofs when all variables in the (in)equalities are of integer sort. The Listing 7 shows an example of a linear arithmetic proof in Alethe.

```
(set-logic LIA)
(declare-const x Int)
(declare-const y Int)
(assert (= x 2))
(assert (= 0 y))
(assert (or (< (+ x y) 1) (< 3 x)))
(check-sat)
(get-proof)
```

Listing 6: Example of a Linear Arithmetic Input Problem

```
(assume a0 (or (< (+ x y) 1) (< 3 x)))
(assume a2 (= 0 y))
(assume a1 (= x 2))
(step t1 (cl (< (+ x y) 1) (< 3 x)) :rule or :premises (a0))
(step t2 (cl (not (< 3 x)) (not (= x 2))) :rule la_generic :args (1/1 -1/1))
(step t3 (cl (not (< 3 x))) :rule resolution :premises (a1 t2))
(step t4 (cl (< (+ x y) 1)) :rule resolution :premises (t1 t3))
(step t5 (cl (not (< (+ x y) 1)) (not (= x 2)) (not (= 0 y))) :rule la_generic
:args (1/1 1/1 -1/1))
```

Rule	Description
la_generic	Tautologous linear inequalities
lia_generic	Tautologous linear integer inequalities
la_disequality	$t_1 \approx t_2 \vee \neg(t_1 \geq t_2) \vee \neg(t_2 \geq t_1)$
la_totality	$t_1 \geq t_2 \vee t_2 \geq t_1$
la_rw_eq	$(t \approx u) \approx (t \geq u \wedge u \geq t)$
comp_simplify	Simplification of arithmetic comparisons
arith-int-eq-elim	$(t \approx s) \rightarrow t \geq s \wedge t \leq s$
arith-refl-geq	$t \geq t \rightarrow \top$
arith-refl-lt	$t < t \rightarrow \bot$
arith-refl-leq	$t \leq t \rightarrow \top$
arith-elim-leq	$t \leq s \rightarrow s \geq t$
arith-elim-gt	$t > s \rightarrow \neg(t \leq s)$
arith-leq-norm	$t \leq s \rightarrow \neg(t \geq s + 1)$
arith-geq-norm1	$t \geq s \rightarrow (t - s) \geq 0$
arith-geq-norm2	$t \geq s \rightarrow -t \leq -s$
arith-geq-tighten	$\neg(t \geq s) \rightarrow s \geq t + 1$ with t and s of sort **Int**
arith-poly-norm	polynomial normalization
evaluate	evaluate constant terms

Table 3: Examples of linear arithmetic rules in Alethe.

```
9 (step t6 (cl) :rule resolution :premises (t5 t4 a1 a2))
```

Listing 7: Alethe Proof of Listing 6 found by cvc5

The proof begins with `assume` steps `a0`, `a1`, `a2` that restate the assertions from the *input problem*. Step `t1` transforms the disjunction `a0` into a clause by using the Alethe rule `or`. Steps `t2` and `t5` are tautologies introduced by the main rule `la_generic` in Linear Real Arithmetic (LRA) logic and also used in LIA logic, where $l_1, l_2, \ldots, l_n$ represent linear inequalities. The **Real** terms in LRA and LIA logic are constructed over the **Real** and **Int** signatures from SMT-LIB with free variables, but containing only linear atoms; that is atoms of the form `d`, `(* d x)`, or `(* x d)` where `x` is a free variable and `d` is an integer or rational constant. A linear inequality is an expression of the form:

$$\sum_{i=0}^{n} c_i \times t_i + d_1 \bowtie \sum_{i=n+1}^{m} c_i \times t_i + d_2 \tag{12}$$

where $\bowtie \in \{=, <, >, \leq, \geq\}$, $m \geq n$, c_i, d_1, d_2 are constants either **Int** or **Real**, and where c_i and t_i have the same sort for all i. Checking the clause validity of `t2` and `t5` in Listing 7, amounts to checking the unsatisfiability of the system of linear equations e.g. $3 < x$ and $x = 2$ in `t2`. Coefficients for each inequality are passed as arguments e.g. $(\frac{1}{1}, \frac{1}{1})$ in `t2`. Steps `t3` and `t4` apply the `resolution` rule to the premises `a1`, `t2` (respectively `t1` and `t3`). Finally, the step `t6` concludes

Algorithm 2 la_generic rule

Let $\varphi_1, \ldots, \varphi_n$ be rational numbers linear inequalities, but different from $s_1 \approx s_2$, and $a_1, \ldots, a_n$ rational numbers. Then a `la_generic` step has the general form

$$i.\ \triangleright \quad \varphi_1, \ldots, \varphi_n \quad \texttt{la_generic} \quad [a_1, \ldots, a_n]$$

The constants a_i are of sort `Real`. To check the unsatisfiability of the negation of $\varphi_1, \ldots, \varphi_n$ one performs the following steps for each literal.
First, If all variables of $\varphi_1 \ldots \varphi_n$ are integer-sorted and the coefficients $a_1 \ldots a_n$ are in $\mathbb{Q}$, then $a_i := a_i \times lcd(a_1 \ldots a_n)$ where lcd is the least common denominator of $[a_1 \ldots a_n]$.
For each i, let $\varphi := \varphi_i$, $a := a_i$ and we write $s1 \bowtie s2$ to denote the left and right-hand sides of an inequality of the form (12).

1. If $\varphi = \neg(s_1 < s_2)$ or $s_1 \geq s_2$, then let $\varphi := \neg(-s_1 \geq -s_2)$. If $\varphi = \neg(s_1 \leq s_2)$ or $s_1 > s_2$, then let $\varphi := \neg(-s_1 > -s_2)$. If $\varphi = s_1 < s_2$, then let $\varphi := \neg(s_1 \geq s_2)$. If $\varphi = s_1 \leq s_2$, then let $\varphi := \neg(s_1 > s_2)$. Otherwise, leave φ unchanged. This step puts the literal in a normalized form, by negating it when needed. The goal is to produce a canonical form that uses only the operators $>$, $\geq$, and $=$.

2. Replace $\varphi = \neg(\sum_{i=0}^{n} c_i \times t_i + d_1 \bowtie \sum_{i=n+1}^{m} c_i \times t_i + d_2)$ by $\neg((\sum_{i=0}^{n} c_i \times t_i) - (\sum_{i=n+1}^{m} c_i \times t_i) \bowtie d_2 - d_1)$.

3. Now φ has the form $\neg(s_1 \bowtie d)$. If all variables in s_1 are integer-sorted then replace $\neg(s_1 \bowtie d)$ by $\neg(s_1 \bowtie \lceil d \rceil)$, otherwise by $\neg(s_1 \bowtie \lfloor d \rfloor + 1)$.

4. If $\bowtie$ is $=$, then replace φ by $\neg(\sum_{i=0}^{m} a \times c_i \times t_i = a \times d)$, otherwise replace it by $\neg(\sum_{i=0}^{m} |a| \times c_i \times t_i \bowtie |a| \times d)$.

5. Finally, the sum of the resulting literals is trivially contradictory,

$$\sum_{k=1}^{n} \sum_{i=1}^{m} c_i^k * t_i^k \bowtie \sum_{k=1}^{n} d^k$$

where c_i^k and t_i^k are the constant and term from the literal φ_k, and d^k is the constant d of φ_k. The operator $\bowtie$ is $=$ if all operators are $=$, $>$ if at least one is $>$, and $\geq$ otherwise. Finally, the sum on the left-hand side is 0 and the right-hand side is > 0 (or ≥ 0 if $\bowtie$ is $>$).

the proof by generating the empty clause $\bot$, denoted as `(cl)` in Listing 7. Notice that the contexts Γ of each step are all empty in this proof.

The `la_generic` rule can be verified by showing that the negated inequalities are jointly unsatisfiable. After applying strengthening rules,

the resulting conjunction is unsatisfiable, even when `Int` variables are treated as `Real` variables. Although the rule may introduce rational coefficients, they often reduce to integers—as shown in Listing 7, where the coefficients are $(\frac{1}{1}, \frac{1}{1})$. Cases where coefficients cannot be reduced to integers are rare in practice, however, we reduce them to integers by *clearing denominators* using their least common denominator.

The above algorithm is adapted from the Alethe specification [11], except that we clarified step 1: the subsequent steps in the original algorithm are designed for $>$ and $\geq$ and do not clearly address how to handle $<$ and $\leq$. Additionally, we added step 4 in order to ensure that our construction is independent of $\mathbb{Q}$.

Example 15. Consider the `la_generic` step in the logic `QF_UFLIA` with the uninterpreted function symbol `(f Int)`:

```
1 (step t11 (cl (not (<= f 0)) (<= (+ 1 (* 4 f)) 1))
2   :rule la_generic :args (1/1 1/4))
```

The algorithm then performs the following steps:

First, we Replace $a = [\frac{1}{1}, \frac{1}{4}]$ by $a = [4, 1]$ due to clearing denominators. We then obtain

$$\neg(-f > 0),\ \neg(4f > 0) \qquad \text{(Steps 1 and 2)}$$
$$\neg(-f > 0),\ \neg(4f \geq 1) \qquad \text{(Step 3)}$$
$$\neg(|4| * -f > |4| * 0),\ \neg(|1| * 4f \geq |1| * 1) \qquad \text{(Step 4)}$$
$$-4f + 4f \geq 1 \qquad \text{(Step 5)}$$

Which sums to the contradiction $0 \geq 1$.

SPECIAL RULES.

$$\text{i.}\quad \rhd \quad \varphi \quad (\texttt{hole}\ p_1, \ldots p_n)[a_1, \ldots a_m] \tag{13}$$

where φ is any well-formed formula.

This rule serves as a placeholder for incomplete proof steps not yet covered by the defined rules. A proof checker must reject any proof containing this rule, even if it can be verified. Checkers may, however, assign a special status to otherwise valid proofs containing it. Other tools may choose whether to accept or reject such proofs.

4.5 CHECKING ALETHE PROOFS

4.5.1 *Abstract procedure*

We present an abstract procedure to verify whether an Alethe proof is *well-formed* and *valid*. An Alethe proof is well-formed if and only if its anchors and steps are properly balanced. To check this property, we iteratively replace innermost subproofs by holes until no subproofs remain. If the resulting reduced proof is free of anchors and of steps that require a concluding rule, then the proof is well-formed.

Beyond well-formedness, a proof must also be valid. By validity we mean that every inference step in the proof respects the conditions

prescribed by its corresponding rule under the appropriate logical context. To check if a proof is valid we have to check if all steps of a subproof adhere to the conditions of their rules before replacing the subproof by a hole. If all subproofs are valid and all steps in the reduced proof adhere to the conditions of their rule, then the entire proof is valid. Consequently, a proof is valid if and only if all of its subproofs are valid and all of its top-level steps are valid.

Formally, an Alethe proof P is represented as a finite sequence $[C_1, \ldots, C_n]$ of steps and anchors. Since each step employs a unique index, we assume that step C_i in P uses i as its index. The logical context changes only at anchors and subproof-concluding steps. Therefore, the step elements $C_1, \ldots, C_n$ are not directly associated with a context. Instead, the context can be computed from the preceding anchors, as anchors serve exclusively to extend the context.

Definition 4.5.1 (First-Innermost Subproof)**.** Let P be the proof $[C_1, \ldots, C_n]$ and let $1 \leq \mathit{start} < \mathit{end} \leq n$ be two indices such that:

- C_{start} is an anchor,
- C_{end} is a step that uses a concluding rule,
- no C_k with $k < \mathit{start}$ uses a concluding rule, and
- no C_k with $\mathit{start} < k < \mathit{end}$ is an anchor or a step that uses a concluding rule.

Then $[C_{\mathit{start}}, \ldots, C_{\mathit{end}}]$ is called the *first-innermost subproof* of P. We denotes as P_{last} the final proof in the sequence obtained by iteratively eliminating first-innermost subproofs.

Definition 4.5.2 (Eliminate subproof)**.** Let $P = [C_1, \ldots, C_n]$ be a proof. A *first-innermost subproof* of P is $[C_{\mathit{start}}, \ldots, C_{\mathit{end}}]$ that contains no proper subproofs. If P contains no first-innermost subproof, then eliminating subproofs leaves P unchanged. Otherwise, we eliminate the first-innermost subproof by replacing it with a single step C' that uses the `hole` rule and has the index, conclusion, and premises of C_{end}. By repeatedly eliminating the first-innermost subproof of P, we obtain a finite sequence of proofs:

$$[P_0, P_1, P_2, \ldots, P_{\mathit{last}}]$$

where $P_0 = P$, each P_{i+1} is obtained from P_i by eliminating its first-innermost subproof, and P_{last} contains no subproofs. Since P is finite, this process always terminates after finitely many steps, and P_{last} is uniquely determined.

Definition 4.5.3 (Well-Formed Proof)**.** An Alethe proof P is well-formed if every step uses a unique index and P_{last} contains no anchor or or step that uses a concluding rule (`subproof`, `bind`, etc.).

Definition 4.5.4 (Valid Alethe Proof)**.** A proof P is a *valid Alethe proof* if and only if:

- P is well-formed,
- P does not contain any step that uses the hole rule,
- P_{last} contains a step that has the empty clause as its conclusion,
- the first-innermost subproof of every P_i with $i < last$ is valid,
- all steps C_i in P_{last} only use premises C_j in P_{last} with $1 \leq j < i$, and
- all steps C_i in P_{last} adhere to the conditions of their rule under the empty context.

The condition that P contains no holes ensures that the original proof is complete and holes are only introduced by eliminating valid subproofs.

It is sometimes useful to speak about the steps that are not within a subproof. We call such a step an *outermost step*. In a well-formed proof, these are precisely the steps of P_{last}.

Theorem 2 (Soundness of Concrete Alethe Proofs)**.** If there is a valid proof $P = [C_1, \ldots, C_n]$ that has the formulas $\varphi_1, \ldots, \varphi_n$ as the conclusions of the outermost assume steps, then

$$\varphi_1, \ldots, \varphi_n \models \bot$$

Here, $\models$ represents semantic consequence in the many-sorted first order logic of SMT-LIB with the theories of uninterpreted functions and linear arithmetic extended to clauses.

4.5.2 *Challenges in Proof Verification*

While the abstract procedure described above provides a clear framework for validating Alethe proofs, practical verification presents significant challenges due to the implicit nature of certain proof steps. The coarse-grained representation used by Alethe contrasts sharply with the fine-grained proofs required by proof assistants such as Lambdapi, creating a substantial gap that must be bridged during verification.

The primary difficulty stems from the fact that Alethe treats several fundamental logical operations implicitly, requiring proof checkers to reconstruct missing intermediate steps. This implicit treatment manifests in multiple ways, making direct verification challenging without additional proof reconstruction.

The following aspects are treated implicitly by Alethe:

- Symmetry of equalities that are not top-most equalities in steps with non-empty context;
- The order of literals in the clauses;
- The trace of rewrites in simplification steps is not provided by Alethe.

- The unfolding of names introduced by `(! t :named n)`;
- The unfolding of function symbols introduced by `define-fun`.

Moreover, in simplification rules, the order of the rewrites matters since some rules are not confluent. Additionally, some rules such as `la_generic`, correspond to algorithmic procedures (Algorithm 2) rather than to logical deductions.

These implicit operations necessitate sophisticated proof reconstruction algorithms that can infer and generate the missing intermediate steps, effectively transforming coarse-grained Alethe proofs into the fine-grained format required by formal verification systems.

5 ELABORATION OF ALETHE PROOFS

To improve the success rate of reconstructing Alethe proofs in Lambdapi, we leverage several elaboration passes provided by Carcara. These passes address common challenges in proof traces such as implicit symmetry, clause reordering, and coarse-grained steps by transforming them into fine-grained, explicit proofs that are easier to formalize and verify in Lambdapi.

5.1 ELABORATION WITH CARCARA

Unfortunately, Alethe proof traces provided by SMT solvers such as veriT and cvc5 can be challenging to reconstruct in a proof assistant. As we mentioned in Section 4.5.2, the order of literals in the clauses is not determined, *symmetry* of equality is sometimes used implicitly, and the `resolution` proof rule may not indicate *pivots* explicitly and remove duplicated literals implicitly.

Carcara provides an elaboration mechanism [3, §3] for Alethe proof traces where "elaboration passes" are applied to replace coarse-grained steps with fine-grained proofs of their conclusions, adding details that can make proof reconstruction easier. For example, a clause `(or p (= b a))` may later be used as the term `(or p (= a b))`, with the equality `(= b a)` implicitly reordered. Carcara applies an elaboration pass that introduces intermediate steps to convert the original assertion into one with equality reordered and consistently uses that version in the rest of the proof. The list of elaboration passes performed by Carcara can be found in [3, §3.2], and besides removing implicit equality reordering it includes removing some trivial transient steps, rewriting the order of literals in a clause, and computing a proof for arithmetic steps that lack explicit justification by calling an external SMT solver.

5.1.1 *Eliminating clause reordering steps*

In Alethe, the `reordering` rule is used to reorder the terms of a clause. Concretely, steps of this rule take a single clause as a premise, and conclude a permutation of it. Translating these steps into Lambdapi is not ideal, as formally proving that the two permutations are equivalent would be costly to check. Instead, we have added a new elaboration step to Carcara that completely removes all `reordering` steps from the proof.

Removing a `reordering` step may invalidate subsequent steps that depend on the original clause order. To address this, we must recompute the conclusion of these steps so that they are again valid, and

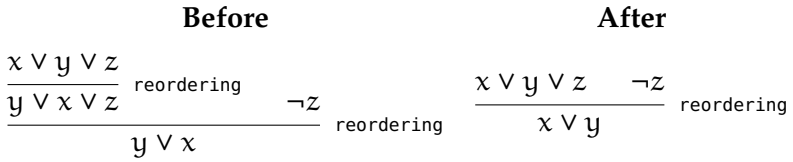


Figure 6: A proof snippet before and after elimination of `reordering` steps

keep propagating these changes throughout the proof. Fortunately, all rules in Alethe that depend on the order of the terms in a premise clause (namely, `resolution`, `contraction` and `or_intro`) can be recomputed to still be valid by only changing the order of the terms in the conclusion. No other rules are affected. This means that we do not need to add any new steps when doing this elaboration — changes in the conclusion of a step may only cause other steps to have their conclusion changed, down until the end of the proof.

The Figure 6 shows an example of this elimination in action. Note how the conclusion of the `resolution` step had to be recomputed — if any step further down in the proof used this `resolution` step as a premise, it would also have to be recomputed.

5.1.2 *Elaboration of resolution steps*

The `resolution` rule in Alethe is very flexible. Namely, it does not require the pivots to be provided, it allows implicit reorderings of terms in the conclusion clause, and it allows duplicate literals to be removed implicitly. This flexibility significantly complicates the translation of resolution steps to Lambdapi, so we again use elaboration in Carcara to address it. There is an existing elaboration step in Carcara that can compute the pivots used if they are not provided [3], and the implicit reordering of the conclusion clause can be handled by adding a `reordering` step. As for the implicit removal of duplicate literals, we had to develop a new elaboration procedure in Carcara to address it.

This elaboration algorithm makes the removal of duplicate literals explicit, using the `contraction` rule. Since in Alethe the `resolution` rule actually represents a chain of several binary resolution steps, we may need to break this chain into multiple `resolution` proof steps, with `contraction` steps in between. To do this, the algorithm first computes the "naive" conclusion of the resolution step, without removing any duplicates, and determines which literals are present in this "naive" conclusion, but not in the actual conclusion. These are called the *crowding* literals, and represent pivots whose duplicates were implicitly removed before they were eliminated. To make duplicate removal explicit, there needs to be, for each crowding literal, a contraction step between the last time it is introduced and the time it is eliminated.

Figure 7 shows an example of this elaboration in action. Notice that, in the original resolution chain, the b literal is introduced twice in the first two premises. However, when it is eliminated by the $\neg b$ literal

Before

$$\frac{\neg a \vee b \vee x \vee c \quad a \vee b \vee c \quad \neg b \vee y \vee z \quad \neg c}{x \vee y \vee z}\ \texttt{Res}\ a, b, c$$

After

$$\dfrac{\dfrac{\dfrac{\neg a \vee b \vee x \vee c \quad a \vee b \vee c}{b \vee x \vee c \vee b \vee c}\ \texttt{Res; a}}{b \vee x \vee c}\ \texttt{contra} \qquad \neg b \vee y \vee z \quad \neg c}{x \vee y \vee z}\ \texttt{Res}\ b, c$$

Figure 7: A resolution step before and after elaboration.

in the third premise, this duplicate is implicitly removed. The same thing happens with the c literal. In the elaborated `resolution` step, this duplicate removal is made explicit, with a `contraction` step.

5.1.3 *Elaboration of* `lia_generic` *steps*

The rule `lia_generic` is similar to `la_generic`, but omits the coefficients, i.e. $[a_1 \dots a_r]$ is empty. We decided to leverage the elaboration process of `lia_generic` performed by Carcara, as doing otherwise would require implementing Fourier-Motzkin elimination for integers, as done in [19, 80], hence reimplementing work that was already done by the solver.

Carcara treats `lia_generic` steps as proof holes, since "their checking is as hard as solving" [3, §3.2]. To address this, Carcara leverages an external tool that reformulates each `lia_generic` step as a separate problem and produces Alethe proofs not containing `lia_generic` steps. The proof is then imported and validated, replacing the original step. Thus, the step

```
(step S (cl (not l1) ... (not ln)) :rule lia_generic)
```

concluding the clause $\neg l_1 \vee \dots \neg l_n$ where all l_i are inequalities, generates an SMT-LIB problem asserting $l_1, \dots, l_n$ and invokes the solver cvc5 on it, expecting an Alethe proof $\pi : (l_1 \wedge \dots \wedge l_n) \to \bot$ that does not use `lia_generic`. Carcara will check this subproof and then replace the original step by a proof of the form

```
1 (anchor :step S.t_m+1)
2 (assume S.h_1 l1)
3 ...
4 (assume S.h_n ln)
5 ...
6 (step t.t_m (cl false) :rule ...)
7 (step t.t_p (cl (not l1) ... (not ln) false) :rule subproof)
8 (step t.t_q (cl (not false)) :rule false)
9 (step S (cl (not l1) ... (not ln)) :rule resolution :premises (S.t_p S.t_q))
```

Listing 8: Elaboration of `lia_generic`

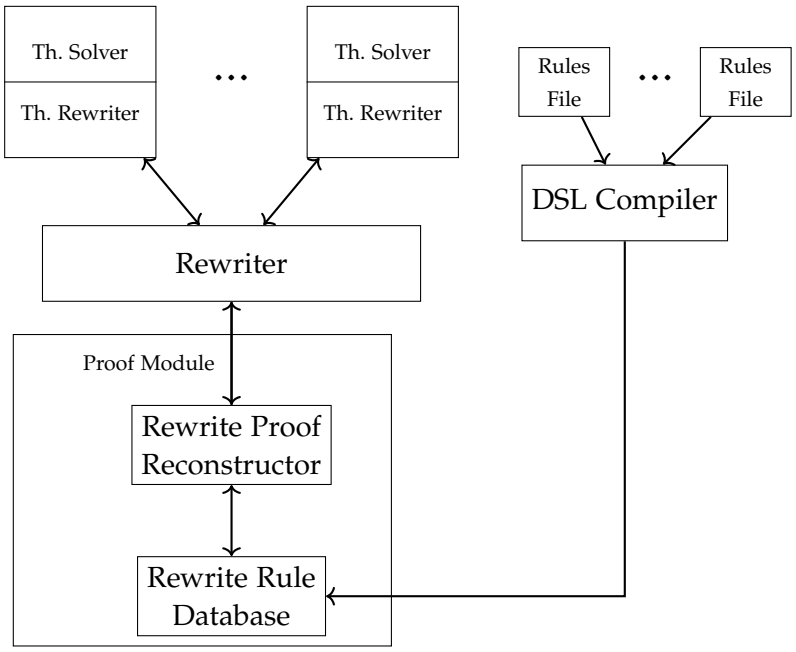


Figure 8: Overview of the components of our approach

5.2 ELABORATION OF SIMPLIFICATIONS WITH RARE

To reconstruct coarse-grained simplification proofs, we use the RARE framework [78]. Modern SMT solvers implement hundreds of theory-specific rewrite rules for normalizing and simplifying terms to achieve state-of-the-art performance. Conceptually, the implementation of these theory-specific rewrite rules can be viewed as *theory rewriter modules* within individual theory solvers. A *Rewriter* is a module that traverses a given term and invokes the appropriate theory rewriter on each subterm by examining the top-most symbol of the subterm and calling the theory whose signature contains that symbol.

However, instrumenting rewriters is challenging, as it requires modifying performance-critical code without sacrificing efficiency. RARE provides a domain-specific language for declaratively specifying rewrite rules and automatically reconstructing detailed proofs from coarse-grained atomic rewrites, making it particularly well-suited for generating the fine-grained simplification proofs required to reconstruct Alethe proofs in the Lambdapi proof assistant. The RARE framework has been implemented in cvc5 and replaces the original simplification rules of Alethe with its built-in rules.

RARE treats the SMT solver's rewriter as a black box. It uses post-processing to expand coarse-grained rewrites into fine-grained proofs (Figure 8). The proof module, which manages proofs, utilizes a rewrite proof reconstructor to fill in the missing subproofs for rewrites. The rewrite proof reconstructor bases its reconstruction on a set of rewrite rules stored in a rewrite rule database (Rules file), which is generated at compile-time from declaratively specified rules written in the RARE domain-specific language.

RARE supports three categories of rewrite rules that are particularly useful for Alethe simplification proofs. Basic unconditional rules handle straightforward rewrites, such as

```
(define-rule arith-refl-leq ((t ?)) (<= t t) true)
```

which simplifies any `t` ≤ `t` into `true`. The ? denotes that the term can be of any theory sort or `Bool`. In contrast to SMT-LIB, parameterized sorts such as arrays and bit-vectors do not need to be concrete. Instead, RARE is gradually typed and allows the parameters of such sorts to remain abstract. This enables the specification of rewrites that are, e.g., independent of the bit-width or the sorts. Conditional rules enable context-dependent simplifications, exemplified by rules:

```
(define-cond-rule bool-not-true ((t Bool))
    (= t false) (not t) true)
```

that simplifies `(not t)` to `true` under the condition that `t` evaluates to `false`. Fixed-point rules support iterative simplifications, such as:

```
(define-rule* bool-or-de-morgan ((x Bool) (y Bool) (zs Bool :list
))
  (not (or x y zs))
  (not (or y zs))
  (and (not x) _))
```

that implements De Morgan's law by iteratively rewriting:

$$\neg(x \vee y \vee \dots) \text{ to } (\neg x \wedge \neg(y \vee \dots))$$

Fixed-point rules take a match expression, a target expression, and, optionally, a context expression as arguments. The target expression indicates the recursion step, i.e., the term that should be rewritten next. The context expression indicates how to use the result of the recursion step to construct the final result. It is a term with a placeholder _ for the location of the result of the recursion step. The variables declared with the `:list` attribute to match an arbitrary number of arguments of an operator. In `bool-or-de-morgan`, the match is `(not (or x y zs))`, the recursive target is `(not (or y zs))`, and the reconstruction context is `(and (not x) _)`.

The RARE rule used and its arguments are made explicit in the `:args` parameter of the rule `rare_rewrite` in the proof trace. cvc5 elaborates the proof by rendering each coarse step into a fined-grain one with atomic rewrite. Although RARE is currently only supported by cvc5, its use has allowed us to increase the success rate of proof reconstruction.

Example 16 demonstrates how RARE refines an Alethe proof trace into a finer-grained version. The original proof step uses the coarse rule `or_simplify`, implicitly applying the transformation `bool-or-false` twice. When RARE is enabled, cvc5 replaces this with explicit steps, each annotated with the rule and its arguments. For instance, the original step:

Example 16 (RARE example)**.** An original step `or_simplify` below

```
(step t1 (cl (= (or false x y false z) (x y z))) :rule or_simplify)
```

is elaborated using the RARE rule:

```
(define-rule* bool-or-false ((xs Bool :list) (ys Bool :list))
  (or xs false ys)
  (or xs ys))
```

This results in the following explicit proof steps:

```
1 (step t1 (cl (= (or false x y false z) (or x y false z)))
2       :rule rare_rewrite :args ("bool-or-false" rare-list
3                                 (rare-list (x y false z))))
4 (step t2 (cl (= (or x false y z) (or x y z)))
5       :rule rare_rewrite :args ("bool-or-false" (rare-list x)
6                                 (rare-list (y z))))
```

6 RELATED WORKS

6.1 THE EUNOIA PROOF FORMAT

Eunoia is a ongoing development of a logical framework that provides a formally defined language for representing proof systems and serves as the specification basis for the recently introduced Cooperating Proof Calculus (CPC) employed by the SMT solver cvc5. [1] It is grounded in the SMT-LIB 3 standard [2], which generalizes and extends the widely adopted SMT-LIB 2 language in order to support higher-order logic with polymorphism and dependent types but classical semantics. The current general syntax for dependent types is:

```
(→ (! A :var x) B)
```

The symbol → will denote the function type constructor, where `x` is a bound variable whose scope is `B`. This syntax can be understood as an abbreviation of the λΠ type or kind $\Pi\, x : A, B$.

In spirit, Eunoia is comparable to the *Alethe* proof format. However, while Alethe primarily aims to provide a standardized exchange format for SMT solver proofs, Eunoia is conceived as a more general logical framework in which proof systems can be defined, extended, and modularly verified. Unlike Alethe, which relies on a predefined set of proof rules, Eunoia allows users to define their own custom rules using the `declare-rule` command, offering greater flexibility and extensibility in proof construction. For example:

```
1 (declare-parameterized-const = ((T Type :implicit)) (→ T T Bool))
2 (declare-rule symm ((T Type) (t T) (s T))
3     :premises ((= t s))
4     :conclusion (= s t))
5 (declare-type Int ())
6 (declare-const a Int)
7 (declare-const b Int)
8 (assume @p0 (= a b))
9 (step @p1 (= b a) :rule symm :premises (@p0))
```

Listing 9: An Eunoia proof example

The rule `symm` (at line 2) specifies symmetry of equality. This rule takes as premise an equality `(= t s)` and no arguments. In detail, an application of this proof rule for premise proof `(= a b)` for concrete terms a, b will compute the substitution $[t \mapsto a, s \mapsto b]$ and apply it to the conclusion term to obtain `(= b a)`. The Eunoia proof format

1 See https://cvc5.github.io/docs/cvc5-1.2.1/proofs/output_cpc.html. A complete list of the proof rules in CPC can be found at https://cvc5.github.io/docs/cvc5-1.2.1/api/cpp/enums/proofrule.html. The semantics of the rules is also defined in the Eunoia logical framework, described in the user manual of the Ethos proof checker: https://github.com/cvc5/ethos/blob/main/user_manual.md.

2 https://smt-lib.org/version3.shtml

follows a structure that is almost identical to that of Alethe (`assume`, `step`) as we can observe lines 8 and 9.

The Listing 9 shows an example of an Eunoia proof script. Line 1, a polymorphic binary predicate = is declared. The proof rule `symm` (lines 2 to 5) is then defined to capture the symmetry property of equality: if `t = s`, then `s = t`. Following this, an integer type `Int` (line 6) and two integer constants, `a` and `b`, are declared (lines 7,8). The proof begins line 8 by assuming the equality of `a` and `b`, labelled as `@p0`. Finally, at line 9, the symmetry rule `symm` is applied to derive the equality of `b` and `a`, labelled as `@p1`, using the assumption `@p0` as the premise. This concise proof demonstrates Eunoia's capability to formally represent and verify basic properties of equality, highlighting its support for polymorphic types and structured proof rules.

Recent work[3] has focused on embedding the Alethe proof format within Eunoia, ensuring backward compatibility and enabling users to leverage existing Alethe proofs in the more general Eunoia framework.

6.2 PROOF RECONSTRUCTION FOR HAMMERS

Several proof assistants employ *hammers* for discharging goals to ATPs, including SMT solvers. Examples include Sledgehammer [20] for Isabelle/HOL and CoqHammer [39, 40] for Coq. The hammer translates the conjecture and facts supplied by the user or harvested from the context to the input language of the back-end, invokes it, and in the case of success attempts to reconstruct the proof in the logic of the proof assistant, based on a trace of the proof found by the back-end.

This process can benefit significantly from detailed proofs by the back-ends. For example, adoption by Sledgehammer of the Alethe format generated by the SMT solver veriT [10, 89] cut the failure rate of reconstruction by 50%. According to Sledgehammer benchmarks, proof reconstruction now succeeds in over 90% of cases [50, 61], a substantial improvement over previous formats. Across other formalizations, including the Prime Number Theorem and Simplex algorithm, the failure rate was reduced by 56% to 63%, consistently enhancing the reliability of the proof reconstruction pipeline. The introduction of fine-grained proof formats and optimized solver strategies not only increased the number of automatically reconstructed proofs but also reduced the total reconstruction time by up to 15%. These results encouraged us to base our approach on the Alethe trace format.

In Sledgehammer, the reconstruction procedure uses the RARE framework with IsaRare [60], a tool that can automatically translate RARE rules into Isabelle/HOL lemmas. For arithmetic steps, it uses Isabelle's procedure `linarith`. This tactic is a decision procedure for real numbers, but not for integers or natural numbers. Internally, it uses the Fourier–Motzkin elimination [85]: it derives a contradiction via a linear combination of the equations.

3 https://github.com/cvc5/AletheInEunoia

In CoqHammer, proof reconstruction is performed by an *auto*-type algorithm implemented in Ltac [43]. The proof search procedure does primarily backward Prolog-style reasoning modifying the goal by applying hypotheses from the context. The core of the search procedure may be seen as an extension of the Ben-Yelles algorithm to first-order intuitionistic logic with all connectives [98]. The proof reconstruction procedure is augmented with the use of existential metavariables like in `eauto`, a looping check, some limited forward reasoning, the use of the `congruence tactic`, and heuristic rewriting using equational hypotheses. CoqHammer is typically evaluated on a benchmark consisting of 4841 lemmas from the Coq standard library and the Mathematical Components library, which serves as the standard benchmark for hammering in Coq. The reconstruction mechanism currently is able to re-prove only 85.2% (4215 out of 4841) of the proofs founds by the ATPs. It is capable of re-proving over 40% of Coq standard library lemmas.

6.3 SMT INTEGRATION IN PROOF ASSISTANTS

In contrast to Hammers, which are designed to interface with general-purpose automated theorem provers (ATPs) such as Vampire [59], E Prover [86], or SMT solvers [73], other approaches establish direct connections between proof assistants and SMT solvers. For instance, SMTCoq [4] and Lean-SMT [71] integrate Coq and Lean, respectively, with external SAT and SMT solvers, thereby leveraging the theory reasoning capabilities of SMT solvers that are not fully exploited by Hammers.

SMTCOQ is a fully certified Coq plugin that checks proof witnesses coming from external SAT and SMT solvers. For proof reconstruction, SMTCoq relies on computational reflection: the certificate is directly processed by the reduction mechanism of Coq's kernel. We initially attempted to convert the SMTCoq proof certificate to Lambdapi. However, we found the proof term generated by computational reflection, including subterms corresponding to arithmetic decision procedures in Micromega [19], to be too complex to be converted to Lambdapi. Moreover, SMTCoq is based on an old version of the Alethe proof format, which is only supported by older versions of the veriT solver.

LEAN-SMT is a tactic that provides SMT solver integration in the Lean proof assistant, similar to how SMTCoq integrates SMT solvers with Coq. Lean-SMT operates by translating Lean proof goals into SMT-LIB problems and leveraging the proof-producing SMT solver cvc5 to generate proofs, which are then reconstructed as native Lean proofs. Lean-SMT employs the Cooperating Proof Calculus (CPC) based on Eunoia. Unlike SMTCoq, which employs a formally verified checker that validates entire SMT proofs within Coq, Lean-SMT uses a proof replay approach that reconstructs each individual proof step within Lean. This choice was motivated by performance con-

siderations: while Coq excels at efficient proof checking, Lean does not [6]. Additionally, the proof replay approach offers greater flexibility compared to SMTCoq's verified checker, as modifications to the supported proof format only require changes to the corresponding reconstruction tactics rather than re-proving the entire checker's correctness theorem. Lean-SMT currently supports around 200 of cvc5's proof rules which amounts to approximately 30%, and has been evaluated on both Sledgehammer benchmarks and SMT-COMP benchmarks, showing competitive performance with other proof checkers while maintaining a small trusted base.

6.4 PROOF CHECKERS FOR SMT SOLVERS

CARCARA [3] is an independent tool, implemented in Rust [94], that acts as both a *proof checker* and a *proof elaborator* for SMT proof traces in the Alethe format. As a proof checker, it ensures the correctness of proofs by validating each step against the logical rules, thus guaranteeing the soundness of the entire trace. As a proof elaborator, it refines high-level or incomplete proofs into fully detailed, low-level versions by filling in implicit steps or normalizing terms. Unlike certified checkers such as SMTCoq or Lean-SMT, Carcara supports the elaboration of coarse-grained proof traces—including implicit steps—into fine-grained ones, which can simplify verification in proof assistants. This capability is particularly valuable for our work, as it compensates for the limited meta-programming support in Lambdapi's vernacular language.

ETHOS is a proof checker currently under active development [4], written in C++, designed to verify proofs produced in the Eunoia format. Analogously, Ethos plays a role similar to that of Carcara, which provides an independent proof checker for SMT proofs.

6.5 THE UNVERIFIED BACKEND PROBLEM

Beyond the hammer and SMT integration approaches discussed, we highlight formal verification tools that rely on SMT solvers for automation but lacks proof verification capabilities, thereby presenting an opportunity to apply the reconstruction methods developed in this thesis.

THE B METHOD The B method [1] has been increasingly adopted in the development of high-assurance software systems, particularly in safety-critical domains such as railways. The verification of B models is facilitated by integrated development environments such as Atelier B [30] and Rodin [2], which provide a unified framework for both system development and formal verification. Proof obligations are systematically generated from B components and translated into machine-verifiable formats to be fed into automated solvers. In these

4 https://github.com/cvc5/ethos/tree/main

environments, proof obligations are automatically generated from the B components and translated into machine-verifiable formulas for processing by automated solvers. One class of such solvers is SMT solver, a dedicated backend [42], and its modern successor BEer [97], can encode B proof obligations into SMT. However, while SMT solvers are effective at finding proofs, the correctness of the proofs they produce still requires independent verification to ensure complete trustworthiness. In this context, we can see a direct application of the work presented in this thesis, as our reconstruction framework can serve as a foundation for certifying SMT proofs generated during the verification of B models.

TLA$^+$ AND TLAPS TLA$^+$ is a specification language based on the Temporal Logic of Actions and Zermelo–Fraenkel set theory with the Axiom of Choice (ZFC) [63, 64]. It is particularly used for specifying concurrent and distributed systems. The TLA$^+$ Proof System (TLAPS) extends the language with a formal proof syntax and a tactical proof language [36].

TLAPS includes a *Proof Manager* that decomposes proofs into individual proof obligations, which are then dispatched to various backend provers. These currently include Isabelle/TLA$^+$ [77], Zenon [44], the SMT solvers cvc5 [12], veriT [28] and Z3 [73], and finally the LS4 prover for temporal logic. Each obligation is translated into the logic supported by the corresponding backend. For SMT solvers [41], the supported logics include **UF**, **LIA**, and **NIA**. At present, TLAPS does not verify the correctness of solver outputs, except in the case of Zenon, whose proofs can be independently validated by Isabelle, and in the case of Isabelle/TLA$^+$ backend.

The work that will be discussed in Part ii can be leveraged to verify TLA$^+$ proofs from the SMT solver backend. The proposed approach involves extracting the output generated by the SMT backend, retrieving the corresponding proof from cvc5 or veriT, and reconstructing it within Lambdapi to establish its validity.

Validating TLAPS proofs

TLA$^+$ proofs are hierarchically structured and are generally written top-down. The assertion that the Cantor theorem stating that no function (f) from a set (S) to its power set (SUBSET S) can be surjective is formalized in TLA$^+$ as the theorem in Figure 9. Each proof in the hierarchy ends with a QED step that asserts the proof's goal. However, in Figure 9, the proof is flat and concludes with a unique QED step that asserts the proof's goal. Proofs are incrementally decomposed into simpler subproofs across multiple levels. Each proof consists of labelled steps ⟨i⟩k., that reflect their depth in the proof tree, where level i indicates the nesting depth. Steps may introduce assumptions, intermediate claims, or invoke previously proven results (with BY), using TLAPS tactics such as ASSUME/PROVE `take` or `witness`. The

MODULE *Cantor*

```
THEOREM cantor ≜
∀S :
  ∀f ∈ [S → SUBSET S] :
    ∃A ∈ SUBSET S :
      ∀x ∈ S :
        f[x] # A
⟨1⟩1. TAKE S
⟨1⟩2. TAKE f ∈ [S → SUBSET S]
⟨1⟩3. DEFINE T ≜ {z ∈ S : z ∉ f[z]}
⟨1⟩4. WITNESS T ∈ SUBSET S
⟨1⟩5. TAKE x ∈ S
⟨1⟩6. QED BY x ∈ T ∨ x ∉ T
```

Figure 9: Cantor theorem in TLA$^+$

```
ASSUME NEW CONSTANT S,
       NEW CONSTANT f ∈ [S → SUBSET S],
       NEW CONSTANT X ∈ S,
         X ∈ T ∨ X ∉ T
PROVE  f[X] ≠ T
```

Figure 10: Proof obligation of step ⟨1⟩6.

proof is completed when the root goal is reduced to a collection of leaf obligations marked as QED and discharged by backend provers.

For each step ⟨i⟩k. in TLA$^+$ proof, there is a set of proof obligations that has to be proved. The obligation contains a context of known facts, definitions, declarations, and a goal. Obligations are dispatched by the proof manager (PM) to the available backends (SMT solvers, Zenon, Isabelle/TLA) in a default order (SMT, Zenon, Isabelle) until one of them successfully proves the obligation. In the case of Figure 9, we will have 5 proof obligations (⟨1⟩1 – ⟨1⟩6 except ⟨1⟩3) that will be discharged to the backend. For example, the proof obligation relative to the final step ⟨1⟩6. is given as follows:

It assumes an arbitrary set S, a function $f \in [S \to \text{SUBSET } S]$, and an arbitrary element $X \in S$. The goal is to prove that $f[X] \neq \{z \in S \mid z \notin f[z]\}$. This corresponds to the diagonalization step: given the definition $T \triangleq \{z \in S \mid z \notin f[z]\}$ from ⟨1⟩3., we aim to show that T is not in the image of f. To this end, we must show that for any $X \in S$, $f[X] \neq T$. The assumption $X \in T \vee X \notin T$ provides a complete case split, and in either case, the assumption $f[X] = T$ leads to a contradiction, which suffices to discharge the proof obligation.

```
1  (set-logic UFLIA)
2  (declare-sort Idv 0)
3  (declare-fun Mem (Idv Idv) Bool)
4  (declare-fun FunApp (Idv Idv) Idv)
5  (declare-fun TrigEq_Idv (Idv Idv) Bool)
6
7  (declare-fun S () Idv)
8  (declare-fun f () Idv)
9
10 ;; f ∈ [S → subset S]
11 (assert (Mem f (FunSet S (Subset S))))
12
13 (declare-fun X () Idv)
14
15 ;; take x ∈ S
16 (assert (Mem X S))
17
18 (declare-fun SetSt (Idv) Idv)
19
20 (assert
21   (!
22     (forall ((x Idv) (y Idv))
23       (! (= (TrigEq_Idv x y) (= x y))
24         :pattern ((TrigEq_Idv x y))))
25     :named |ExtTrigEqDef Idv|))
26
27 (assert
28   (! (forall ((a Idv) (x Idv))
29     (!
30         (=
31             (Mem x (SetSt a))
32             (and (Mem x a) (not (Mem x (FunApp f x)))))
33     :pattern ((Mem x (SetSt a)))
34     :pattern ((Mem x a)
35                 (SetSt a))))
36     :named |SetStDef SetSt|))
37
38 ;; x ∈ T ∨ x ∉ T
39 (assert (or (Mem X (SetSt S)) (not (Mem X (SetSt S)))))
40
41 (assert (!
42     (not (not (TrigEq_Idv (FunApp f X) (SetSt S))))
43     :named |Goal|))
44 (check-sat)
```

Listing 10: Proof obligation of step ⟨1⟩6. in Cantor theorem

There would be 4 proof obligations that will be covered by the SMT solvers, and 2 by Zenon. The SMT backend of TLA$^+$ generates the corresponding SMT input problem presented in Listing 10. Since TLA$^+$ is based on ZFC which is untyped, the backend introduces a universal uninterpreted sort `Idv` (line 2) to represent the domain of all values. Within this setting, the function symbol `Mem` (line 3) denotes set membership ($\in$) , while `FunApp` (line 4) encodes function application (i.e. $f[x]$) . The declared constant symbols `S` and `f` correspond to the set S and function f in the original proof. The auxiliary function `SetSt_flatnd_1` defines the diagonal set $T = \{z \in S \mid z \notin f[z]\}$ by comprehension; this is specified by a universally quantified axiom stating that an element X belongs to `SetSt_flatnd_1 A` if and only if $X \in A$ and $X \notin f[X]$. This axiom captures the definition of T in a first-order setting. The disjunctive assertion `(or (Mem x ...) (not (Mem x ...)))` line 39 encodes a *classical* reasoning by cases over whether x belongs to T, reflecting how the proof splits based on $x \in T$ or $x \notin T$

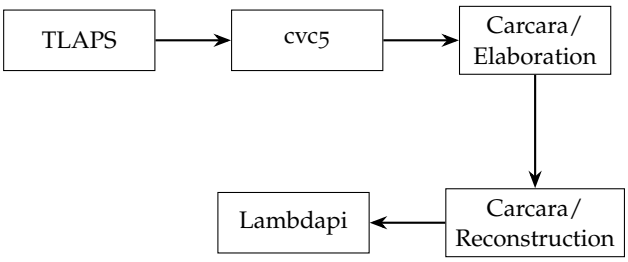


Figure 11: Complete Verification Pipeline for TLAPS Proofs

in the original TLA$^+$ argument. Finally, the goal is asserted as the double negation of the equation `(FunApp f x) = (SetSt_flatnd_1 S)`, encoded via a triggerable equality predicate `TrigEq_Idv`. Triggers play a crucial role in optimizing the encoding, as discussed in [41, §3]. This encoding faithfully mirrors the structure of the original TLA$^+$ proof obligation Figure 10. We used the *Cantor* example to illustrate the general approach taken to encode set-theoretic reasoning, as all proof obligations produced by the backend follow a similar structure. Full details of the SMT encoding for TLA$^+$ are provided in [41].

By leveraging the work presented in this thesis, we can establish a complete verification pipeline for TLAPS proofs as illustrated in Figure 11: the SMT solver proof traces can be elaborated using Carcara to produce detailed Alethe format proofs, which are then reconstructed in Lambdapi using our reconstruction framework to verify their correctness. This approach would transform TLAPS from a partially verified proof assistant into a fully certified system where all SMT-backed proofs undergo rigorous verification, thereby enhancing the trustworthiness of TLA+ specifications and their proofs.

Part II

CONTRIBUTIONS

7 ENCODING THE LOGIC OF SMT IN LAMBDAPI

We describe our embedding of the logic of SMT and Alethe into Lambdapi. Our work applies to input problems expressed in the logics `UF`, `LIA/LA` or their sub-logics. We begin by detailing the encoding of the SMT-LIB sorts used in these logics, which is necessary for reconstructing their corresponding Alethe proofs. We then present our method for translating these proofs.

7.1 A PRELUDE ENCODING FOR ALETHE

We base our following definitions on the prelude presented in Section 2.5. We represent user-declared SMT sorts in $\lambda\Pi/\equiv$ as elements of type `Set`. We represent the terms of sort **`Bool`** of SMT by elements of type `El o`. The constructor $\rightsquigarrow$ (written infix) is used to encode SMT functions and predicates. Since Alethe extends the SMT format with the `choice` operator ([11, §2.1]), we add it to the prelude.

Definition 7.1.1. We define the choice operator named ϵ.

$$\epsilon : \Pi\,[a : \texttt{Set}], (\texttt{El}\ a \to \texttt{Prop}) \to \texttt{El}\ a \tag{14}$$

$$\epsilon_i : \Pi\,[a : \texttt{Set}], \Pi(x : \texttt{El}\ a), \Pi(P : \texttt{El}\ a \to \texttt{Prop}) \tag{15}$$
$$\to \texttt{Prf}\,(P\,x) \to \texttt{Prf}\,P\,(\epsilon\,P)$$

$$\epsilon_d : \Pi\,[a : \texttt{Set}], \Pi(P\,Q : \texttt{El}\ a \to \texttt{Prop}) \tag{16}$$
$$\to \Pi\,x : \texttt{El}\ a, \texttt{Prf}\,(P\,x = Q\,x) \to (\epsilon\,P = \epsilon\,Q)$$

The intended interpretation for the ϵ operator is that $(\epsilon\,x,\ P\ x)$ denotes some x satisfying the predicate P, if there is one. The characteristic proof rules for the ϵ operator are represented by the constants ϵ_i for proving a predicate $P\,(\epsilon\,P)$ by exhibiting a witness x such that $P\,x$ is provable, and ϵ_d that asserts that the epsilon operator assigns the same witness to equivalent predicates P and Q.

Overall, this corresponds to a shallow embedding, where the logical connectives in SMT are directly represented by their counterparts in Lambdapi. SMT logic enjoys the property of propositional completeness (also referred to as *propositional degeneracy*) asserting that:

$$\forall p, (p = \top) \vee (p = \bot)$$

Moreover, propositionally equivalent formulas are equal.

In order to interface with SMT solvers—whose reasoning are based on classical logic, we extend our system with the law of excluded middle and propositional extensionality as an additional axioms.

Definition 7.1.2 (Classical facts)**.** We introduce the classical axioms:

$$\texttt{em} : \Pi\,[p : \texttt{Prop}], \texttt{Prf}\,(p \vee \neg p)$$
$$\texttt{prop_ext} : \Pi\,[p\ q : \texttt{Prop}], \texttt{Prf}\,(p \iff q) \to \texttt{Prf}\,(p = q)$$

7.2 REPRESENTING LINEAR INTEGER ARITHMETIC (LIA)

Definition 7.2.1 (Encoding integers)**.** The definition we use of natural numbers and integers in Lambdapi follows a common encoding found in many other theories, including the one in the Rocq standard library [93].

$\mathbb{N} : \texttt{TYPE}$	$\mathbb{P} : \texttt{TYPE}$	$\mathbb{Z} : \texttt{TYPE}$
$\mid 0 : \mathbb{N}$	$\mid \texttt{H} : \mathbb{P}$	$\mid \texttt{Z0} : \mathbb{Z}$
$\mid +1 : \mathbb{N} \to \mathbb{N}$	$\mid \texttt{O} : \mathbb{P} \to \mathbb{P}$	$\mid \texttt{ZPos} : \mathbb{P} \to \mathbb{Z}$
	$\mid \texttt{I} : \mathbb{P} \to \mathbb{P}$	$\mid \texttt{ZNeg} : \mathbb{P} \to \mathbb{Z}$
$\texttt{nat} : \texttt{Set}$	$\texttt{pos} : \texttt{Set}$	$\texttt{int} : \texttt{Set}$
$\texttt{El nat} \hookrightarrow \mathbb{N}$	$\texttt{El pos} \hookrightarrow \mathbb{P}$	$\texttt{El int} \hookrightarrow \mathbb{Z}$

Natural numbers are defined inductively with the two constructors 0 and the the successor function +1 written postfix. The type $\mathbb{P}$ is an inductive type representing strictly positive integers in binary form. Starting from 1 (represented by the constructor `H`), one can add a new least significant digit via the constructor `O` (digit 0) or the constructor `I` (digit 1). For example, we represent 5 (101b) as `I(O(H))`. The type $\mathbb{Z}$ represents integers in binary form. An integer is either zero (with constructor `Z0`) or a strictly positive number `ZPos` (coded as a $\mathbb{P}$) or a strictly negative number `ZNeg`. To enable quantification over elements of these types, we introduce constants such as `int : Set` that represent codes for these types along with a rule for rewriting codes to their corresponding types, for example `El int` $\hookrightarrow \mathbb{Z}$.

We make use of Lambdapi's **`builtin`** mechanism to enable decimal notation for numeric values, for instance, writing 2 for `ZPos(O H)` to convert natural representations of values into elements of $\mathbb{Z}$.

Definition 7.2.2 ($\mathbb{Z}$ operators)**.** Arithmetic operations over these types are implemented using rewrite rules, which necessitates proofs of both confluence and termination. We provide only the definitions of $(+)$ and $(*)$ over $\mathbb{Z}$:

```
+ : ℤ → ℤ → ℤ
Z0 + y ↪ y
x + Z0 ↪ x
(Zpos x) + (Zpos y) ↪ (ZPos (add x y))
(Zpos x) + (Zneg y) ↪ (sub x y)
(Zneg x) + (Zpos y) ↪ (sub y x)
(Zneg x) + (Zneg y) ↪ ZNeg (add x y)

* : ℤ → ℤ → ℤ
Z0 * _ ↪ Z0
_ * Z0 ↪ Z0
ZPos x * ZPos y ↪ ZPos(mul x y)
ZPos x * ZNeg y ↪ ZNeg(mul x y)
ZNeg x * ZPos y ↪ ZNeg(mul x y)
ZNeg x * ZNeg y ↪ ZPos(mul x y)
```

The definition of the auxiliaries functions `add`, `sub` and `mul` can be found in Section A.2.

Notation. We will use the notation $+_{\mathbb{Z}}$ or $*_{\mathbb{Z}}$ to refer the operators defined in Definition 7.2.2 when the context needs to be specified.

Definition 7.2.3 (Inequality operators for integer)**.** We introduce inductive types `Comp` and $\mathbb{B}$ representing comparison operators and Booleans.

```
Comp : TYPE
| Eq : Comp
| Lt : Comp
| Gt : Comp
comp : Set
El comp ↪ Comp
```

```
𝔹 : TYPE
| true : 𝔹
| false : 𝔹

bool : Set
El bool ↪ 𝔹
```

The type $\mathbb{B}$ is a simple inductive type with two constructors to represent binary value and providing an induction principle, while the type `Prop` represent the universe of propositions without an induction principle.

The function $\doteq$ below compares two integers, relying on the auxiliary function `cmp` (defined in Definition A.2.3) to perform comparisons between strictly positive binary numbers.

```
≐: ℤ → ℤ → Comp
Z0 ≐ Z0 ↪ Eq
Z0 ≐ ZPos _ ↪ Lt
Z0 ≐ ZNeg _ ↪ Gt
ZPos _ ≐ Z0 ↪ Gt
ZPos p ≐ ZPos q ↪ cmp p q
ZPos _ ≐ ZNeg _ ↪ Gt
ZNeg _ ≐ Z0 ↪ Lt
ZNeg _ ≐ ZPos _ ↪ Lt
ZNeg p ≐ ZNeg q ↪ cmp q p
isGt : Comp → 𝔹
isGt Eq ↪ false
isGt Lt ↪ false
isGt Gt ↪ true
```

Based on this comparison mechanism, we define inequality operators over $\mathbb{Z}$ as binary predicates, by reducing them to the decidable comparison $\doteq$. They reduce to $\top$, $\bot$ (or negated).

```
isEq : Comp → 𝔹
isEq Eq ↪ true
isEq Lt ↪ false
isEq Gt ↪ false

isLt : Comp → 𝔹
isLt Eq ↪ false
isLt Lt ↪ true
isLt Gt ↪ false

isGt : Comp → 𝔹
isGt Eq ↪ false
isGt Lt ↪ false
isGt Gt ↪ true

istrue : 𝔹 → Prop
istrue true ↪ ⊤
istrue false ↪ ⊥
```

$$\leq : \mathbb{Z} \to \mathbb{Z} \to \texttt{Prop} \coloneqq \lambda x, \lambda y, \neg(\texttt{istrue}(\texttt{isGt}(x \doteq y)))$$
$$< : \mathbb{Z} \to \mathbb{Z} \to \texttt{Prop} \coloneqq \lambda x, \lambda y, (\texttt{istrue}(\texttt{isLt}(x \doteq y)))$$
$$\geq : \mathbb{Z} \to \mathbb{Z} \to \texttt{Prop} \coloneqq \lambda x, \lambda y, \neg(x < y)$$
$$> : \mathbb{Z} \to \mathbb{Z} \to \texttt{Prop} \coloneqq \lambda x, \lambda y, \neg(x \leq y)$$

Example 17 (Inequality reduction)**.** For example,

$$\begin{aligned} 1 < 2 &\hookrightarrow \texttt{istrue}(\texttt{isLt}(1 \doteq 2)) \\ &\hookrightarrow \texttt{istrue}(\texttt{isLt}(\texttt{Lt})) \\ &\hookrightarrow \texttt{istrue}(\texttt{true}) \\ &\hookrightarrow \top, \end{aligned}$$

with $1 = \texttt{ZPos H}$ and $2 = \texttt{ZPos (O H)}$.

The SMT sort **Int** is mapped to `El int`, and the arithmetic operations of SMT to their counterparts in the Lambdapi encoding.

Notation. We will refer to the rewrite rules of Definition 7.2.2 and Definition 7.2.3 as $\rightarrow_{\mathbb{Z}}$ and $\rightarrow_{\mathbb{P}}$ in the following sections.

Lemma 3 (Confluence of $\rightarrow_{\mathbb{Z}}$ and $\rightarrow_{\mathbb{P}}$)**.** *Proof.* CSI [102] automatically proves the confluence of $\rightarrow_{\mathbb{Z}}$ and $\rightarrow_{\mathbb{P}}$. □

7.3 REPRESENTING LINEAR ARITHMETIC (LA)

SMT solvers define the sort **`Real`**, but in practice the reconstruction of Alethe proof with the logic **LA** only requires the rational number ($\mathbb{Q}$). SMT solvers involve Real number in non-linear arithmetic logics such as **NRA**, since rationals alone are insufficient because solutions may require algebraic numbers (roots of polynomials, e.g., $\sqrt{2}$) or even transcendental numbers (e.g., π, e). Our encoding of rationals (Definition 7.3.1) adapts the approach from [34], which employs a balanced ternary numeral system [1] to define integers and rationals. This representation ensures convergence (modulo AC for addition/multiplication), unique normal forms, and efficient reduction—minimizing the number of steps required to reach normal form. However, we replace the ternary encoding with our binary integer representation (Section 7.4). Our encoding of rational numbers is still a work in progress. While core properties, such as normalization and basic arithmetic operations have been formalized, certain lemmas including the associativity of addition and multiplication, are currently accepted and remain to be proven.

Definition 7.3.1 (Rational)**.** We define the type of rational numbers and some operators as follows:

$\mathbb{Q} : \texttt{TYPE}$	$\texttt{frac} : \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Q}$
$\texttt{rat} : \texttt{Set}$	$/ : \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Q}$ (infix)
$\texttt{El rat} \hookrightarrow \mathbb{Q}$	$+_{\mathbb{Q}} : \mathbb{Q} \rightarrow \mathbb{Q} \rightarrow \mathbb{Q}$ (infix)
	$*_{\mathbb{Q}} : \mathbb{Q} \rightarrow \mathbb{Q} \rightarrow \mathbb{Q}$ (infix)

The symbol `frac` denotes any fraction, while $/$ denotes irreducible fractions. Since the SMT-LIB standard (Section 3.4.3) allows denominators to be zero, we use $\mathbb{Z}$ instead of $\mathbb{P}$ but normalize such cases via rewrite rules.

The Mathematical Components library MathComp [103] in Rocq [93], and the mathlib library [33] in Lean [74] encode rational numbers as pairs of integers reduced by their greatest common divisor (GCD), accompanied by a proof certifying their coprimality and the positivity of the denominator. This invariant ensures a canonical representation of each rational number, which simplifies comparisons and arithmetic operations.

1 Balanced ternary is a ternary numeral system (i.e. base 3 with three digits) that uses a balanced signed digit representation of the integers in which the digits have the values -1, 0, and 1.

In contrast, our approach exploits Lambdapi's rewrite rules to define normalization rules that automatically yield a canonical form for rational numbers, obviating the need for explicit proofs of the invariant. Consequently, our construction consists solely of a pair of integers, eliminating the overhead of carrying invariant proof, unlike the MathComp library's design.

A well-known method for reducing a rational number $\frac{x}{y}$ to its normal form is to divide both x and y by their GCD. We avoid this due to two complications:

1. Introducing a division operator (risking division-by-zero errors)
2. Implementing GCD computation separately.

Instead, we introduce `rgcd` in Definition 7.3.2, which simultaneously computes the GCD and divides the inputs by exploiting J. Stein's algorithm [58], an efficient division-free method adapted to binary representations.

Definition 7.3.2 (irreducible fractions)**.** The auxiliary operator `stein` implements for positive binary integers ($\mathbb{P}$) the logic to compute the GCD and divide the first parameter (x in `rgcd x y`), while `rgcd` extends it to $\mathbb{Z}$ by handling signs:

$$\texttt{frac}\ x\ y \hookrightarrow ((\texttt{sign}\ y) * \texttt{rgcd}\ x\ y)/((\texttt{sign}\ y) * \texttt{rgcd}\ y\ x)$$

$$\begin{aligned}
&\texttt{stein} : \mathbb{P} \to \mathbb{P} \to \mathbb{P}\\
&\texttt{stein}\ \texttt{H}\ _ \hookrightarrow \texttt{H}\\
&\texttt{stein}\ a\ \texttt{H} \hookrightarrow a\\
&\texttt{stein}\ (\texttt{O}\ a)\ (\texttt{O}\ b) \hookrightarrow \texttt{stein}\ a\ b\\
&\texttt{stein}\ (\texttt{O}\ a)\ (\texttt{I}\ b) \hookrightarrow \texttt{O}\ (\texttt{stein}\ a\ (\texttt{I}\ b))\\
&\texttt{stein}\ (\texttt{I}\ a)\ (\texttt{O}\ b) \hookrightarrow \texttt{stein}\ (\texttt{I}\ a)\ b\\
&\texttt{stein}\ (\texttt{I}\ a)\ (\texttt{I}\ b)\\
&\quad\hookrightarrow \begin{cases} \texttt{H} & a = b\\ \texttt{stein}\ (\texttt{I}\ a)\ (\texttt{sub}\ (\texttt{I}\ b)\ (\texttt{I}\ a)) & a < b\\ \texttt{stein}\ (\texttt{I}\ a)\ (\texttt{sub}\ (\texttt{I}\ a)\ (\texttt{I}\ b)) & a > b \end{cases}
\end{aligned}$$

$$\begin{aligned}
&\texttt{rgcd} : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}\\
&\texttt{rgcd}\ \texttt{Z0}\ _ \hookrightarrow \texttt{Z0}\\
&\texttt{rgcd}\ a\ \texttt{Z0} \hookrightarrow \texttt{sign}\ a\\
&\texttt{rgcd}\ (\texttt{Zpos}\ a)\ (\texttt{Zpos}\ b) \hookrightarrow \texttt{Zpos}\ (\texttt{stein}\ a\ b)\\
&\texttt{rgcd}\ (\texttt{Zpos}\ a)\ (\texttt{Zneg}\ b) \hookrightarrow \texttt{Zpos}\ (\texttt{stein}\ a\ b)\\
&\texttt{rgcd}\ (\texttt{Zneg}\ a)\ (\texttt{Zpos}\ b) \hookrightarrow \texttt{Zneg}\ (\texttt{stein}\ a\ b)\\
&\texttt{rgcd}\ (\texttt{Zneg}\ a)\ (\texttt{Zneg}\ b) \hookrightarrow \texttt{Zneg}\ (\texttt{stein}\ a\ b)
\end{aligned}$$

Here, `sign` returns the sign of the argument in $\{-1, 1\}$.

Definition 7.3.3 (Normalization rules)**.** The following rewrite rules normalise fractions written with `/`. To account for division by zero (allowed by SMT solvers), rule Equation (17) rewrites such terms to the

canonical form $\frac{0}{1}$. Rules Equation (18)–Equation (21) reduce denominators to 1 whenever the numerator is 0, while rule Equation (22) and Equation (23) ensures that the denominator is always positive by moving the negative sign to the numerator.

$$_/0 \hookrightarrow 0/1 \tag{17}$$
$$0/\,\texttt{ZPos}\ (\texttt{I}\ _) \hookrightarrow 0/1 \tag{18}$$
$$0/\,\texttt{ZPos}\ (\texttt{O}\ _) \hookrightarrow 0/1 \tag{19}$$
$$0/\,\texttt{ZNeg}\ (\texttt{I}\ _) \hookrightarrow 0/1 \tag{20}$$
$$0/\,\texttt{ZNeg}\ (\texttt{O}\ _) \hookrightarrow 0/1 \tag{21}$$
$$(\texttt{ZPos}\ x)/(\texttt{ZNeg}\ y) \hookrightarrow (\texttt{ZNeg}\ x)/(\texttt{ZPos}\ y) \tag{22}$$
$$(\texttt{ZNeg}\ x)/(\texttt{ZNeg}\ y) \hookrightarrow (\texttt{ZPos}\ x)/(\texttt{ZPos}\ y) \tag{23}$$

These rules ensure that algebraic operators on rational numbers are total functions, handling all edge cases (e.g., division by zero) by mapping them to the canonical form $\frac{0}{1}$.

We can then define the rational operators as usual:

Definition 7.3.4 (operator fractions)**.**

$$x/y +_{\mathbb{Q}} u/v \hookrightarrow \texttt{frac}\ ((x * v) + (u * y))\ (y * v) \tag{24}$$
$$x/y *_{\mathbb{Q}} u/v \hookrightarrow \texttt{frac}\ (x * u)\ (y * v) \tag{25}$$

7.4 SMT-LIB HELPER OPERATORS

SMT-LIB provides the operators:

- `(par (A) (distinct A A Bool :pairwise))`
- `(par (A) (ite Bool A A A))`

The **`distinct`** denotes the function that returns true if its two arguments are not identical as defined in Table 4, and `ite` denotes the function that returns its second argument or its third depending on whether its first argument is true or not.

Definition 7.4.1 (Distinctness predicate)**.** We encode the distinct parameter of SMT by introducing a predicate over dependent vectors Definition 2.6.6, since both the size of the collection and the type of its elements influence how the predicate is computed.

$$\texttt{distinct}\ [a : \mathit{Set}][n : \mathbb{N}] : \texttt{Vec}\ a\ n \to \texttt{Prop}$$

Definition 7.4.2 (Distinctness eliminator)**.** To implement the behavior specified in Table 4, we formalize the elimination of distinct through a series of rewriting rules.

$$\text{distinct}\ _\ \square_{\mathbb{V}} \hookrightarrow \bot$$
$$\text{distinct}\ _\ (x ::_{\mathbb{V}} \square_{\mathbb{V}}) \hookrightarrow \top$$
$$\text{distinct}\ _\ (x ::_{\mathbb{V}} y ::_{\mathbb{V}} \square_{\mathbb{V}}) \hookrightarrow \neg(x = y)$$
$$\text{distinct}\ o\ (x ::_{\mathbb{V}} y ::_{\mathbb{V}} z ::_{\mathbb{V}} tl)))) \hookrightarrow \bot$$
$$\text{distinct}\ _\ (x ::_{\mathbb{V}} y ::_{\mathbb{V}} tl) \hookrightarrow \neg(x = y) \wedge \texttt{distinct}\ tl$$

Table 4: Specification of the `distinct_elim` rule in Alethe.

Rule 93: R = `distinct_elim`

This rule eliminates the `distinct` predicate.

When arguments are type **Bool**:

$$i.\ \Gamma \rhd\ (\text{distinct}\ \varphi\ \psi) \approx \neg(\varphi \approx \psi) \qquad (\mathsf{R})\,[\,]$$

$$i.\ \Gamma \rhd\ (\text{distinct}\ \varphi_1\ \varphi_2\ \varphi_3\ \ldots) \approx \bot \qquad (\mathsf{R})\,[\,]$$

For all other sorts the rule is

$$i.\ \Gamma \rhd\ (\text{distinct}\ t_1\ \ldots\ t_n) \approx \bigwedge_{i=1}^{n} \bigwedge_{j=i+1}^{n} t_i \neq t_j \qquad (\mathsf{R})\,[\,]$$

The first two rules capture the base cases: a distinct check on an empty vector returns $\bot$, while a vector containing a single element trivially returns $\top$. The third rule handles the case of two elements, reducing to the negation of their equality. For longer vectors, the fourth rule reflects the immediate elimination to $\bot$ when parameter are type `Prop` (**`Bool`**), while the final rule encodes the recursive elimination for others types than `Prop`. It process by requiring that the head element differs from the second and that the distinct property holds for the remaining tail.

Example 18. distinct $(\bot ::_{\mathbb{V}} \top ::_{\mathbb{V}} \top ::_{\mathbb{V}} \square_{\mathbb{V}}) \hookrightarrow \bot$ and distinct $(\top ::_{\mathbb{V}} \bot ::_{\mathbb{V}} \square_{\mathbb{V}}) \hookrightarrow \neg(\top = \bot)$.

Definition 7.4.3 (`ite`)**.** We define a conditional expression:

$$\texttt{ite} : \Pi[a : \texttt{Set}],\ \texttt{Prop} \to \texttt{El}\ a \to \texttt{El}\ a \to \texttt{El}\ a$$

This conditional expression has the particularity that its condition is of type `Prop` and not in $\mathbb{B}$. We also define the introduction principle:

$$\begin{aligned}&\text{ite_ind} : \Pi[a],\ \Pi(c : \texttt{El}\ o)\,(t\ e : \texttt{El}\ a)\,(p : \texttt{El}\ a \to \text{Prop})\\&\quad(\texttt{Prf}\ c \to \texttt{Prf}(p\ t)) \to\\&\quad((\texttt{Prf}\ c \to \texttt{Prf}\ \bot) \to \texttt{Prf}(p\ e)) \to\\&\quad\texttt{Prf}(p\ (\texttt{ite}\ c\ t\ e)))\end{aligned}$$

7.5 ENCODING CLAUSES

Alethe distinguishes between clauses that appear in steps, such as $(\text{cl}\ l_1 \ldots l_n)$ in Equation (1), and ordinary disjunction [11, §4]. The syntax for clauses uses the `cl` operator, while disjunction is represented as the standard SMT-LIB `or`. Alethe identifies a clause as a set of literals. As mentioned in Chapter 4, Alethe identifies clauses that differ only in the order of literals. In particular, note that the "literals" appearing in clauses may actually be arbitrary formulas in Alethe. It provides the `or` rule for converting disjunctions into clauses:

$$\begin{aligned}&\text{i.}\quad \rhd\quad \varphi_1 \vee \cdots \vee \varphi_n\quad (\ldots)\\&\text{j.}\quad \rhd\quad (\texttt{cl}\ \varphi_1 \ldots \varphi_n)\quad (\texttt{or}\ \text{i})[\,]\end{aligned}$$

Remark 8 (Lack of Structural Canonical Form in Disjunctive Clauses). By interpreting a clause as a disjunction, a clause `cl` $(l_1\ l_2\ l_3\ l_4)$ will be then translated into $l_1 \vee (l_2 \vee (l_3 \vee l_4))$. Therefore, the concatenation of two clauses (what is happening in the conclusion of `resolution`) need not unify with the translated clause of the step. For example, taking 3 arbitrary steps:

A. $\triangleright$ (cl $x_1\ x_2\ x_3$) (...)[...]

B. $\triangleright$ (cl $\neg x_1\ y_2\ y_3$) (...)[...]

C. $\triangleright$ (cl $x_2\ x_3\ y_2\ y_3$) (`resolution A B`)[(x_1 `true`)]

Interpreting a clause as a disjunction, we obtain a concatenated form such as $((x_2 \vee x_3) \vee (y_2 \vee y_3))$. This aligns with the classical formulation of the `resolution` inference rule, expressed as $X \vee Y$ and $\neg X \vee Z \vdash Y \vee Z$. However, in $\lambda\Pi/\equiv$ we have:

$$((x_2 \vee x_3) \vee (y_2 \vee y_3)) \not\equiv_{\beta\Sigma} (x_2 \vee (x_3 \vee (y_2 \vee y_3)))$$
$$\not\equiv_{\beta\Sigma} (((x_2 \vee x_3) \vee y_2) \vee y_3)$$

even though these forms are semantically identical.

Additionally, the pivot must occur at the head of the clause in order to apply the `resolution` rule. As a result, resolving clauses requires reasoning modulo associativity and commutativity. To address the associativity issue, we propose a canonical representation for clauses. For commutativity, instead of permuting the clauses, we simply provide the index of the pivot literal, allowing us to disregard its position in the clause.

Definition 7.5.1 (Clause encoding)**.** We define the type `Clause` to represent an Alethe clause as a list of propositions, thereby ensuring a canonical form.

```
Clause : TYPE
■ : Clause
∨ : Prop → Clause → Clause
```

The constructor $\vee$ prepends an element to a list, and ■ denotes the empty list.

Definition 7.5.2 (Concatenation of clauses)**.** The operator ++ is used to concatenate lists.

```
++ : Clause → Clause → Clause
■ ++ x ↪ x
(x ∨ y) ++ z ↪ x ∨ (y ++ z)
```

Definition 7.5.3 (Proof on `Clause`)**.** Clauses are logically interpreted as disjunctions via the function `cl2disj`, defined through rewriting

rules. For convenience, we also introduce the predicate $\texttt{Prf}^\bullet$, which asserts the provability of a clause.

$$\texttt{cl2disj} : \texttt{Clause} \to \texttt{Prop}$$
$$\texttt{cl2disj}\ \blacksquare \hookrightarrow \bot$$
$$\texttt{cl2disj}\ (x \vee y) \hookrightarrow x \vee (\texttt{cl2disj}\ y)$$
$$\texttt{Prf}^\bullet(c : \texttt{Clause}) := \texttt{Prf}\,(\texttt{cl2disj}\ c)$$

We can now prove the equivalence between clause and disjunction.

Lemma 4 (Clause elimination)**.** For any $p : \texttt{Prop}$ and $q, r : \texttt{Clause}$, If $\texttt{Prf}^\bullet(p \vee q)$ and $(\texttt{Prf}^\bullet(p \vee \blacksquare) \to \texttt{Prf}^\bullet r)$ and $(\texttt{Prf}^\bullet q \to \texttt{Prf}^\bullet r)$ then $\texttt{Prf}^\bullet r$.

Proof. Since we have the equivalence in hypothesis

$$\texttt{Prf}^\bullet(p \vee q) \equiv_{\beta\Sigma} \texttt{Prf}\,(p \vee \texttt{cl2disj}\ q)$$

we apply the classic disjunction elimination rule. Then we can conclude $\texttt{Prf}\,p \to \texttt{Prf}\,(\texttt{cl2disj}\,r)$ wih the hypothesis $\texttt{Prf}^\bullet(p \vee \blacksquare) \to \texttt{Prf}^\bullet r$, and we can conclude $\texttt{Prf}\,(\texttt{cl2disj}\,q) \to \texttt{Prf}\,(\texttt{cl2disj}\,r)$ with the hypothesis $\texttt{Prf}^\bullet q \to \texttt{Prf}^\bullet r$. □

Lemma 5 (Concatenation is disjunction)**.** For any two clauses $a\ b$, the equivalence $\texttt{Prf}\,(\texttt{cl2disj}(a \mathbin{+\!\!+} b)) \iff \texttt{Prf}\,(\texttt{cl2disj}\,a \vee \texttt{cl2disj}\,b)$ holds.

Proof. By induction on a.

- For the base case $a = \blacksquare$ we must show that

  $$\texttt{Prf}\,(\texttt{cl2disj}(\blacksquare \mathbin{+\!\!+} b)) \iff \texttt{Prf}\,(\texttt{cl2disj}\,\blacksquare \vee \texttt{cl2disj}\,b)$$

  The implication "$\Rightarrow$" is trivial because $\blacksquare \mathbin{+\!\!+} b$ rewrites to b. The implication "$\Leftarrow$" is proved using the standard elimination rule $\vee_e$ for disjunction, which requires proving

  – $\texttt{Prf}\,(\texttt{cl2disj}\,\blacksquare) \to \texttt{Prf}\,(\texttt{cl2disj}\,(\blacksquare \mathbin{+\!\!+} b))$

  – as well as $\texttt{Prf}\,(\texttt{cl2disj}\,b) \to \texttt{Prf}\,(\texttt{cl2disj}\,(\blacksquare \mathbin{+\!\!+} b))$.

  Both proofs are immediate since $\texttt{cl2disj}\,\blacksquare$ rewrites to $\bot$ and $\blacksquare \mathbin{+\!\!+} b$ rewrites to b.

- For the case $a = h \vee tl$ we may use the induction hypothesis $\texttt{Prf}\,(\texttt{cl2disj}\,(tl \mathbin{+\!\!+} b)) \iff \texttt{Prf}\,(\texttt{cl2disj}\,tl \vee \texttt{cl2disj}\,b)$. The conclusion is then easy to prove based on the rewriting rules for the operators $\texttt{cl2disj}$ and $+\!\!+$, together with associativity of $\vee$.

□

8 RECONSTRUCTING SMT PROOFS

We now describe how we encode input problems expressed in a given SMT-LIB signature [18, §5.2.1]. In order to avoid a notational clash with the Lambdapi signature Σ, we denote the set of SMT-LIB sorts as $\Theta^{\mathcal{S}}$, the set of function symbols $\Theta^{\mathcal{F}}$, the set of natural numbers $\Theta^{\mathcal{N}}$, and the set of variables $\Theta^{\mathcal{X}}$. Alethe does not support the sorts `Array` and `String`. Our translation is based on the following functions:

- $\mathcal{D}$ translates declarations of SMT sorts and functions in $\Theta^{\mathcal{S}}$ and $\Theta^{\mathcal{F}}$ into declarations of constants,
- $\mathcal{S}$ maps SMT sorts to Σ types,
- $\mathcal{E}$ translates SMT expression to $\lambda\Pi/\equiv$ terms,
- $\mathcal{C}$ translates a list of commands $c_1 \dots c_n$ of the form $i.\ \Gamma \rhd\ \varphi\ (\mathcal{R}\ P)[A]$ to typing judgments $\Gamma \vdash_\Sigma i : N$ with N the corresponding type of the clause φ.

Definition 8.0.1 (Function $\mathcal{D}$ translating SMT sort and function symbol declarations)**.** For each sort symbol s with arity n in $\Theta^{\mathcal{S}}$ we create a constant $\mathcal{D}(s) : \mathtt{Set} \to \cdots \to \mathtt{Set}$. For each function symbol $f\ \sigma^+$ in $\Theta^{\mathcal{F}}$ we create a constant $\mathcal{D}(f) : \mathcal{S}(\sigma^+)$.

In other words, all SMT sorts used in the Alethe proof trace will be defined as constants that inhabit the type `Set` in the signature context Σ. For every function declared in the SMT prelude, we define a constant whose arity follows the sort declared in the SMT prelude. The translation of sorts is formally defined as follows.

Definition 8.0.2 (Function $\mathcal{S}$ translating sorts of expression)**.** The definition of $\mathcal{S}(s)$ is as follows.

- Case s = **`Bool`**, then $\mathcal{S}(s)$ = `El o`,
- Case s = **`Int`**, then $\mathcal{S}(s)$ = `El int`,
- Case s = **`Real`**, then $\mathcal{S}(s)$ = `El rational`,
- Case $s = \sigma_1\ \sigma_2 \dots \sigma_n$ then $\mathcal{S}(s)$ = $\mathtt{El}(\mathcal{S}(\sigma_1) \rightsquigarrow \cdots \rightsquigarrow \mathcal{S}(\sigma_n))$,
- otherwise $\mathcal{S}(s)$ = $\mathtt{El}\ \mathcal{D}(s)$.

Example 19. Translation of the prelude in Listing 5.

```
1 symbol U : Set;
2 symbol a : El U;
3 symbol b : El U;
4 symbol p : El (U ⇝ o);
```

Definition 8.0.3 (Function $\mathcal{E}$ translating SMT expressions)**.** The definition of $\mathcal{E}(e)$ is as follows.

- Case $e = c$, with $c \in \Theta^{\mathcal{N}}$ then $\mathcal{E}(e) = c$
- Case $e = (u/d)$, then $\mathcal{E}(e) = \texttt{frac}\ \mathcal{E}(u)\ \mathcal{E}(d)$
- Case $e = \texttt{true}$, then $\mathcal{E}(e) = \top$
- Case $e = \texttt{false}$, then $\mathcal{E}(e) = \bot$
- Case $e = (p\ t_1\ t_2 \ldots\ t_n)$ and p a logic operator, then $\mathcal{E}(e) = \mathcal{E}(t_1)\ p\ \ldots\ p\ \mathcal{E}(t_n)$.
- Case $e = (f\ t_1 \ldots\ t_n)$ with $f \in \Theta^{\mathcal{F}}$,
  then $\mathcal{E}(e) = (\mathcal{D}(f)\ \mathcal{E}(t_1)\ \ldots\ \mathcal{E}(t_n))$.
- Case $e = (\approx\ t_1\ t_2)$ then $\mathcal{E}(e) = (\mathcal{E}(t_1) = \mathcal{E}(t_2))$.
- Case $e = (Q\ x_1 \ldots x_n\ t)$ and $Q \in \{\texttt{forall}, \texttt{exists}\}$
  then $\mathcal{E}(e) = Q\, x_1, \ldots, Q\, x_n, \mathcal{E}(t)$.
- Case $e = (\texttt{choice}\ x : \sigma\ t)$ then $\mathcal{E}(e) = \epsilon\ x : \mathcal{S}(\sigma), \mathcal{E}(t)$.
- Case $e = (x : \sigma)$ with $x \in \Theta^{\mathcal{X}}$ a sorted variable, then $\mathcal{E}(e) = x : \mathcal{S}(\sigma)$.
- Case $e = (\texttt{ite}\ c\ t\ e)$, then $\mathcal{E}(e) = \texttt{ite}\ \mathcal{E}(c)\ \mathcal{E}(t)\ \mathcal{E}(e)$.
- Case $e = (\texttt{xor}\ a\ b)$, then $\mathcal{E}(e) = \texttt{xor}\ \mathcal{E}(a)\ \mathcal{E}(b)$.
- Case $e = (\texttt{distinct}\ t_1 \ldots t_n)$,
  then $\mathcal{E}(e) = \texttt{distinct}\ (\mathcal{E}(t_1) ::_{\mathbb{V}} \ldots ::_{\mathbb{V}}\ \mathcal{E}(b) ::_{\mathbb{V}} \square_{\mathbb{V}})$.

Definition 8.0.4 (Function $\mathcal{C}$ translating step)**.** We define the function $\mathcal{C}$ that translates an Alethe step. Given a step:

$$i.\, \Gamma \rhd l_1 \ldots l_n\ (R\ P)[A]$$

the function $\mathcal{C}(i)$ produces a constant

$$\mathcal{C}(i) = i : \texttt{Prf}^{\bullet}(\mathcal{E}(l_1) \vee \cdots \vee \mathcal{E}(l_n) \vee \blacksquare) \coloneqq M$$

The function $\mathcal{C}$ is defined by cases on the rule R. In the case of an `assume` command, we do not provide a proof term M so i is considered as an axiom. Its proof term may refer to constants representing steps that precede i in the trace and that must therefore have been introduced when Lambdapi checks the definition of constant i.

In the following sections, we present how different categories of rules are translated by $\mathcal{C}$. We will sometimes omit writing $\texttt{Prf}^{\bullet}$ and $\texttt{Prf}$ in order to simplify the notation.

9 RECONSTRUCTING FOL PROOFS

9.1 TAUTOLOGOUS RULES AND SIMPLE DEDUCTION

Many Alethe rules introduce tautologies or derive their conclusion from a single premise. These rules are primarily used for clausification and to simplify Boolean connectives during preprocessing. Figure 12 illustrates the definition of $\mathcal{C}$ for the rules `assume`, `equiv_pos2` and `cong` from the running example of Listing 5. The definitions use the following lemmas proved in our Lambdapi encoding:

- $\texttt{cong}_1 : \Pi(a\ b : \texttt{Set}),\ \Pi f : \texttt{El}\ a \to \texttt{El}\ b,$
  $\Pi x\ x',\ \texttt{Prf}(x = x') \to \texttt{Prf}(f\ x = f\ x'),$
- $\texttt{equiv_pos2} : \Pi(\varphi_1\ \varphi_2 : \texttt{Prop}),\ \texttt{Prf}^{\bullet}(\neg(\varphi_1 = \varphi_2) \vee \neg\varphi_1 \vee \varphi_2 \vee \blacksquare)$

RESOLUTION RULE However, Alethe's `resolution` rule, shown in Figure 13, represents hyper-resolution applied to a set of ground first-order clauses $i_1 \dots i_n$, where the resulting clause $l_{s_1}^{r_1} \dots l_{s_m}^{r_m}$ is obtained by a chain of predicate resolution steps that remove complementary literals from the input clauses, with double negations being removed implicitly. For example, the formulas $\neg\neg\neg P$ and $\neg\neg P$ can serve as pivots during resolution. The first formula is interpreted as $\neg P$ and the second as just P to perform resolution steps. Alethe allows resolution steps without providing the pivots; however, Carcara's elaborated proof incorporates the pivots as arguments in the resolution rule, eliminating the need for an additional intermediate step to search for the pivots in our translation to Lambdapi. In addition, pivots may appear anywhere in a clause rather than just as the head literal.

In our encoding of the `resolution` rule, we will represent clauses as disjunctions of literals constructed from $\mathbb{L}\ o$, using the function `disj`, rather than the type `Clause`. This choice allows us to work within the type `Prop`, where equality is already defined and can be exploited for reasoning (Lemma 9 below). In particular, we need to express that a disjunction remains identical when an element is removed and reinserted, an equivalence that relies on propositional equality.

Definition 9.1.1 (disj)**.** The function `disj` converts a list of proposition formulas into a disjunction.

$$\texttt{disj} : \mathbb{L}\ o \to \texttt{El}\ o$$
$$\texttt{disj}\ (l_1 :: \ell) \hookrightarrow (l_1 \vee \texttt{disj}\ \ell)$$
$$\texttt{disj}\ \blacksquare \hookrightarrow \bot$$

R = assume	
$i.\ \Gamma \rhd \varphi \quad (R)[]$	$i : \mathtt{Prf}^\bullet(\mathcal{E}(\varphi) \vee \blacksquare)$
R = equiv_pos2	
$i.\ \Gamma \rhd \neg(a \approx b), \neg a, b \quad (R)[]$	$i : \mathtt{Prf}^\bullet(\neg(\mathcal{E}(a) = \mathcal{E}(b)) \vee \neg\mathcal{E}(a) \vee \mathcal{E}(a) \vee \blacksquare)$ $:= \mathtt{apply\ equiv_pos2}$
R = cong	
$i_1 \quad \Gamma \rhd t_1 \approx u_1 \quad (\ldots)$ $i_2 \quad \Gamma \rhd t_2 \approx u_2 \quad (\ldots)$ $\vdots$ $i_n. \quad \Gamma \rhd t_n \approx u_n \quad (\ldots)$ $j. \quad \Gamma \rhd (f\ t_1 \ldots t_n) \approx (f\ u_1 \ldots u_n)$ $(R\ i_1\ i_2\ ..\ i_n)[]$	$j : \mathtt{Prf}^\bullet(\mathcal{E}(f\ t_1 \ldots t_n) = \mathcal{E}(f\ u_1 \ldots u_n) \vee \blacksquare)$ $:= \mathtt{apply}\ (\mathtt{cong}_n\ f\ \mathtt{Prf}^\bullet{}_1(i_1) \ldots \mathtt{Prf}^\bullet{}_1(i_n)))$

Figure 12: Translations for three representative Alethe commands.

$$
\begin{array}{lll}
i_1.\ \rhd & l_1^1, \ldots, l_{k^1}^1 & (\ldots) \\
i_n.\ \rhd & l_1^n, \ldots, l_{k^n}^n & (\ldots) \\
 & \vdots & \\
j.\ \rhd & l_{s_1}^{r_1}, \ldots, l_{s_m}^{r_m} & (\mathtt{resolution}\ i_1 \ldots i_n)[]
\end{array}
$$

Figure 13: Resolution rule

To avoid explicit conversions, we rely on Lambdapi's coercion mechanism (detailed in Section 2.7.3) to implicitly translate between `Clause` and $\mathbb{L}\ o$, allowing the two representations to be used interchangeably in proofs. We recall that a clause C is rewritten into disjunctions in $\mathtt{Prf}^\bullet\ C$.

Definition 9.1.2 (Clause⇆List Coercions)**.** The two coercions functions below define mutual conversions between clauses and lists of literals. The function `cl2list` translates a clause into its corresponding list representation, while `list2cl` reconstructs a clause from a list of literals.

$$
\begin{array}{l}
\mathtt{cl2list} : \mathtt{Clause} \to \mathbb{L}\ o \\
\mathtt{cl2list}\ \blacksquare \hookrightarrow \square \\
\mathtt{cl2list}\ (x \vee xs) \hookrightarrow x :: (\mathtt{cl2list}\ xs) \\
\\
\mathtt{list2cl} : \mathbb{L}\ o \to \mathtt{Clause} \\
\mathtt{list2cl}\ \square \hookrightarrow \blacksquare \\
\mathtt{list2cl}\ (x :: xs) \hookrightarrow x \vee (\mathtt{list2cl}\ xs) \\
\\
\mathtt{coerce}\ \mathtt{Clause}\ (\mathbb{L}\ o)\ x \hookrightarrow \mathtt{cl2list}\ x \\
\mathtt{coerce}\ (\mathbb{L}\ o)\ \mathtt{Clause}\ x \hookrightarrow \mathtt{list2cl}\ x
\end{array}
$$

To prove our resolution rule, we first establish a sequence of auxiliary lemmas. Each lemma captures a structural property of disjunc-

tions at the clause level, and together they provide the tools needed for the final resolution argument.

The Lemma 6 lemma below ensures that if a subclause of a clause is provable, then the entire clause is also provable, thus allowing provability to lift from parts to wholes.

Lemma 6 (Sublist disjunction)**.** Let $\mathtt{ps}, \mathtt{qs} : \mathbb{L}\ \mathtt{o}$ and two index $\mathtt{i}, \mathtt{j} \in \mathbb{N}$. If

$$\mathtt{Prf}(\mathtt{disj\ ps}) \quad \text{and} \quad \mathtt{Prf}(\mathtt{ps} = \mathtt{subclause\ qs\ i\ j}),$$

then

$$\mathtt{Prf}(\mathtt{disj\ qs}).$$

Here $(\mathtt{subclause\ qs\ i\ j})$ abbreviates $\mathtt{drop\ i\ (take\ j\ qs)}$ which take the part of the clause $\mathtt{qs}$ between positions $\mathtt{i}$ and $\mathtt{j} - 1$. So the result is simply the slice of $\mathtt{qs}$ starting at index $\mathtt{i}$ and ending right before $\mathtt{j}$.

Proof. The proof proceeds by structural induction on $\mathtt{qs}$.

Case $\mathtt{qs} = \square$, the second premise forces $\mathtt{ps} = \square$; hence the hypothesis $\mathtt{Prf}(\mathtt{disj\ ps}) \equiv_{\beta\Sigma} \mathtt{Prf}(\bot)$, thus the goal follows immediately by *ex falso quodlibet*.

Case $\mathtt{qs} = \mathtt{t} :: \ell$ with the induction hypothesis saying that for every clause $\mathtt{L}$ interpreted as list, and every pair of indices $\mathtt{n}$ and $\mathtt{m}$, if $\mathtt{Prf}(\mathtt{disj\ L})$ and $\mathtt{L}$ is equal to the subclause of ℓ from position $\mathtt{n}$ up to $\mathtt{m}$, then $\mathtt{Prf}(\mathtt{disj}\ \ell)$. Now we must show that:

$$\mathtt{Prf}(\mathtt{disj\ t} :: \ell) \equiv_{\beta\Sigma} \mathtt{Prf}(\mathtt{t} \vee \mathtt{disj}\ \ell).$$

We reason by cases on the index $\mathtt{j}$ and then $\mathtt{i}$.

- For $\mathtt{j} = 0$, we have $\mathtt{take}\ 0\ (\mathtt{t} :: \ell) = \square$, hence the subclause $\mathtt{ps}$ of $\mathtt{qs}$ is empty i.e. $\mathtt{ps} = \square$. Similar to the previous case, the hypothesis $\mathtt{Prf}(\mathtt{disj\ ps}) \equiv_{\beta\Sigma} \mathtt{Prf}(\bot)$, thus the goal follows immediately by *ex falso quodlibet*.
- When $\mathtt{j} > 0$, we then analyze $\mathtt{i}$.
  - If $\mathtt{i} = 0$, then

    $$\mathtt{subclause}(\mathtt{t} :: \ell, 0, \mathtt{j}{+}1) = \mathtt{t} :: \mathtt{subclause}(\ell, 0, \mathtt{j}).$$

    and the equality hypothesis yields

    $$\mathtt{Prf}(\mathtt{ps} = \mathtt{t} :: \mathtt{take\ j}\ \ell) \text{ and so } \mathtt{Prf}(\mathtt{t} \vee \mathtt{disj\ (take\ j}\ \ell))$$

    Thus, there are two subcases to consider. When $\mathtt{Prf}(\mathtt{t})$ holds, we immediately obtain $\mathtt{Prf}(\mathtt{disj\ t} :: \ell)$ by left introduction. In the second case, $\mathtt{Prf}(\mathtt{disj\ (take\ j}\ \ell))$ holds; by the induction hypothesis with the identities that $\mathtt{sublist}\ \ell\ 0\ \mathtt{j} = \mathtt{drop}\ 0\ (\mathtt{take\ j}\ \ell)$, we lift provability to $\mathtt{Prf}(\mathtt{disj}\ \ell)$, and then conclude $\mathtt{Prf}(\mathtt{disj\ t} :: \ell)$ by right introduction.

* If $i > 0$, we peel one element from both `drop` and qs. So the hypothesis specializes to $ps = \texttt{subclause}\ \ell\ i\ j$, and the desired claim follows from the induction hypothesis on ℓ.

In all cases we infer $\texttt{Prf}(\texttt{disj}\ qs)$, as required.

□

The next two lemmas show that provability is stable under appending clauses on either side. These capture the monotonicity of disjunction ($\vee$) with respect to clause concatenation.

Lemma 7 (Right Append Preservation)**.** Let the two lists of propositons $ps, qs : \mathbb{L}\ o$, if $\texttt{Prf}(\texttt{disj}\ qs)$ then also $\texttt{Prf}(\texttt{disj}\ (ps +\!\!+ qs))$.

Proof. The clause qs appears as a subclause of $ps +\!\!+ qs$, starting at position $\texttt{size}(ps)$. By Lemma 6, provability of the disjunction of a sublist lifts to provability of the whole list. Therefore from $\texttt{Prf}(\texttt{disj}\ qs)$ we conclude $\texttt{Prf}(\texttt{disj}\ ps +\!\!+ qs)$.

Concretely, if

$$qs = q_1 \vee q_2 \vee \cdots \vee q_m$$

then

$$(ps +\!\!+ qs) = p_1 \vee \cdots \vee p_n \vee q_1 \vee \cdots \vee q_m.$$

Since $q_1 \vee \cdots \vee q_m$ is provable, the longer disjunction that includes all the p_i as additional disjuncts is also provable. This captures the monotonicity of $\vee$ under extension of the clause. □

Lemma 8 (Left Append Preservation)**.** Let $ps, qs : \mathbb{L}\ o$, If $\texttt{Prf}(\texttt{disj}\ ps)$, then also $\texttt{Prf}(\texttt{disj}\ ps +\!\!+ qs)$.

Proof. The proof follows analogously to that of Lemma 7. □

The final auxiliary lemma explains how a clause can be decomposed into the disjunction of any chosen literal and the clause obtained by removing it.

Lemma 9 (Remove disjunction)**.** Let $qs : \mathbb{L}\ o$ and an index j with the condition that $j < \text{size}(qs)$. Then

$$\texttt{Prf}\Big((qs[j] \vee \texttt{disj}\ (\texttt{remove}\ qs\ j)) = \texttt{disj}\ qs\Big).$$

Here $(\texttt{remove}\ qs\ j)$ denotes the list qs with its j-th literal removed, and $qs[j]$ denotes the j-th literal of qs (with $\bot$ as default if j is out of bounds).

Proof. We proceed by induction on the structure of qs.

Case $qs = \square$. The assumption $j < \text{size}(qs)$ cannot hold, so this case is impossible.

Case $\mathtt{qs} = \mathtt{t} :: \ell$. We must show that:

$$\mathtt{Prf}\big((\mathtt{t} :: \ell)[\mathtt{j}] \vee \mathtt{disj}\ (\mathtt{remove}(\mathtt{t} :: \ell)\ \mathtt{j}) = \mathtt{disj}\ \mathtt{t} :: \ell\big).$$

We distinguish two subcases according to $\mathtt{j}$.

- If $\mathtt{j} = 0$, then $(\mathtt{remove}\ \mathtt{qs}\ 0) = \ell$ and $\mathtt{qs}[0] = \mathtt{t}$. Removing the head element $\mathtt{t}$ and then reintroducing it yields the same disjunction. Hence the equality holds immediately.
- If $\mathtt{j} = \mathtt{j}' + 1$, we need to show that

  $$(\mathtt{t} :: \ell)[\mathtt{j}' + 1] \vee \mathtt{disj}\ (\mathtt{remove}\ (\mathtt{t} :: \ell)\ (\mathtt{j}' + 1)) = \mathtt{disj}\ (\mathtt{t} :: \ell)$$

  First, we have the identities

  $$\mathtt{disj}\ \mathtt{remove}(\mathtt{t} :: \ell)\ (\mathtt{j}' + 1) \equiv_{\beta\Sigma} \mathtt{t} \vee \mathtt{disj}\ (\mathtt{remove}\ \ell\ \mathtt{j}')$$
  $$\text{and } (\mathtt{t} :: \ell)[\mathtt{j}' + 1] \equiv_{\beta\Sigma} \ell[\mathtt{j}']$$

  So it is enough to prove that:

  $$\ell[\mathtt{j}'] \vee \mathtt{t} \vee \mathtt{disj}\ (\mathtt{remove}\ \ell\ \mathtt{j}')) = \mathtt{t} \vee \mathtt{disj}\ \ell$$

  By the induction hypothesis applied to ℓ at index $\mathtt{j}'$, we know that:

  $$\ell[\mathtt{j}'] \vee (\mathtt{disj}\ (\mathtt{remove}\ \ell\ \mathtt{j}')) = \mathtt{disj}\ \ell.$$

  Combining this with the identities above, we obtain

  $$\begin{aligned}
  &\ell[\mathtt{j}'] \vee \mathtt{t} \vee \mathtt{disj}\ (\mathtt{remove}\ \ell\ \mathtt{j}') \\
  &= \mathtt{disj}\ (\mathtt{t} :: \ell) \\
  &= \mathtt{t} \vee \mathtt{disj}\ \ell \\
  &= \mathtt{t} \vee \ell[\mathtt{j}'] \vee \mathtt{disj}\ (\mathtt{remove}\ \ell\ \mathtt{j}') \\
  &= \ell[\mathtt{j}'] \vee \mathtt{t} \vee \mathtt{disj}\ (\mathtt{remove}\ \ell\ \mathtt{j}')
  \end{aligned}$$

  as required.

In all cases we obtain the desired equality □

Together, these results provide the structural tools needed to reason about clause transformations during `resolution`. The proof of the resolution theorem then proceeds by case analysis on the pivot literal, combining these lemmas to show that removing complementary literals from two provable clauses preserves overall provability.

Theorem 3 (Resolution)**.** Let $\mathtt{ps}, \mathtt{qs} : \mathbb{L}\ \mathtt{o}$ such that $\mathtt{Prf}(\mathtt{disj}\ \mathtt{ps})$ and $\mathtt{Prf}(\mathtt{disj}\ \mathtt{qs})$ hold. Suppose $\mathtt{i} < (\mathtt{size}\ \mathtt{ps})$ and $\mathtt{j} < (\mathtt{size}\ \mathtt{qs})$, and assume that the i-th literal of $\mathtt{ps}$ is the negation of the j-th literal of $\mathtt{qs}$. Then the clause obtained by removing these two complementary

literals and concatenating the remaining parts of ps and qs is also provable, that is:

```
resolution [ps, qs : 𝕃 o] (i, j : ℕ)
(hps : Prf(disj ps))
(hqs : Prf(disj qs))
(hi : Prf(i < size ps)
(hj : Prf(j < size qs)
(hij : Prf(ps[i] = ¬qs[j]))
: Prf(disj (remove ps i ++ remove qs j))
```

Proof. We reason by case analysis on the pivot literal truth.

Case 1: `Prf(ps[i])`. Then by the relation `ps[i] = ¬ps[j]`, we immediately obtain `Prf(¬ps[j])`. But under this assumption and by a classical reasoning the `ps[j]` is false.

From Lemma 9 we know that:

```
ps = ps[i] ∨ disj (remove ps i)
qs = qs[j] ∨ disj (remove qs j)
```

Combining this with the fact that `qs[j]` is false, we deduce `Prf(disj (remove qs j))`. Now observe that `disj (remove qs j)` is appended on the right to `disj remove ps i` in the resolvent disjunctions:

```
disj (remove ps i ++ remove qs j).
```

By Lemma 7, whenever the right-hand disjunctions is provable, appending it to any left-hand disjunctions preserves provability. Therefore, applying the lemma with `remove ps i` and `remove qs j`, we conclude

```
Prf(disj (remove ps i ++ remove qs j)).
```

Case 2: `Prf(¬ps[i])`. This case is analogous to the first, but uses Lemma 8 instead.

□

Example 20. Listing 11 illustrates the result of our main function $\mathcal{C}$ applied to the first steps in the running example of Listing 5. In the code below, all the `assume` commands (corresponding to the `asserts` of the input problem) are transformed into constants by $\mathcal{C}$. A symbol without a definition is considered as an axiom. Each step is encoded as an **`opaque symbol`** that represents a lemma. Splitting the proof into multiple lemmas is beneficial for larger proofs because it reduces Lambdapi's checking time for the proof.

```
1  symbol p2 ≔ p b;
2  symbol p4 ≔ p a = p b;
3  symbol p5 ≔ p a;
4  symbol a0 : Prf• (p5 ∨ ■);
5  symbol a1 : Prf• (a = b ∨ ■);
6  symbol a2 : Prf• (¬ p b ∨ ■);
7
8  opaque symbol t1 : Prf• ((¬ ((p_5 = p_2))) ∨ (¬ (p_5)) ∨ p_2 ∨ ■) ≔
9  begin apply equiv_pos2; end;
10
11 opaque symbol t2 : Prf• (p_4 ∨ ■) ≔
12 begin apply ∨ i1; apply cong (p) (headcl a1) end;
13
14 opaque symbol t3 : Prf• (p_2 ∨ ■) ≔
15 begin
16 have t0_t1 : Prf• ((¬ (p_5))  ∨ p_2 ∨ ■) {
17   apply resolution 0 0 t1 t2 ⊤ᵢ ⊤ᵢ (eq_refl _)
18 };
19 have t0_t1_a0 : Prf• (p_2 ∨ ■) {
20     apply resolution 0 0 t0_t1 a0 ⊤ᵢ ⊤ᵢ (eq_refl _)
21 }; refine t0_t1_a0;
22 end;
23
24 opaque symbol t4 : Prf• ■ ≔
25 begin
26   apply resolution 0 0 a2 t3 ⊤ᵢ ⊤ᵢ (eq_refl _);
27 end;
```

Listing 11: Trace from encoded in Lambdapi.

The function `headcl` at line 11 has type $\texttt{Prf}^{\bullet}(a \vee \blacksquare) \to \texttt{Prf}\, a$, and $\top_i : \texttt{Prf}(\top)$ denote the introduction for $\top$.

9.2 SIMPLIFICATION RULES

The Alethe format includes rules that encode operator-level simplifications typically performed by SMT solvers. A key challenge arises from the fact that the solver does not explicitly specify which transformations are applied, nor the order in which they are performed. This makes certain rules, such as `or_simplify`, difficult to certify. For instance, the rule `or_simplify`, presented in Equation (26), simplifies disjunctions by applying equivalence-preserving transformations repeatedly until a fixpoint is reached. Its general form is:

$$i.\quad \Gamma \rhd \quad l_1 \vee \cdots \vee l_n \;\approx\; \psi \quad \texttt{or_simplify} \tag{26}$$

where ψ is the transformed term. The possible transformations are:

(1) $\bot \vee \cdots \vee \bot \Rightarrow \bot$

(2) $l_1 \vee \cdots \vee l_n \Rightarrow l'_1 \vee \cdots \vee l'_m$ where the right-hand side has some $\bot$ literals removed.

(3) $l_1 \vee \cdots \vee l_n \Rightarrow l'_1 \vee \cdots \vee l'_m$ where the right-hand side has some repeated literals removed.

(4) $l_1 \vee \cdots \vee \top \vee \cdots \vee l_n \Rightarrow \top$

(5) $l_1 \vee \cdots \vee l_i \vee \cdots \vee l_j \vee \cdots \vee l_n \Rightarrow \top$ where $l_i = \neg^{2p} x$, $l_j = \neg^{2q+1} x$.

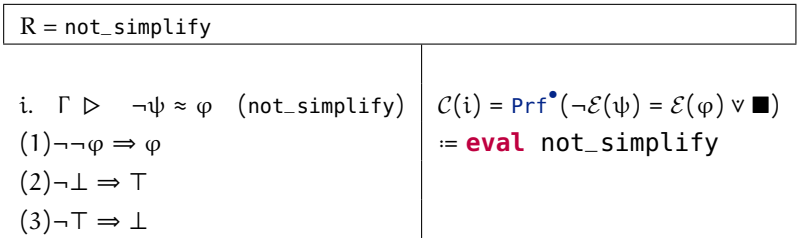


Figure 14: Translating the command `not_simplify`.

To manage these transformations, we leverage the RARE language provided by cvc5 [78], which supports elaborated rewrite proofs. As detailed in Section 5.2, RARE generates explicit proof terms on demand, using built-in rewrite rules. The RARE rule used and its arguments are made explicit in the `:args` parameter of the rule `rare_rewrite` in the proof trace.

However, in some cases, cvc5 may emit proof traces using the legacy simplification rules instead of RARE. In such situations, we reconstruct the simplification steps using tactic scripts, as detailed in Chapter 2. These scripts enable us to approach a fixpoint-style interpretation.

To illustrate this process, we describe how we reconstruct a step involving the `not_simplify` rule, summarized in Figure 14 since it is simpler than `or_simplify` rule. Each transformation associated with this rule is implemented by a separate lemma e.g., `not_simplify1`, `not_simplify2`, etc. which are then tried sequentially on the current goal.

To apply these tactics sequentially, we define a meta-tactic `applyAny` that attempts each tactic in the list until one succeeds. The general concept of meta-tactics is described in Section 2.7.2.

```
all_not_simplify := (#rewrite not_simplify1)::
  (#rewrite not_simplify2) :: (#rewrite not_simplify3)
```

The combinator t_1 `#orelse` t_2 tries t_1 on the current goal; if it fails to make progress, t_2 is applied instead. We then define the tactic for reconstructing the simplification step as follows:

```
applyAny : 𝕃 tactic → Tactic
applyAny □ ↪ #fail
applyAny(t :: ts) ↪ t #orelse (applyAny ts)

not_simplify := #repeat(applyAny all_not_simplify)
  #and #reflexivity
```

Here, `#repeat` repeatedly applies the simplification tactics until none can be applied further. Finally, a step using the `not_simplify` rule can be reconstructed as shown in Figure 14, by simply running the tactic script using the meta-tactic `eval`.

R = `forall_inst`
i. $\Gamma \quad \triangleright\ (\neg \forall \bar{x}, \varphi) \vee (\varphi[\bar{t}]) \quad$ (`forall_inst`)
$i : \texttt{Prf}^{\bullet}(\forall x_1, \ldots\ x_n, \mathcal{E}(\varphi) = \mathcal{E}(\psi) \vee \blacksquare) :=$
begin apply $\vee_{i1}$;
apply imply_to_or;
assume H;
refine $\forall_e$H
end

Figure 15: Translating the `forall_inst` command.

9.3 QUANTIFIERS

The Alethe format defines rules for quantifier instantiation, substitution, and other manipulations of bound variables. For instance, the rule `forall_inst` is used to express quantifier instantiation. It produces a unit clause with a formula of the form $(\neg \forall \bar{x}, \varphi) \vee (\varphi[\bar{t}])$, where φ is a term containing the free variables $\bar{x}$, and each term t is a ground term of the same sort as the corresponding variable x.

Example 21 (Quantifier instantiation)**.** The following example shows a simple Alethe proof expressed in the abstract notation used in this document. It uses quantifier instantiation and resolution to show a contradiction.

a0.	$\triangleright$	$\forall x, (P\ x)$	`(assume)[]`
a1.	$\triangleright$	$\neg(P\ a)$	`(assume)[]`
t1.	$\triangleright$	$\neg(\forall x, P\ x) \vee (P\ a)$	`(forall_inst)[(x,a)]`
t2.	$\triangleright$	$\neg(\forall x, P\ x), (P\ a)$	`(or t1)[]`
t3.	$\triangleright$	$\bot$	`(resolution a0 a1 t2)[]`

As described Section 2.6, we encode universal and existential quantifiers using a shallow embedding into Lambdapi's dependent function type $\Pi A, B$. This allows us to reconstruct the rule `forall_inst` directly, as shown in Figure 15.

9.4 SKOLEMIZATION

The skolemization rules `sko_forall` and `sko_exists` replace the bound variables with `choice` terms instead of fresh symbols. Note that some functions can introduce fresh symbols in the proof trace with the command (`define-fun`). Nevertheless, Carcara eliminates those commands by unfolding the definition during the elaboration process. Moreover, cvc5 preprocesses the input formula by converting existential quantifiers into negated universal quantifiers, thus, the proof traces we are working with only use the `sko_forall` rule.

We now prove a lemma that underlies the translation of the Alethe rule `sko_forall`. We first prove that the universal quantifier can be interpreted by the epsilon operator in our encoding.

R = `sko_forall`
i. $\Gamma, x_1 \mapsto \epsilon\, x_1.\neg\varphi \ldots x_n \mapsto \epsilon\, x_n.\neg\varphi$ $\quad \triangleright \varphi \approx \psi\ (\ldots)$ k. $\Gamma \quad \forall x_1, \ldots x_n.\ \varphi \approx \psi \qquad (R)$
i : $\texttt{Prf}^\bullet(\mathcal{E}(\varphi) = \mathcal{E}(\psi) \vee \blacksquare) \ldots$ k : $\texttt{Prf}^\bullet(\forall x_1, \ldots\ x_n, \mathcal{E}(\varphi) = \mathcal{E}(\psi) \vee \blacksquare) :=$ begin apply $\vee_{i1}$; (apply `sko_forall`; assume x_i H_i; rewrite H_i;)^n reflexivity; end

Figure 16: Translating the `sko_forall` command.

Lemma 10 (Reduction of $\forall$ to choice)**.** For every a : `Set` and predicate p : `El` $a \to$ `Prop`, $(\forall x, p\ x) \Leftrightarrow p\ (\epsilon\, x, \neg(p\ x))$

Proof. **"⇒":** We have to prove that $\forall x, p\, x$ implies $p(\epsilon\, x, \neg(p\, x))$, which follows immediately from the hypothesis.

"⇐": We prove the contrapositive assertion. Assuming $\neg\forall x, p\, x$, we obtain $\exists x, \neg p\, x$. Combining the elimination rule for $\exists$ and the introduction rule ϵ_i, we conclude $\neg p(\epsilon\, x, \neg(p\ x))$.

□

We now prove the Skolemization lemma for the universal quantifier that is used to reconstruct steps using the rule `sko_forall`.

Lemma 11 (skolemize forall)**.** For any a : `Set`, predicate p : `El` $a \to$ `Prop` and q : `Prop`, if for every x : `El` a, `Prf`$(x = (\epsilon\, y, \neg(p\, y)))$ implies `Prf`$(p\, x = q)$, we have that `Prf`$((\forall x, p\, x) = q)$.

Proof. We apply rule `prop_ext` and then prove both directions,

"⇒": We have to prove that $\forall x, p\ x$ implies q. By Lemma 10, the former is equivalent to $p\ (\epsilon\, y,\ \neg(p\ y))$, and together with the assumption of the lemma, we obtain q.

"⇐": Conversely, we have to prove that q implies $\forall x, p\ x$. For x defined as $(\epsilon\ y, \neg(p\ y))$, we have $p\ x = q$ by assumption and thus obtain $p\ (\epsilon\ y, \neg(p\ y))$. By Lemma 10, the latter is equivalent to $\forall x, p\ x$.

□

Figure 16 describes the translation of the `sko_forall` command. We use the notation $(T)^n$ to express that the tactic T will be repeated n times where n is the number of bound variables on the left, and x_i, H_i are fresh new variables in the context. The H_i hypotheses stand for the equalities `Prf`$(x_i = (\epsilon\, y, \neg(p\ y)))$ in Lemma 11.

9.5 ALETHE SUBPROOFS

Alethe uses subproofs to prove lemmas and to create and manipulate the context Γ. To prove lemmas, a subproof can introduce local assumptions. From an assumption φ and a formula ψ proved from φ, the subproof rule deduces the clause $\neg\varphi, \psi$ that discharges the local assumption φ. A subproof step cannot use a premise from a subproof nested within the current subproof. Subproofs are also used to manipulate the context. Alethe contexts are a general mechanism to write substitutions and to change them by attaching new elements. We recall that a context is a possibly empty list $x_1 \dots x_n$ where each element is either a variable or a variable-term tuple denoted $x \mapsto t$.

As shown in the example of Listing 12, the **anchor** command indicates that a subproof will be introduced and it is concluded by a concluding rule such as `subproof`, `bind` or `sko_forall`. Anchors are provided with two annotations. The annotation **:step** provides the name of the step that concludes the subproof whereas the annotation **:args** provides the context as sorted variables and assignments. The example shows a proof that uses a subproof with a context to rename a bound variable. The subproof starts at the **anchor** command at line 1 and ends at line 5 with the *bind* rule that concludes the α-conversion proof of $z2$ to $vr4$. The sub-steps t9.t1 and t9.t2 are carried out in the context $\Gamma = \{z2 \mapsto vr4\}$, hence, all occurrences of $z2$ in the clauses are substituted by $vr4$, allowing in particular step `t9.t1` to succeed using rule `refl`.

```
1 (anchor :step t9 :args ((vr4 U) (:= (z2 U) vr4)))
2 (step t9.t1 (cl (= z2 vr4)) :rule refl)
3 (step t9.t2 (cl (= (p z2) (p vr4))) :rule cong :premises (t9.t1))
4 (step t9 (cl (= (forall ((z2 U)) (p z2))
5                 (forall ((vr4 U)) (p vr4)))) :rule bind)
```

Listing 12: Alethe subproof example.

We define in Figure 17 the definitions of the function $\mathcal{C}$ for the `subproof` and `bind` commands. The translation of `bind` and `subproof` relies on the following lemmas that have been proved in Lambdapi.

Lemma 12 ($\texttt{subproof}_1$)**.** For φ, ψ : `Prop`, if `Prf`φ implies `Prf`ψ then `Prf`$^\bullet(\neg\varphi \vee \psi)$.

Lemma 13 (bind$\forall$)**.** Given an arbitrary a : `Clause` and predicates p, q : `El` $a \to$ `Prop`, if Πx : `El` a, `Prf`$(p\ x = q\ x)$ then `Prf`$((\forall x, p\ x) = (\forall y, q\ y))$.

Lemma 14 (bind$\exists$)**.** Given an arbitrary a : `Clause` and predicates p, q : `El` $a \to$ `Prop`, if Πx : `El` a, `Prf`$(p\ x = q\ x)$ then `Prf`$((\exists x, p\ x) = (\exists y, q\ y))$.

Since proof by reflection is a key methodological ingredient of our reconstruction framework, we dedicate the next section to introduce this concept in detail before applying it to specific rules.

R = `subproof`
$i_1.\ \Gamma \rhd \varphi_1 \quad$ (`assume`) $\vdots$ $i_n.\ \Gamma \rhd \varphi_n \quad$ (`assume`) $\vdots$ $i_j.\ \Gamma \rhd \psi \quad (\ldots) \quad$ with $u \in 1 \ldots n$ $k.\ \Gamma \quad \rhd \neg\varphi_1, \ldots, \neg\varphi_n, \psi \quad$ (`subproof`)
$\mathcal{C}(k) = \texttt{Prf}^\bullet((\neg\mathcal{E}(\varphi_1) \vee \cdots \vee \neg\mathcal{E}(\varphi_n) \vee \mathcal{E}(\psi)) \vee \blacksquare)$ $\coloneqq$ `apply subproof`$_n$
R = `bind`
$j.\ \Gamma\ \overline{y}, \overline{x} \mapsto \overline{y} \rhd \varphi \approx \psi \quad (\ldots)$ with $\overline{a} = a_1 \ldots a_n$ and $Q \in \{\forall, \exists\}$ $k.\ \Gamma \rhd (Q\ \overline{x}, \varphi) \approx (Q\ \overline{y}, \psi) \quad$ (`bind`)
$\mathcal{C}(k) = \texttt{Prf}^\bullet(\mathcal{E}(Q\ \overline{x}, \varphi \approx Q\ \overline{y}, \psi) \vee \blacksquare) \coloneqq$ begin `apply` $\vee_{i1}$`;apply bindQ;` `assume` x_j`;apply (`$\texttt{Prf}^\bullet{}_l$ `j);` end

Figure 17: Translating `subproof` and `bind` commands.

9.6 PROOF BY REFLECTION

Proof by reflection [27] is a technique for constructing certified procedures for automated reasoning by reducing the verification of a logical statement to a symbolic computation. This method is particularly applicable to two broad classes of problems. In the first class, we consider a predicate $P : T \to \texttt{Prop}$, where T is a data type, together with a semi-decision procedure $f : T \to \mathbb{B}$ such that, whenever f returns `true` on input x, the predicate $P(x)$ holds. Formally, this relationship is captured by the following correctness theorem:

$$\texttt{f_correct} : \Pi x : T, \texttt{Prf}(f\ x = \texttt{true}) \to \texttt{Prf}(P\ x).$$

Hence, to establish $P\ y$ for a particular value y, it suffices to verify that $f\ y$ reduces to `true`. In that case, we can simply apply the lemma `f_correct` to y along with a proof that `true` = `true` i.e. $(\texttt{eq_refl}\ \texttt{true})$ with $\texttt{eq_refl} : \Pi[a : \texttt{Set}], \Pi(x : \texttt{El}\ a), \texttt{Prf}(x = x)$. The resulting proof of $P\ y$ is thus given by:

$$\texttt{f_correct}\ y\ (\texttt{eq_refl}\ \texttt{true}) : \texttt{Prf}\ (P\ y)$$

As an illustration, consider the property of a natural number being even. In Lambdapi, this can be expressed using a computable boolean function $\texttt{even} : \mathbb{N} \to \mathbb{B}$ that checks whether a given number is even.

Definition 9.6.1 (Even decidable predicate)**.**

```
even : ℕ → 𝔹
even 0 ↪ true
even 1 ↪ false
even (x +1 +1) ↪ even x
```

The connection between the boolean value and the logical property is established through a correctness lemma stating that if `even n = true`, then n is even. Thus, to prove that a specific number, such as 4, is even, one can reduce the goal to the boolean check `even 4 = true`, which is decidable by computation. Applying the correctness lemma then lifts this computational fact back into the logical framework to conclude that 4 is even. This approach automates routine reasoning steps involving decidable predicates, allowing for concise and efficient proofs. This first class of proofs by reflection is particularly suited for properties such as arithmetic equalities, inequalities, and simple inductive predicates.

The size of such a proof only depends on the size of a particular argument y and does not depend on the number of implicit $\equiv_{\beta\Sigma}$ steps: explicit rewriting steps have been replaced by implicit $\equiv_{\beta\Sigma}$ reductions. The efficiency of this technique of course strongly depends on the efficiency of the system to reduce the application of the decision procedure $f(y)$, hence on the efficiency of the decision procedure (f) itself.

Moreover, while the proof term of the correctness lemma for f may be large, it needs to be constructed only once. This proof can be shared across all instantiations and does not require rechecking by the type-checker in subsequent uses.

The second class of problems where computation can help is the class of algebraic proofs such as proofs relying on rewriting modulo the associativity or the commutativity of some operators e.g. disjunction ($\vee$) or conjunction ($\wedge$). For these proofs, we again consider a type B : `Set` and we exhibit an "abstract type" A : `Set`, with two functions `denote`: `El` $A \to$ `El` B and g : `El` $A \to$ `El` A. The function `denote` is an interpretation function that we can use to associate terms in the concrete type B with abstract terms of type A. The function g reasons on the abstract terms. The reflection process relies on a theorem that expresses that the function g does not change the value of the interpreted term:

```
g_ident : Π x : A, Prf(denote (g x) = denote x)
```

Thus, to prove that two terms t_1 and t_2 are equal in B , we only need to show that they are the images of two terms a_1 and a_2 in A such that $g\,a_1 = g\,a_2$. As an illustration of this second class of proofs by reflection, consider the commutativity of disjunction: for any two propositions P and Q, we wish to prove that $P \vee Q$ is logically equivalent to $Q \vee P$. To apply reflection, we first define an *abstract* syntax for formulas, for instance, a type Monoid with constructors for

Table 5: N-ary operators rules

Rule	Description
and (i)[k]	i. $\triangleright$ $(\varphi_1 \wedge \cdots \wedge \varphi_n)$ j. $\triangleright$ φ_k with $1 \leq k \leq n$
not_and (i)	i. $\triangleright$ $\neg(\varphi_1 \wedge \cdots \wedge \varphi_n)$ j. $\triangleright$ $\neg\varphi_1, \ldots, \neg\varphi_n$
not_or (i)[k]	i. $\triangleright$ $\neg(\varphi_1 \vee \cdots \vee \varphi_n)$ j. $\triangleright$ $\neg\varphi_k$ with $1 \leq k \leq n$
and_pos [k]	$\triangleright$ $\neg(\varphi_1 \wedge \cdots \wedge \varphi_n), \varphi_k$ with $1 \leq k \leq n$
and_neg	$\triangleright$ $(\varphi_1 \wedge \cdots \wedge \varphi_n), \neg\varphi_1, \ldots, \neg\varphi_n$
or_pos	$\triangleright$ $\neg(\varphi_1 \vee \cdots \vee \varphi_n), \varphi_1, \ldots, \varphi_n$
or_neg [k]	$\triangleright$ $\neg(\varphi_1 \vee \cdots \vee \varphi_n), \neg\varphi_k$ with $1 \leq k \leq n$

atomic propositions and for disjunction. We then define an *interpretation function* denote: Monoid → `Prop` that maps abstract formulas to their logical meaning. Next, we define a normalization function $\mathrm{norm} : \text{Monoid} \to \text{Monoid}$ that rewrites disjunctions into a canonical order, such as ordering operands lexicographically. The key theorem states that this normalization preserves meaning:

$$\text{norm_mono} : \forall x : \text{Monoid}, \text{denote}(\mathrm{norm}\, x) = \text{denote}(x).$$

Hence to prove that $P \vee Q$ and $Q \vee P$, are logically equivalent, it suffices to represent them as abstract expressions m_1 and m_2 of type Monoid, apply the function norm with the lemma norm_mono to normalize both, and check that $\mathrm{norm}(m_1) = \mathrm{norm}(m_2)$.

In this thesis, we make use of both classes of reflection to encode certain `Alethe` rules. For example, the reconstruction of arithmetic steps (**LIA** and **LRA**) Alethe rules primarily relies on the first class, while the permutation of clauses for the `resolution` rule relies on the second class.

We will make concrete use of both classes of proof by reflection in the subsequent sections. In particular, Section 9.7 and Section 9.8 (N-ary rules and `evaluate` cvc5 rule) employs proof by reflection of the first class, while Section 9.9 (`contraction` rule) illustrates the use of the second class.

9.7 N-ARY RULES

The Alethe format includes n-ary tautological rules—relying on arbitrary n-ary conjunctions and disjunctions as outlined in Table 5. A direct approach to reconstruct these rules would involve generating tactic scripts during the translation in Carcara. However, this could

lead to proofs requiring hundreds of tactic applications. To address this, we adopt an alternative strategy: reconstructing these rules via a *proof by reflection* of the first kind. We illustrate our approach by detailing the reconstruction of `and_pos`. In our encoding, n-ary conjunctions and disjunctions are interpreted as values of the abstract type list of proposition: $\mathbb{L}\,o$. We begin by introducing the following configuration:

- The function `indexes` : $\Pi[a : \texttt{Set}], \mathbb{L}\ a \to \mathbb{L}\ \texttt{nat}$ takes a list of propositions and returns a list where each proposition is replaced by its position in the list.

  Example 22 (indexes). $\texttt{indexes}(x :: y :: x :: z :: \square) \equiv_{\beta\Sigma} 0 :: 1 :: 2 :: 3 :: \square$

  The function `indexes` n will be used to represent a range $0..n$.

- The function $\in_k : \mathbb{N} \to \mathbb{L}\ \texttt{nat} \to \mathbb{B}$ determines whether a given index is present in a list of indexes ($\mathbb{L}$ `nat`).

- The denotation function `conj` : $\mathbb{L}\ o \to$ `Prop` decodes a list of propositions into an n-ary conjunction:

$$\begin{aligned}&\texttt{conj}\ (x :: b :: \ell) \hookrightarrow (x \wedge \texttt{conj}\ (b :: \ell))\\&\texttt{conj}\ (x :: \square) \hookrightarrow x\\&\texttt{conj}\ \square \hookrightarrow \top\end{aligned}$$

- The total function `literal` : $\Pi(k : \mathbb{N}), \mathbb{L}\ o \to$ `Prop` returns the k-th element of the list, or $\bot$ otherwise.

The rule `and_pos` is an instance of the classical reasoning $\texttt{Prf}^\bullet(\neg A \vee B) \equiv \texttt{Prf} A \to \texttt{Prf}^\bullet B$, with A is $\varphi_1 \wedge \cdots \wedge \varphi_n$, and B is the singleton clause φ_k. Hence, after reification, it suffices to prove `conj` *cnf* implies `literal` *cnf* k. Given the hypothesis that $\texttt{Prf}\,(k \in_k \texttt{indexes}\ \textit{cnf})$, the k-th member is indeed one literal in the conjunction, so from $\texttt{Prf}\,(\texttt{conj}\ \textit{cnf})$ we may project $\texttt{Prf}\,(\texttt{literal}\ \textit{cnf}\ k)$. In other words, if all formulas in `conj` *cnf* hold, then the k-th one holds.

Lemma 15 (and_pos). Given an index $k : \mathbb{N}$ and a list $\textit{cnf} : \mathbb{L}\ o$ representing a reified n-ary conjunction, if

$$\texttt{Prf}(k \in_k (\texttt{indexes}\ \textit{cnf}))$$

i.e. the literal at index k is in the conjunction then

$$\texttt{Prf}^\bullet(\neg(\texttt{conj}\ \textit{cnf}) \vee (\texttt{literal}\ k\ \textit{cnf}) \vee \blacksquare)$$

Proof. By interpreting clause as disjunction it suffices to prove:

$$\neg(\texttt{conj}\ \textit{cnf}) \vee (\texttt{literal}\ \textit{cnf}\ k).$$

We proceed by outer induction on k, and inner induction on the list *cnf*.

R = pos_and
i. ▷ $\neg(\varphi_1 \wedge \cdots \wedge \varphi_n)$, φ_k (pos_and)[k]
i : $\mathtt{Prf}^{\bullet}(\neg(\mathcal{E}(\varphi_1) \wedge \cdots \wedge \mathcal{E}(\varphi_n)) \vee \mathcal{E}(\varphi_k) \vee \blacksquare)$ begin apply $\vee_{i1}$; apply and_pos $\mathcal{E}(k)$ $(\mathcal{E}(\varphi_1) :: \cdots :: \mathcal{E}(\varphi_n) :: \square)$ $\top_i$ end

Figure 18: Translating the pos_and command.

- (Case $k = 0$),
  - (Case *cnf* $= \square$), so the hypothesis $0 \in_k$ (indexes $\square$) leads to a contradiction since the reified conjunction is empty.
  - (Case *cnf* $= a :: \ell$), then the element k is the head literal a so we can immediately conclude.
- (Case $k > 0$),
  - (Case *cnf* $= \square$), so the hypothesis $k + 1 \in_k$ (indexes $\square$) leads to a contradiction since the reified conjunction is empty.
  - (Case *cnf* $= a :: \ell$), we have to show that:

    $$\mathtt{literal}\ (a :: \ell)\ (k + 1)$$

    which is equivalent to prove literal ℓ k. By the induction reasoning, using that $k + 1 \in_k$ indexes $(a :: \ell)$ implies $k \in_k$ indexes ℓ, we obtain a proof of the literal at position k in ℓ. Thus, the conclusion follows directly from this argument.

□

We can reconstruct the proof such as described in Figure 18. We first need to retrieve a disjunction from a clause by applying the disjunction introduction rule $\vee_{i1}$, and then we apply Lemma 15 with the index k given by Alethe, the conjunction reified $(\mathcal{E}(\varphi_1) :: \cdots :: \mathcal{E}(\varphi_n) :: \square)$ and a proof that $\mathtt{Prf}(k \in_k (\mathtt{indexes}\ c))$ which should reduce to $\top$.

9.8 THE evaluate CVC5 RULE

The SMT solver cvc5 can produce proof traces with steps generated by internal theories: evaluate, TRUST_THEORY_REWRITE (TTR). The clauses derived using evaluate are propositional and numerical constant equalities proved internally by cvc5, using constant propagation. The solver will then use the special rule "hole", a placeholder for proof steps that cannot be expressed with Alethe rules. The Carcara checker accepts the conclusion of these hole steps without further justification. In contrast, we attempt to reconstruct them using a combination of simple internal solvers and external tools.

```
1 (step t1 (cl (= (not true) false)) :rule hole :args ("evaluate"))
2 (step t2 (cl (= (>= 0.0 -1.0) true)) :rule hole :args ("evaluate"))
```

Listing 13: An example of proof trace using the cvc5 `evaluate` rule.

For propositional formulas such as line 1 in Listing 13, we reconstruct the proof using an embedded proof procedure based on proof by reflection of the first kind. For equalities over numerical constants, such as line 2, we simplify the expressions using the rewriting rules defined in $\mathcal{R}_{\mathbb{Z}}$ and $\mathcal{R}_{\mathbb{P}}$, and aim to conclude by **`reflexivity`**. The rule TRUST_THEORY_REWRITE or `TTR` for short describes steps performed by cvc5's internal theory rewriter. These steps may involve rewriting quantified propositions or other expressions too complex for the tautology solver. Instead, we attempt reconstruct them by invoking the first-order automated theorem prover Zenon Modulo [44] with the tactic **`why3`** of Lambdapi, configured to call Zenon Modulo. This prover produces proof certificates in both Lambdapi and Dedukti formats, which are currently not integrated, though this step could potentially be automated. Although Zenon modulo could in principle be used more extensively during the certification of Alethe proof traces, in order to control the complexity of proof reconstruction we restrict the usage of an external solver to the `TTR` command.

We now describe the propositional tautological solver based on *proof by reflection* of first kind. We start by defining the target theory that represents formulas of propositional logic.

Definition 9.8.1 ($\mathcal{P}$)**.** This is represented by the inductive type $\mathcal{P}$, with seven constructors listed in Definition 9.8.1.

$$
\begin{aligned}
&\mathcal{P} : \mathtt{TYPE}\\
&\mid \mathbf{pTrue} : \mathcal{P}\\
&\mid \mathbf{pAnd} : \mathcal{P} \to \mathcal{P} \to \mathcal{P}\\
&\mid \mathbf{pFalse} : \mathcal{P}\\
&\mid \mathbf{pOther} : \mathbb{N} \to \mathcal{P}\\
&\mid \mathbf{pOr} : \mathcal{P} \to \mathcal{P} \to \mathcal{P}\\
&\mid \mathbf{pImpl} : \mathcal{P} \to \mathcal{P} \to \mathcal{P}\\
&\mathtt{propS} : \mathtt{Set}\\
&\mathtt{El}\ \mathtt{propS} \hookrightarrow \mathcal{P}
\end{aligned}
$$

We define the constant `propS` : `Set` to be able to quantify over $\mathcal{P}$. Each `pOther` term carries an index in $\mathbb{N}$, which serves as a pointer into a context map $\sigma : \mathbb{L}\ o$.

A context σ stores the corresponding uninterpreted expressions of type `Prop`. Although the automation does not inspect the contents of σ, it can determine whether two `pOther` terms reference the same index and are thus syntactically equal. Note that during reification, repeated expressions are assigned the same index in σ whenever possible. However, we do not attempt to prove that different indices refer to distinct expressions.

Definition 9.8.2 (Reification and denotation $\mathcal{P} \rightleftarrows \texttt{Prop}$)**.** We define a reification function $\Uparrow _ : \mathbb{L}\ o \to \texttt{Prop} \to \mathcal{P}$:

$$
\begin{aligned}
&\Uparrow : \mathbb{L}\ o \to \texttt{Prop} \to \texttt{El}(\texttt{propS} \times \texttt{list}\ o) \\
&\Uparrow \sigma\ \top \hookrightarrow (\texttt{pTrue}, \sigma) \\
&\Uparrow \sigma\ \bot \hookrightarrow (\texttt{pFalse}, \sigma) \\
&\Uparrow \sigma\ (x \wedge y) \hookrightarrow \text{let } (p_1, \sigma_1) = \Uparrow \sigma\ x \text{ in} \\
&\quad \text{let } (p_2, \sigma_2) = \Uparrow \sigma_1\ y \text{ in} \\
&\quad (\texttt{pAnd}\ p_1\ p_2,\ \sigma_2) \\
&\Uparrow \sigma\ (x \vee y) \hookrightarrow \text{let } (p_1, \sigma_1) = \Uparrow \sigma\ x \text{ in} \\
&\quad \text{let } (p_2, \sigma_2) = \Uparrow \sigma_1\ y \text{ in} \\
&\quad (\texttt{pOr}\ p_1\ p_2,\ \sigma_2) \\
&\Uparrow \sigma\ (\neg x) \hookrightarrow \text{let } (p, \sigma') = \Uparrow \sigma\ x \text{ in } (\texttt{pNot}\ p,\ \sigma') \\
&\Uparrow \sigma\ (x \Rightarrow y) \hookrightarrow \text{let } (p_1, \sigma_1) = \Uparrow \sigma\ x \text{ in} \\
&\quad \text{let } (p_2, \sigma_2) = \Uparrow \sigma_1\ y \text{ in} \\
&\quad (\texttt{pImpl}\ p_1\ p_2,\ \sigma_2) \\
&\Uparrow \sigma\ (x \Leftrightarrow y) \hookrightarrow \text{let } (p_1, \sigma_1) = \Uparrow \sigma\ x \text{ in} \\
&\quad \text{let } (p_2, \sigma_2) = \Uparrow \sigma_1\ y \text{ in} \\
&\quad (\texttt{pIff}\ p_1\ p_2,\ \sigma_2) \\
&\Uparrow \sigma\ x \hookrightarrow \\
&\quad \begin{cases} (\texttt{pOther}\ i, \sigma) & \text{if index } i \text{ already exists in } \sigma \\ (\texttt{pOther}\ i, (\sigma\ +\!\!+\ (x :: \square))) & \text{otherwise } i = \texttt{length}(\sigma) + 1 \end{cases}
\end{aligned}
$$

and an interpretation function $\Downarrow _ : \mathbb{L}\ o \to \mathcal{P} \to \texttt{Prop}$:

$$
\begin{aligned}
&\Downarrow : \mathbb{L}\ o \to \mathcal{P} \to \texttt{Prop} \\
&\Downarrow \sigma\ \texttt{pTrue} = \top \\
&\Downarrow \sigma\ \texttt{pFalse} = \bot \\
&\Downarrow \sigma\ \texttt{pOr}\ x\ y = \Downarrow \sigma\ x \vee \Downarrow \sigma\ y \\
&\Downarrow \sigma\ \texttt{pAnd}\ x\ y = \Downarrow \sigma\ x \wedge \Downarrow \sigma\ y \\
&\Downarrow \sigma\ \texttt{pImpl}\ x\ y = \Downarrow \sigma\ x \Rightarrow \Downarrow \sigma\ y \\
&\Downarrow \sigma\ \texttt{pOther}\ i = \sigma[i]
\end{aligned}
$$

which convert between the source logic $\texttt{Prop}$ and the target theory $\mathcal{P}$ using the context map σ.

Example 23 (Reification)**.** Let $\sigma := a :: b :: \square$, the reification of $a \wedge b$ is:

$$\Uparrow \sigma\ (a \wedge b) \equiv_{\beta\Sigma} \texttt{pAnd}(\texttt{pOther}\ 0)(\texttt{pOther}\ 1)$$

and its denotation

$$\Downarrow \sigma\ (\texttt{pAnd}(\texttt{pOther}\ 0)(\texttt{pOther}\ 1)) \equiv_{\beta\Sigma} a \wedge b.$$

Next, we define a decidable equality between terms of type $\mathcal{P}$.

Definition 9.8.3 (Decidable equality $=_{\mathcal{P}} : \mathcal{P} \to \mathcal{P} \to \mathbb{B}$)**.**

$\texttt{pTrue} =_{\mathcal{P}} \texttt{pTrue} \hookrightarrow \texttt{true}$

$\texttt{pFalse} =_{\mathcal{P}} \texttt{pFalse} \hookrightarrow \texttt{true}$

$\texttt{pAnd}\ l_1\ r_1 =_{\mathcal{P}} \texttt{pAnd}\ l_2\ r_2 \hookrightarrow l_1 =_{\mathcal{P}} l_2\ \texttt{andb}\ r_1 =_{\mathcal{P}} r_2$

$\texttt{pOr}\ l_1\ r_1 =_{\mathcal{P}} \texttt{pOr}\ l_2\ r_2 \hookrightarrow l_1 =_{\mathcal{P}} l_2\ \texttt{andb}\ r_1 =_{\mathcal{P}} r_2$

$\texttt{pImpl}\ l_1\ r_1 =_{\mathcal{P}} \texttt{pImpl}\ l_2\ r_2 \hookrightarrow l_1 =_{\mathcal{P}} l_2\ \texttt{andb}\ r_1 =_{\mathcal{P}} r_2$

$\texttt{pOther}\ x =_{\mathcal{P}} \texttt{pOther}\ y \hookrightarrow x =_{\mathbb{N}} y$

with $=_{\mathbb{N}} : \mathbb{N} \to \mathbb{N} \to \mathbb{B}$ a decidable equality between $\mathbb{N}$.

We now describe the notion of context Γ that is encoded as a $\mathbb{L}$ `propS`. The context will be used for learning propositions introduced by implication (`pImpl`) in the reified goal. We define what it means for all members of a context Γ to represent true propositions, and we prove some lemmas about this notion.

The predicate `all` (Equation (27)) transforms a list of $\mathcal{P}$ into conjunctions to represent a context. It used a fold right function from the Lambdapi standard library.

$$\texttt{all} : \texttt{list}\ \mathcal{P} \to \mathcal{P} \coloneqq \texttt{foldr}\ (\texttt{pAnd})\ \texttt{pTrue} \tag{27}$$

The function `knows` (Equation (28)) searches for an element decidably equal to p. It also use the function `existsb` from the standard library.

$$\texttt{knows}\ (p : \mathcal{P})\ (\Gamma : \texttt{list}\ \mathcal{P}) \coloneqq \texttt{existsb}\ (=_{\mathcal{P}})\ \Gamma \tag{28}$$

The function `learn` extends the list of known facts Γ to include one more fact. We destruct conjunctions so that we can prove theorems such as $p \wedge q \Rightarrow p$. For simplicity, in the case of other constructors, we just learn it.

$\texttt{learn}\ (p : \mathcal{P})\ (\Gamma : \texttt{list}\ \mathcal{P}) : \texttt{list}\ \mathcal{P}$

$\texttt{learn}\ \texttt{pTrue}\ \Gamma \hookrightarrow \Gamma$

$\texttt{learn}\ \texttt{pFalse}\ \Gamma \hookrightarrow (\texttt{pFalse} :: \Gamma)$

$\texttt{learn}\ (\texttt{pAnd}\ x\ y)\ \Gamma \hookrightarrow \texttt{learn}\ x\ (\texttt{learn}\ y\ \Gamma)$

$\texttt{learn}\ p\ \Gamma \hookrightarrow \texttt{learn}\ (p :: \Gamma)$

Correctness of `learn` says that if the original context is valid, and the fact that learning is valid, then the learned context is valid. In the following, we omit the context parameter σ in both $\Downarrow$ and $\Uparrow$ whenever it is clear from the context.

Lemma 16 (Correctness of learning)**.** Given σ : `list` o and a context Γ : `list` $\mathcal{P}$ then for any $p : \mathcal{P}$, if

$$\texttt{Prf}(\Downarrow \texttt{all}\ \Gamma) \text{ and } \texttt{Prf}(\Downarrow p)$$

then

$$\texttt{Prf}(\Downarrow \texttt{all}(\texttt{learn}\ p\ \Gamma))$$

Proof. By induction on p.

Since all the cases follow the same patterns, we provide the case for pAnd, and a similar approach can be follow for the rest. We want to prove that Prf(⇓ all(learn (pAnd a b) Γ)) which rewrites to Prf(⇓ (a)∧ ⇓ (b)∧ ⇓ (foldr pTrue Γ)) by definition of all, and we suppose that Prf(⇓ all Γ) and Prf(⇓ pAnd a b). Then we apply the introduction rule of the conjunction ($\wedge_i$) giving us two case to prove.

- Case Prf ⇓ a which follows directly from the hypothesis that Prf(⇓ pAnd a b),
- Case Prf(⇓ b∧ ⇓ foldr pTrue Γ) which follows from the hypothesis that Prf ⇓ (pAnd a b) and Prf(⇓ all Γ).

□

Now we want to prove that if our function knows returns true, then the proposition is implied by the meaning of the context:

Lemma 17 (knows correct)**.** Given a mapping σ : list o with a context Γ : list $\mathcal{P}$ and a proposition p : $\mathcal{P}$, if Prf ⇓ all Γ and knowns p Γ = true then Prf ⇓ p.

Proof. By induction on the context list Γ.

> **Case** Γ = □, trivial because we cannot have knows p □ = true.
>
> **Case** Γ = x :: xs, assume we have H : knows p (x :: xs) = true and H2 :⇓ all (x :: xs). We know by H, that p is equal to x or p is equal to an element in xs. If x is equal to p, one has ⇓ x from H2, hence we have ⇓ p. In the other case, ⇓ p follows from the induction hypothesis and H2.

□

The function provable implements our decision procedure. It checks if a proposition goal : $\mathcal{P}$ is provable in the context Γ.

```
provable (goal : P) (Γ : list P) : 𝔹
provable pTrue Γ ↪ true
provable pFalse Γ ↪ knows pFalse Γ
provable (pAnd x y) Γ ↪ (provable x Γ) and (provable y Γ)
provable (pOr x y) Γ ↪ (provable x Γ) or (provable y Γ)
provable (pImpl x y) Γ ↪ provable y (learn x Γ)
provable (pOther i) Γ ↪ (knows (pOther i) Γ)
                         or (knows pFalse Γ)
```

Theorem 4 (Correctness of provable)**.** For any goal : $\mathcal{P}$ and context Γ : list $\mathcal{P}$, if Prf(provable goal Γ = true) then given a mapping σ, if the context Prf(⇓ All Γ) represent true propositions then the Prf(⇓ goal) is true.

Proof. By induction on p.

Case (p = pTrue), we have to prove ⇓ pTrue which rewrites into ⊤ which holds trivially.

Case (p = pFalse), It follow from the Lemma 17 with our hypothesis provable pFalse Γ = true and ⇓ All Γ that ⇓ pFalse hold.

Case (p = pAnd a b), follow directly by induction hypothesis since provable (pAnd a b) Γ = true implies provable a Γ = true and provable b Γ = true.

Case (p = pOr a b), follow directly by induction hypothesis since provable pOr a b Γ = true implies provable a Γ = true or provable b Γ = true.

Case (p = pImpl a b), follow directly by induction hypothesis and Lemma 16.

Case (p = pOther i), let's consider an index i : ℕ, by the hypothesis Prf(provable pOther i Γ = true) we know that (pOther i) ∈ Γ or pFalse ∈ Γ. Therefore, if (pOther i) ∈ Γ then by Lemma 17 and the hypothesis ⇓ All Γ we have ⇓ pOther i. However, in the case that pFalse ∈ Γ, by using Lemma 17 and the hypothesis ⇓ All Γ we derive ex falso quodlibet.

□

```
1 symbol reify goal ≔ ⇑ □ goal;
2
3 symbol tauto_aux [goal] ≔
4   let res ≔ reify goal in provable_sound (res 1) □ (res 2);
5
6 symbol tauto ≔ #refine "(tauto_aux ⊤ᵢ ⊤ᵢ)";
7
8 symbol example1: Prf ((⊤ ∧ ⊤) ⇒ ⊤ ∨ ⊤ ∧ (⊤ ⇒ ⊤)) ≔
9 begin eval tauto; end;
```

Listing 14: Usage of Tauto

As shown in Listing 14, we define a new tactic tauto using a tactic script at line 5, and demonstrate its usage at lines 8-9. While this procedure is sound, it is incomplete: it may not always find proofs for all provable statements. Empirical results, however, show that it is sufficient to handle problems like those in Listing 13. That said, the current implementation fails to prove goals involving quantifiers, such as: (∀x, ∀y, x ∧ p y) = (∀x, x ∧ (∀y, p y)). For such cases, we still rely on the Why3 prover until the procedure is extended to support quantifiers, reducing the need for external ATP.

9.9 THE contraction RULE

The Alethe rule contraction removes duplicate literals from a clause without changing its truth. Because of the challenges posed by associativity and commutativity (AC), reconstructing this rule requires using the second kind of proof-by-reflection method.

$$\begin{array}{lll} \mathrm{i}.\ \triangleright & \mathrm{l}_1, \ldots, \mathrm{l}_n & (\ldots) \\ \mathrm{j}.\ \triangleright & \mathrm{l}_{k_1}, \ldots, \mathrm{l}_{k_m} & (\texttt{contraction i}) \end{array}$$

where $m \leq n$ and $k_1 \ldots k_m$ is a monotonic map to $1 \ldots n$ such that: $l_{k_1}, \ldots, l_{k_m}$ are pairwise distinct, and Hence, this rule removes duplicated literals.

In our reconstruction, we use an assignment $\sigma : \mathbb{L}\ o$ computed automatically from the literals of the clause $l_{k_1}, \ldots, l_{k_m}$. Each literal is associated with its position index in σ, which induces a total order on terms. This order allows us to define a canonical representation of clauses: we turn a clause into a list of literal identifiers, stably sort this list using mergesort, and then remove adjacent duplicates. Correctness follows from three ingredients: a denotational interpreter $\Downarrow$ parameterized by σ, a mergesort whose denotation is extensionally equal to its input, and a duplicate-removal pass whose denotation is idempotent with respect to disjunction.

We first encode a simple canonical algebra of literals:

$$\begin{array}{ll} \mathcal{C} : \texttt{TYPE} & \texttt{cl} : \texttt{Set} \\ \mid \mathrm{p} : \mathbb{N} \to \mathcal{C} & \texttt{El cl} \hookrightarrow \mathcal{C} \end{array}$$

A clause is then represented as a list of elements of $\mathcal{C}$. In this construction, contraction is performed solely on the left-hand side term t_1 representing the non-reduced clause $l_1, \ldots, l_n$ of step i.

$$\begin{array}{lccc} \Uparrow t_1 = \Uparrow t_2 & \mathbb{L}\ \texttt{cl} & \xrightarrow{([_],\mathrm{id})} & \mathbb{L}\ \texttt{cl} \\ & \uparrow{\scriptstyle \Uparrow(_)} & \lrcorner & \downarrow{\scriptstyle \Downarrow(_)} \\ t_1 = t_2 & \texttt{Clause} & \cdot \equiv_{\beta\Sigma} \cdot & \texttt{Clause} \end{array} \qquad \begin{array}{l} [\Uparrow t_1] = \Uparrow t_2 \\ \\ (\Downarrow [\Uparrow g_1]) = (\Downarrow \Uparrow g_2) \end{array}$$

with $[_] = \texttt{remove_duplicates} \circ \texttt{sort}$

Definition 9.9.1 (Denotation)**.** Given an assignment $\sigma : \mathbb{L}\ o$, the denotation function

$$\Downarrow : \mathbb{L}\ o \to \mathbb{L}\ \texttt{cl} \to \texttt{Prop}$$

maps a list of identifiers to the big disjunction of the corresponding literals:

$$\begin{array}{l} \Downarrow\ \sigma\ \square \hookrightarrow \bot \\ \Downarrow\ \sigma\ (\mathrm{p\ i} :: \ell) \hookrightarrow \sigma[\mathrm{i}]^{\vee} \Downarrow\ \sigma\ \ell. \end{array}$$

Definition 9.9.2 (Reification into $\mathbb{L}$ cl)**.** Reification translates a clause into its canonical list of identifiers together with the remaining syntactic literals. It proceeds by traversing the clause, looking up each

literal in the current environment, and assigning it a fresh identifier if none is found.

$$\Uparrow : \mathbb{L}\ o \to \texttt{Clause} \to \texttt{El}(\texttt{list cl} \times \texttt{list o})$$

$$\Uparrow (\ell,\ \square) \hookrightarrow (\square,\ \ell)$$
$$\Uparrow (\ell,\ x \vee xs) \hookrightarrow \text{let } (p, \ell') =$$
$$\begin{cases} (p\,i,\ \ell) & \text{if index } i \text{ already exists in } \ell \\ (p\,i, \ell +\!\!+ (x :: \square)) & \text{otherwise } i = \texttt{length}(\ell) + 1 \end{cases}$$
in
then recursively reify the tail: $(xs', \ell'') := \Uparrow\ \ell'\ xs$,
and return $(p :: xs', \ell'')$.

We then define the top-level reification function

$$\Uparrow\ c := \Uparrow\ \square\, c.$$

Example 24 (Reification)**.** Given $\sigma := x :: y :: \square$, the clause:

$$x \vee y \vee x \vee y \vee \blacksquare$$

is then reified into:

$$p\,0 :: p\,1 :: p\,0 :: p\,1 :: \blacksquare$$

We now can start to define the necessary functions and lemmas for the mergesort.

Definition 9.9.3 (split)**.** The function `split` divides a generic list into two sublists by alternating elements:

$$\texttt{split} : \Pi\, a : \texttt{Set}, \mathbb{L}\ a \to \texttt{El}(\texttt{list}\ a \times \texttt{list}\ a)$$
$$\texttt{split}\ \square \hookrightarrow (\square, \square)$$
$$\texttt{split}\ (x :: \square) \hookrightarrow (x :: \square, \square)$$
$$\texttt{split}\ (x :: y :: \ell) \hookrightarrow (x :: (\texttt{split}\ \ell)_1, y :: (\texttt{split}\ \ell)_2)$$

To reason semantically about these lists, we define $\Downarrow\ \sigma\, g$ as the denotation of g under assignment σ. The following two lemmas show that $\Downarrow$ interacts well with list concatenation and with the splitting procedure.

Lemma 18 (Concatenation distributes over $\Downarrow$)**.** For all lists x, y of identifiers and any assignment σ,

$$\texttt{Prf}(\Downarrow\ \sigma\,(x +\!\!+ y)\ =\ \Downarrow\ \sigma\, x \vee\ \Downarrow\ \sigma\, y).$$

Lemma 19 (Split preserves denotation)**.** For all identifier lists g and assignments σ,

$$\texttt{Prf}(\Downarrow\ \sigma\,((\texttt{split}\, g)_1\ +\!\!+\ (\texttt{split}\, g)_2)\ =\ \Downarrow\ \sigma\, g).$$

Definition 9.9.4 (merge)**.** The merge function combines two sorted lists into a single sorted list:

$$\texttt{merge} : \mathbb{L}\ \texttt{cl} \to \mathbb{L}\ \texttt{cl} \to \mathbb{L}\ \texttt{cl}$$
$$\texttt{merge}\ \square\ \ell \hookrightarrow \ell$$
$$\texttt{merge}\ \ell\ \square \hookrightarrow \ell$$
$$\texttt{merge}\ (\texttt{p}\ \texttt{i} :: \ell_1)\ (\texttt{p}\ \texttt{j} :: \ell_2) \hookrightarrow \begin{cases} \texttt{p}\ \texttt{i} :: \texttt{p}\ \texttt{j} :: \texttt{merge}(\ell_1, \ell_2) & \text{if } \texttt{i} = \texttt{j} \\ \texttt{p}\ \texttt{i} :: \texttt{merge}(\ell_1, \texttt{p}\ \texttt{j} :: \ell_2) & \text{if } \texttt{i} < \texttt{j} \\ \texttt{p}\ \texttt{j} :: \texttt{merge}(\texttt{p}\ \texttt{i} :: \ell_1, \ell_2) & \text{if } \texttt{i} > \texttt{j} \end{cases}$$

Lemma 20 (Merge is denotationally additive)**.** For all g_1, g_2 and any context σ,

$$\texttt{Prf}(\Downarrow\ \sigma\,(\texttt{merge}\ g_1\ g_2) = \Downarrow\ \sigma\ g_1 \vee \Downarrow\ \sigma\ g_2).$$

With `split` and `merge`, we can now define mergesort.

Definition 9.9.5 (mergesort)**.**

$$\texttt{mergesort}\ \square \hookrightarrow \square$$
$$\texttt{mergesort}\ (x :: \square) \hookrightarrow x :: \square$$
$$\texttt{mergesort}\ (x :: y :: \ell) \hookrightarrow$$
$$\quad \text{let}\ (l_1, l_2) \coloneqq \texttt{split}\ (x :: y :: \ell)\ \text{in}$$
$$\quad\quad \texttt{merge}\ (\texttt{mergesort}\ l_1)\ (\texttt{mergesort}\ l_2)$$

After sorting, duplicate literals appear as adjacent elements. We now define a function that removes these duplicates.

Definition 9.9.6 (remove identical literals)**.**

$$\texttt{remove}\ \square \hookrightarrow \square$$
$$\texttt{remove}\ (\texttt{p}\ \texttt{i} :: \square) \hookrightarrow \texttt{p}\ \texttt{i} :: \square$$
$$\texttt{remove}\ (\texttt{p}\ \texttt{i} :: \texttt{p}\ \texttt{j} :: \ell) \hookrightarrow \begin{cases} \texttt{remove}\ (\texttt{p}\ \texttt{j} :: \ell) & \text{if } \texttt{i} = \texttt{j}, \\ \texttt{p}\ \texttt{i} :: \texttt{remove}(\texttt{p}\ \texttt{j} :: \ell) & \text{if } \texttt{i} \neq \texttt{j} \end{cases}$$

Theorem 5 (Mergesort correctness)**.** For all $g : \mathbb{L}\ \texttt{cl}$ and context σ,

$$\texttt{Prf}(\Downarrow\ \sigma\,(\texttt{mergesort}\ g)\ = \Downarrow\ \sigma\ g).$$

Theorem 6 (Contraction correctness)**.** Define

$$\texttt{contraction}\ g \coloneqq \texttt{remove}(\texttt{mergesort}\ g).$$

Then, for all g and σ,

$$\texttt{Prf}(\Downarrow\ \sigma\,(\texttt{contraction}\ g) = \Downarrow\ \sigma\ g).$$

Proof. By size-bounded induction on g, using Lemmas 19 and 20 and the distributivity Lemma 18 in the combine step. □

Example 25 (Contraction)**.** To illustrate the use of contraction in practice, consider the clause with a duplicated literal : $\texttt{Prf}^\bullet(a \vee b \vee b \vee \blacksquare)$. We show how to reconstruct the proof for the contracted clause $\texttt{Prf}^\bullet(a \vee b \vee \blacksquare)$ by appealing to the correctness of contraction.

```
1  opaque symbol ti: Prf• (a ∨ b ∨ b ∨ ■) ≔ begin ... end;
2
3  opaque symbol tj:  Prf• (a ∨ b ∨ ■) ≔
4  begin
5    have eqh : Prf (cl2disj (a ∨ b ∨ b ∨ ■) = cl2disj (a ∨ b ∨ ■)) {
6      set r ≔ ⇑ X;
7      change Prf (⇓ (r 2) (r 1) = cl2disj (a ∨ b ∨ ■));
8      rewrite left contraction_correct;
9      reflexivity
10   };
11   refine subst_cl eqh ti
12 end;
```

Here, the intermediate proof `eqh` establishes that the denotations of the original clause $(a \vee b \vee b)$ and the contracted clause $(a \vee b)$ are equal by Theorem 6. The lemma

$$\texttt{subst_cl}\ x\ y : \texttt{Prf}(\texttt{cl2disj}\ x = \texttt{cl2disj}\ y) \to \texttt{Prf}^\bullet(x) \to \texttt{Prf}^\bullet(y)$$

is then applied to transfer the proof of the original clause t_i to the contracted clause t_j.

10 RECONSTRUCTING LINEAR ARITHMETIC PROOFS

10.1 RECONSTRUCTION OF LINEAR INTEGER ARITHMETIC

In this section, we address the challenge of reconstructing the rule `la_generic`, as detailed in Section 4.5.2 when it is used in **LIA** logic. This rule corresponds to a proof step in which a set of linear inequalities is shown to be contradictory. The Alethe specification guarantees that the contradiction has to reduce to a normal form: $0 \bowtie k$ where k is constant with $k > 0$, and $\bowtie \in \{\geq, >\}$.

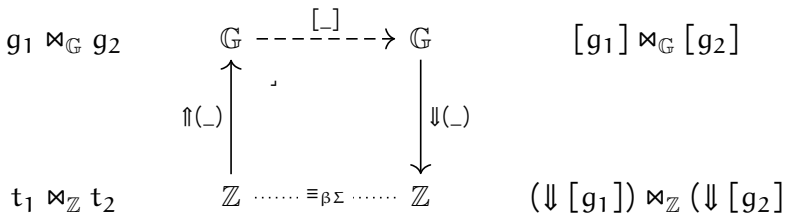
```
1 (step t11 (cl (not (<= f 0)) (<= (+ 1 (* 4 f)) 1))
2   :rule la_generic :args (1/1 1/4))
```

Unlike most of the other arithmetic rules described in Table 3, reconstructing `la_generic` poses a distinct challenge. It often requires reasoning about algebraic identities that hold modulo the axioms of algebraic structures such as groups. These identities are typically derived through elementary combinations of canonical equalities. Other rules are just encoded as regular lemmas following the previous method we described in Chapter 9.

We reconstruct steps using the `la_generic` rule through a second-kind proof by reflection. The goal is to implement an automated procedure that computes the final sum of inequalities, as described in the Alethe algorithm, by combining implicit rewriting steps to normalize and simplify expressions. Our approach relies on a normalization function along with a proof of its correctness. In this work, we support two proof-generation strategies. The first uses **`associative commutative`** modifiers to normalize terms directly within the Lambdapi kernel. This makes the approach efficient but relies on a meta-level correctness argument for normalization. The second encodes the normalization function in Lambdapi and performs symbolic execution to compute the result. While slower, this method provides a fully internal correctness proof within Lambdapi. We will denote the first strategy as the *inner normalization* and the second strategy as the *outer normalization*.

10.2 GENERAL SCHEME OF THE REFLECTIVE RECONSTRUCTION

The procedure operates on a group structure $\mathbb{G}$ that will differ between the two strategies. It includes a base type for its elements and constructors to reflect $\mathbb{Z}$, together with the usual axioms defining a commutative group structure. The goal is to prove the contradiction of an inequality between two terms $g1 \bowtie g2$ of type $\mathbb{G}$ modulo the group axioms. Working by reflection means that we build a semi de-



with $\bowtie \in \{\geq, >\}$.

Figure 19: Reflexive normalization of linear polynomials.

cision procedure like illustrated in Figure 19, which will take t1 and t2 of type $\mathbb{Z}$ as arguments and compare them (i.e. returns a term of type `Comp`) modulo associative-commutative rewriting in the group structure.

Once we have built $g_1, g_2 : \mathbb{G}$, by reifying t_1 and t_2 using the function $\Uparrow$, the idea is to check the equality of the normal forms of g_1 and g_2 denoted by the function $[_]$ and to prove that this implies the same comparison of t_1 and t_2 in $\mathbb{Z}$ through a denotation function $\Downarrow ()$. For this purpose, we should ensure the correctness of the following diagram:

10.3 REFLECTION WITH INNER NORMALIZATION

We start by presenting the first strategy based on a reflection with inner normalization.

10.3.1 *Representation of integers*

Definition 10.3.1 (Type $\mathbb{G}$)**.** The procedure is based on an algebraic group structure, denoted as $\mathbb{G}$, which represents linear polynomials. The base type for its elements is $\mathbb{G} :$ `TYPE`. The unary operator `cst` injects constants from $\mathbb{Z}$ into $\mathbb{G}$.

```
𝔾 : TYPE
| ⊕ : 𝔾 → 𝔾 → 𝔾
| var : ℤ → ℤ → 𝔾
| mul : ℤ → 𝔾 → 𝔾
| opp : 𝔾 → 𝔾
| cst : ℤ → 𝔾
grp : Set
El grp ↪ 𝔾
```

The term `var c x` is intended for representing expressions $c \times x$ that appear as constituents of linear inequalities, where `c` is an integer coefficient and `x` a $\mathbb{Z}$ term, in particular a variable. The constructor `mul` represents the multiplication of an element of $\mathbb{G}$ by a constant. The constructor `opp` corresponds to unary minus. Lastly, the constructor

$\oplus$ represents the addition between two elements of $\mathbb{G}$ and it has the `associative commutative` modifiers.

Definition 10.3.2 (Reification and denotation)**.** Reification ($\Uparrow$) and denotation ($\Downarrow$) functions for $\mathbb{G} \rightleftarrows \mathbb{Z}$ are defined as follows:

$$\Uparrow : \mathbb{Z} \to \mathbb{G}$$
$$\Uparrow \texttt{Z0} \hookrightarrow (\texttt{cst Z0})$$
$$\Uparrow \texttt{ZPos}\, c \hookrightarrow (\texttt{cst}\, (\texttt{ZPos}\, c))$$
$$\Uparrow \texttt{ZNeg}\, c \hookrightarrow (\texttt{cst}\, (\texttt{ZNeg}\, c))$$
$$\Uparrow (x + y) \hookrightarrow (\Uparrow x) \oplus (\Uparrow y)$$
$$\Uparrow (\sim x) \hookrightarrow \texttt{opp}\, (\Uparrow x)$$
$$\Uparrow ((\texttt{ZPos}\, c) * x) \hookrightarrow \texttt{mul}\, (\texttt{ZPos}\, c)\, (\Uparrow x)$$
$$\Uparrow ((\texttt{ZNeg}\, c) * x) \hookrightarrow \texttt{mul}\, (\texttt{ZNeg}\, c)\, (\Uparrow x)$$
$$\Uparrow (x * (\texttt{ZPos}\, c)) \hookrightarrow \texttt{mul}\, (\texttt{ZPos}\, c)\, (\Uparrow x)$$
$$\Uparrow (x * (\texttt{ZNeg}\, c)) \hookrightarrow \texttt{mul}\, (\texttt{ZNeg}\, c)\, (\Uparrow x)$$
$$\Uparrow x \hookrightarrow (\texttt{var 1}\, x)$$

$$\Downarrow : \mathbb{G} \to \mathbb{Z}$$
$$\Downarrow (\texttt{cst}\, c) \hookrightarrow c$$
$$\Downarrow \texttt{opp}\, x \hookrightarrow \sim (\Downarrow x)$$
$$\Downarrow \texttt{mul}\, c\, x \hookrightarrow c * (\Downarrow x)$$
$$\Downarrow x \oplus y \hookrightarrow \Downarrow x + \Downarrow y$$
$$\Downarrow (\texttt{var}\, c\, x) \hookrightarrow c * x$$

10.3.2 *Associative Commutative Inner Normalization*

The transformation to canonical form implemented in Lambdapi ensures that sum expressions of the form $(\texttt{var}\ c_1\ x_1) \oplus (\texttt{cst}\ k_1) \oplus (\texttt{var}\ c_2\ x_2) \oplus \cdots \oplus (\texttt{cst}\ k_m) \oplus \cdots \oplus (\texttt{var}\ c_n\ x_n)$ will be normalized such that any pair of terms $(\texttt{var}\ p\ x)$ and $(\texttt{var}\ q\ x)$ involving the same variable x are placed next to each other, and all $(\texttt{cst}\ k_i)$ will be placed at the left before the first variable $(\texttt{var}\ c_j\ x_j)$.

Example 26. Consider the term below not in normal form:

$$(\texttt{var}\ c_1\ x) \oplus (\texttt{cst}\ k_1) \oplus (\texttt{var}\ c_2\ y) \oplus (\texttt{cst}\ k_m) \oplus (\texttt{var}\ c_3\ x) \oplus (\texttt{var}\ c_4\ y)$$

We would like to have this normal form to reduce after the term:

$$(\texttt{cst}\ k_1) \oplus (\texttt{cst}\ k_2) \oplus (\texttt{var}\ c_1\ x) \oplus (\texttt{var}\ c_3\ x) \oplus (\texttt{var}\ c_2\ y) \oplus (\texttt{var}\ c_4\ y)$$

Definition 10.3.3 (Rewriting rules for $\mathbb{G}$)**.** Rewriting rules for reducing $\mathbb{G}$ expressions:

$$(\texttt{var}\ c_1\ x) \oplus (\texttt{var}\ c_2\ x) \hookrightarrow (\texttt{var}\ (c_1 + c_2)\ x) \tag{29}$$
$$(\texttt{var}\ c_1\ x) \oplus ((\texttt{var}\ c_2\ x) \oplus y) \hookrightarrow (\texttt{var}\ (c_1 + c_2)\ x) \oplus y \tag{30}$$
$$(\texttt{cst}\ c_1) \oplus (\texttt{cst}\ c_2) \hookrightarrow \texttt{cst}(c_1 + c_2) \tag{31}$$
$$(\texttt{cst}\ c_1) \oplus ((\texttt{cst}\ c_2) \oplus y) \hookrightarrow \texttt{cst}(c_1 + c_2) \oplus y \tag{32}$$
$$(\texttt{cst}\ 0) \oplus x \hookrightarrow x \tag{33}$$
$$x \oplus (\texttt{cst}\ 0) \hookrightarrow x \tag{34}$$
$$\texttt{opp}\ (\texttt{var}\ c\ x) \hookrightarrow (\texttt{var}\ (-c)\ x) \tag{35}$$
$$\texttt{opp}\ (\texttt{cst}\ c) \hookrightarrow (\texttt{cst}\ (-c)) \tag{36}$$
$$\texttt{opp}\ (\texttt{opp}\ x) \hookrightarrow x \tag{37}$$
$$\texttt{opp}\ (x \oplus y) \hookrightarrow (\texttt{opp}\ x) \oplus (\texttt{opp}\ y) \tag{38}$$
$$\texttt{opp}\ (\texttt{mul}\ k\ x) \hookrightarrow \texttt{mul}\ (-k)\ x \tag{39}$$
$$\texttt{mul}\ \texttt{Z0}\ (\texttt{var}\ _\ _) \hookrightarrow (\texttt{cst}\ 0) \tag{40}$$
$$\texttt{mul}\ k\ (\texttt{var}\ c\ x) \hookrightarrow (\texttt{var}\ (k * c)\ x) \tag{41}$$
$$\texttt{mul}\ k\ (\texttt{opp}\ x) \hookrightarrow \texttt{mul}\ (-k)\ x \tag{42}$$
$$\texttt{mul}\ k\ (x \oplus y) \hookrightarrow (\texttt{mul}\ k\ x) \oplus (\texttt{mul}\ k\ y) \tag{43}$$
$$\texttt{mul}\ k\ (\texttt{cst}\ c) \hookrightarrow (\texttt{cst}\ (k * c)) \tag{44}$$
$$\texttt{mul}\ c_1\ (\texttt{mul}\ c_2\ x) \hookrightarrow \texttt{mul}\ (c_1 * c_2)\ x \tag{45}$$

Remark 9. The resulting normal forms do not contain the constructors `mul` and `opp`, as the associated rewrite rules eliminate them in favor of `var`, `add` and `cst`.

Definition 10.3.4. The $\leq$ builtin total order on $\mathbb{G}$-terms is defined as follows: Terms are ordered such that $\texttt{cst}(c_1) \leq \texttt{cst}(c_2) < (\texttt{var}\ p\ x)$ for any constants c_1, c_2 and any variable term $(\texttt{var}\ p\ x)$, with $c_1 \leq c_2$. For variable terms, $(\texttt{var}\ p\ x) \leq (\texttt{var}\ q\ y)$ if either $x < y$, or $x = y$ and $p \leq q$. Let $\twoheadrightarrow^{AC}$ be the relation mapping every term t to its unique AC-canonical form denoted $[t]$.

Two terms t and u are AC-equivalent (written $t \simeq_{AC} u$) iff their AC-canonical forms are equal.

Definition 10.3.5. The relation $\longrightarrow_\Sigma^{AC}$ is defined as $\hookrightarrow_\Sigma \twoheadrightarrow^{AC}$, where Σ contains the rewrite rules of Definitions 7.2.2 and 10.3.3.

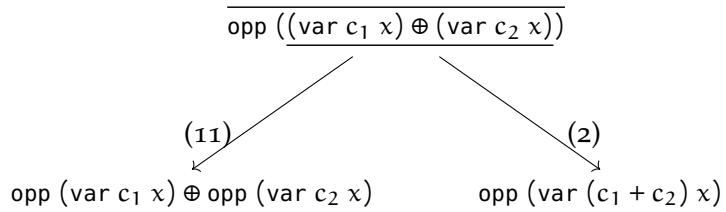
An $\longrightarrow_\Sigma^{AC}$ step is a standard $\hookrightarrow_\Sigma$ step with syntactic matching followed by AC-canonicalization. We now prove that the relation $\longrightarrow_\Sigma^{AC}$ terminates and is confluent.

Lemma 21. The relation $\longrightarrow_{\Sigma/AC} = \simeq_{AC} \hookrightarrow_\Sigma \simeq_{AC}$ of matching modulo AC, which contains $\longrightarrow_\Sigma^{AC}$, terminates.

Proof. AProVE [51] proves the termination of $\longrightarrow_{\Sigma/AC}$. □

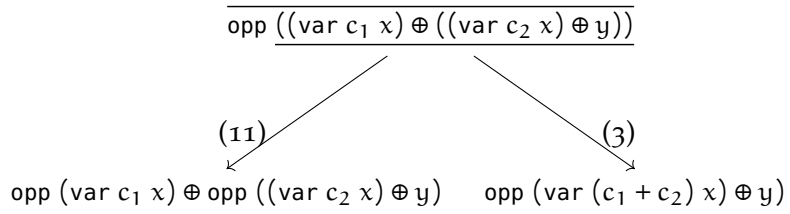
Lemma 22. $\longrightarrow_\Sigma^{AC}$ is locally confluent on AC-canonical terms.

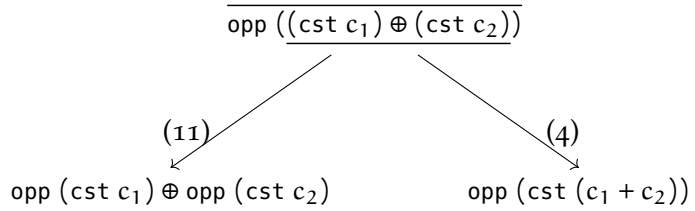
Proof. We show that every critical pair is joinable using $\longrightarrow_{\Sigma}^{AC}$ and confluence of $\rightarrow_{\mathbb{Z}}$ and $\rightarrow_{\mathbb{P}}$ from Lemma 3. We use the notation $\longrightarrow_{\mathcal{R}}^{AC*}$ to denote a multistep rewriting with AC-canonical form.

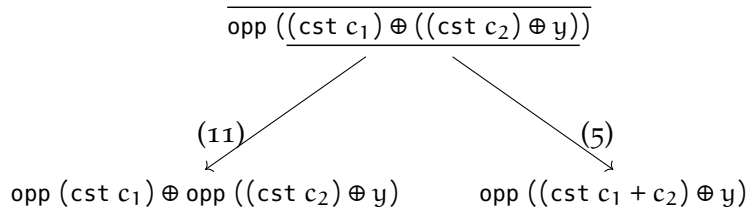


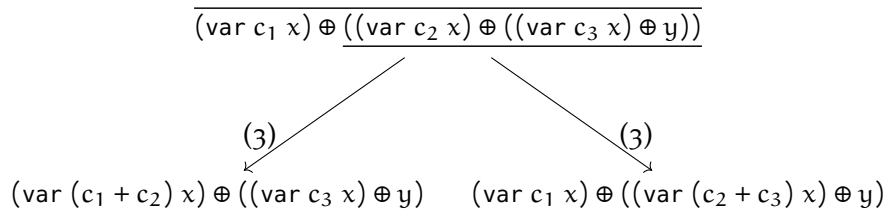
But $\mathtt{opp}\ (\mathtt{var}\ c_1\ x) \oplus \mathtt{opp}\ (\mathtt{var}\ c_2\ x) \longrightarrow_{\mathcal{R}}^{AC*} (\mathtt{var}\ (-c_1 + (-c_2))\ x)$ and $\mathtt{opp}\ (\mathtt{var}\ (c_1 + c_2)\ x) \longrightarrow_{\mathcal{R}}^{AC*} (\mathtt{var}\ -(c_1 + c_2)\ x)$ converge by using the rewrite rule $-(x + y) \hookrightarrow -x + (-y)$ in $\rightarrow_{\mathbb{Z}}$ and its confluence.

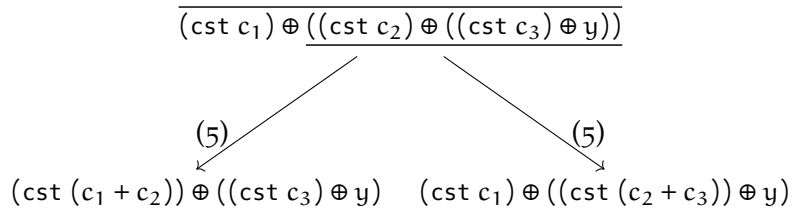
The argument is similar for all the following critical pairs:

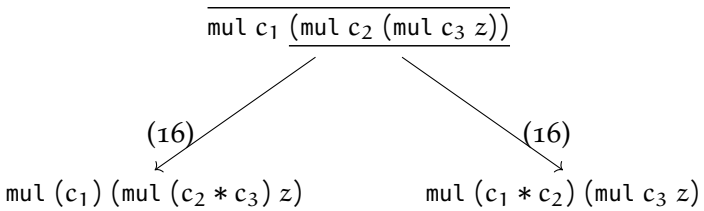












□

We compare two $\mathbb{Z}$-terms t_1 and t_2 wrt $\longrightarrow_{\Sigma}^{AC}$ by reifying them into their corresponding $\mathbb{G}$-terms, denoted $[g_1]$ and $[g_2]$, using the reification function $\Uparrow$, and normalizing them using $\twoheadrightarrow^{AC}$. Following the reduction rules specified in Definition 10.3.3, we can then compare their corresponding $\mathbb{Z}$-terms by applying the denotation function $\Downarrow$. To validate this procedure, it is necessary to establish the correctness of the Figure 19, formally expressed by Theorem 7.

Theorem 7 (Correctness of normalization)**.** For all $\mathbb{G}$-terms t, we have $(\Downarrow [t]) = (\Downarrow t)$ where $[t]$ is the AC-canonical form of t with respect to $\longrightarrow_{\Sigma}^{AC}$.

Proof. This is a *meta-level* proof, proceeding by structural induction on t, and the key case is the one where $t = t_1 \oplus t_2$. We have to show that

$$\Downarrow [t_1 \oplus t_2] = (\Downarrow t_1) + (\Downarrow t_2).$$

By the induction hypothesis, we have $\Downarrow [t_1] = \Downarrow t_1$ and $\Downarrow [t_2] = \Downarrow t_2$. Hence,

$$(\Downarrow t_1) + (\Downarrow t_2) = (\Downarrow [t_1]) + (\Downarrow [t_2]) = \Downarrow ([t_1] \oplus [t_2])$$

It remains to show that $\Downarrow ([t_1] \oplus [t_2]) = \Downarrow [t_1 \oplus t_2]$. Now, $[t_1]$, $[t_2]$, and $[t_1 \oplus t_2]$ are terms built solely from `cst`, `var`, and $\oplus$ since the remaining operators have been eliminated by applying the rules in Definition 10.3.3, and the terms on both sides of the equation contain the same multisets of subterms. The two terms are therefore identified by AC-canonicalization. □

We make use of Lemma 23 to embed $\mathbb{Z}$-terms into $\mathbb{G}$ to normalize and subsequent comparison. In addition, we leverage this normalization process to support the `arith-poly-norm` rule.

Lemma 23 (Conversion)**.** For all $x : \mathbb{Z}$, we have $x = (\Downarrow (\Uparrow x))$.

Proof. By induction on x. We consider the three cases: $x = \texttt{ZPos}(n)$, $x = \texttt{ZNeg}(n)$, and $x = \texttt{Z0}$. In each case, $\Uparrow (x)$ yields the corresponding constant $(\texttt{cst}\ x)$, and by definition of the denotation function, $x = \Downarrow ((\texttt{cst}\ x))$. Hence, $x = (\Downarrow (\Uparrow x))$ in all cases. □

```
1 opaque symbol t2: Prf• ((¬ (3 < x) ∨ ¬ (x = 2)) ∨ ■) {
2     apply ∨ il;
3     rewrite Zinv_lt_eq;
4     rewrite Z_diff_gt_Z0_eq (− 3) (− x);
5     rewrite Z_diff_eq_Z0_eq (x) 2;
6     rewrite Zgt_le_succ_r_eq ((− 3) - (− x)) 0;
```

```
7     rewrite Zmult_ge_compat_eq 1 ((− 3) - (− x)) ((0 + 1));
8     rewrite Zmult_eq_compat_eq (− 1) (x - 2) 0;
9     rewrite imp_eq_or; assume H0 H1;
10
11    set H0l' ≔ (1 * ((− 3) - (− x))); set H0r' ≔ (1 * (0 + 1));
12    set H1l' ≔ ((− 1) * (x - 2)); set H1r' ≔ ((− 1) * 0);
13    have H1': Prf^c (H1l' ≥ H1r') { refine Z_eq_implies_ge H1 }; remove H1;
14    have contra : Prf^c ((⇓ (⇑ (H0l' + H1l'))) ≥ (⇓ (⇑ (H0r' + H1r')))){
15        rewrite ⇓_⇑;
16        rewrite ⇓_⇑;
17        apply (Zsum_geq_s H0l' H0r' H1l' H1r' H0 H1');
18    };
19    apply contra; apply ⊤^c i;
20 };
```

Listing 15: Lambdapi proof of List. 7

We present in List. 15 the translation of Step `t2` from List. 7 using our Carcara module. The **`rewrite`** `Zinv_lt_eq` step at line 3 performs the normalization from Step 1 of the Algorithm 2, rewriting $\neg(3 < x)$ as $\neg(\sim 3 >\sim x)$. The lemmas `Z_diff_gt_Z0` and `Z_diff_eq_Z0_eq` (lines 4–5) implement Step 2. Since the goal involves a $>$ inequality, Step 3 applies with the lemma `Zgt_le_succ_r_eq` (line 6). The coefficients $[\frac{1}{1}, -\frac{1}{1}]$ from Step 4 are trivially cast to $\mathbb{Z}$ and applied in lines 7–8. To express all inequalities uniformly, the remaining equalities are rewritten as $\geq$ using `Z_eq_implies_ge` (line 12). Finally, Step 6 derives the contradiction by summing inequalities from lines 13 to 17.

10.4 REFLECTION WITH OUTER NORMALIZATION

We now describe the second strategy, which implements the normalization function entirely within Lambdapi. This approach benefits from an entirely correctness proof at the object theory level, but it typically results in slower performance compared to the first strategy. As in the first approach, an inductive type $\mathbb{G}$ reflects the syntactic representation of arithmetic expressions over $\mathbb{Z}$.

10.4.1 *Reflecting integers*

Notation. We write $a[i]$ for $(\texttt{nth}\ 0\ i\ a)$, i.e., the i-th element of the list a, with 0 as default value to make the function total. This notation will mainly be used when inspecting an element of a context map during reification and denotation.

Definition 10.4.1. The algebraic group $\mathbb{G}$ reflecting $\mathbb{Z}$ atoms:

$$\mathbb{G} : \texttt{TYPE}$$
$$\mid\ \texttt{var} : \mathbb{N} \to \mathbb{Z} \to \mathbb{G}$$
$$\mid\ \texttt{cst} : \mathbb{Z} \to \mathbb{G}$$

$$\texttt{grp} : \texttt{Set}$$
$$\texttt{El grp} \hookrightarrow \mathbb{G}$$

The main differences with Definition 10.3.1 are as follows. First, the constructor `var` indexes uninterpreted terms by $\mathbb{N}$ rather than encapsulating them directly. Second, the operator $\oplus$ is omitted, since finite

sums are represented by lists ($\mathbb{L}$ `grp`) in order to obtain a flattened structure. Third, the operator `mul` is defined externally, so as to simplify subsequent definitions and proofs.

Definition 10.4.2 (multiplication of coefficient)**.**

$$
\begin{aligned}
&\texttt{mul} : \mathbb{Z} \to \mathbb{L}\ \texttt{grp} \to \mathbb{L}\ \texttt{grp}\\
&\texttt{mul}\ _\ \square \hookrightarrow \square\\
&\texttt{mul}\ k\ (\texttt{cst}\ c :: tl) \hookrightarrow \texttt{mul}\ (k * c)\ tl\\
&\texttt{mul}\ k\ ((\texttt{var}\ i\ c) :: tl) \hookrightarrow (\texttt{var}\ i\ (k * c)) :: \texttt{mul}\ k\ tl
\end{aligned}
$$

Definition 10.4.3 (Reification function)**.** The `reify` function:

$$\Uparrow : \mathbb{L}\ \texttt{int} \to \mathbb{Z} \to \texttt{El}(\texttt{list grp} \times \texttt{list int})$$

performs reification of arithmetic expressions from $\mathbb{Z}$ into the abstract type $\mathbb{G}$, producing a pair consisting of a reified term of type $\mathbb{G}$, and a list of variable values `list int` called the *context map* that represents uninterpreted terms in domain of $\mathbb{G}$ such as variable. It is defined recursively over the structure of $\mathbb{Z}$ expressions:

$$
\begin{aligned}
&(\Uparrow \ell\ (x + y)) \hookrightarrow \text{let}\ (g_1, \ell_1) = (\Uparrow \ell\ x), (g_2, \ell_2) = (\Uparrow \ell_1\ y)\\
&\qquad \text{in}\ (g_1\ +\!\!+\ g_2, \ell_2)\\
&(\Uparrow \ell\ (-x)) \hookrightarrow \text{let}\ (g, \ell') = (\Uparrow \ell\ x)\ \text{in}\ (\texttt{mul}\ (-1)\ g), \ell')\\
&(\Uparrow \ell\ (x - y)) \hookrightarrow (\Uparrow \ell\ (x + (-y)))\\
&(\Uparrow \ell\ 0) \hookrightarrow (\texttt{cst}\ 0\ +\!\!+ \square, l)\\
&(\Uparrow \ell\ (\texttt{ZPos}\ c)) \hookrightarrow (\texttt{cst}\ (\texttt{ZPos}\ c)\ +\!\!+ \square, l)\\
&(\Uparrow \ell\ (\texttt{ZNeg}\ c)) \hookrightarrow (\texttt{cst}\ (\texttt{ZNeg}\ c)\ +\!\!+ \square, l)\\
&(\Uparrow \ell\ (0 * x)) \hookrightarrow (\texttt{cst}\ 0\ +\!\!+ \square, l)\\
&(\Uparrow \ell\ ((\texttt{ZPos}\ c) * x)) \hookrightarrow \text{let}\ (g, \ell') = (\Uparrow \ell\ x)\ \text{in}\\
&\qquad (\texttt{mul}\ (\ (\texttt{ZPos}\ c), g), \ell')\\
&(\Uparrow \ell\ ((\texttt{ZNeg}\ c) * x)) \hookrightarrow \text{let}\ (g, \ell') = (\Uparrow \ell\ x)\ \text{in}\\
&\qquad (\texttt{mul}\ (\ (\texttt{ZNeg}\ c), g), \ell')\\
&(\Uparrow \ell\ (x * (\texttt{ZPos}\ c))) \hookrightarrow \text{let}\ (g, \ell') = (\Uparrow \ell\ x)\ \text{in}\\
&\qquad (\texttt{mul}\ (\ (\texttt{ZPos}\ c), g), \ell')\\
&(\Uparrow \ell\ (x * (\texttt{ZNeg}\ c))) \hookrightarrow \text{let}\ (g, \ell') = (\Uparrow \ell\ x)\ \text{in}\\
&\qquad (\texttt{mul}\ (\ (\texttt{ZNeg}\ c), g), \ell')\\
&(\Uparrow \ell\ x) \hookrightarrow \begin{cases} ((\texttt{var}\ i\ 1)\ +\!\!+ \square, \ell) & \text{if } x \text{ is } \ell[i]\\ ((\texttt{var}\ (\texttt{size}(\ell))\ 1)\ +\!\!+ \square, \ell\ +\!\!+ (x :: \square)) & \text{otherwise} \end{cases}
\end{aligned}
$$

Definition 10.4.4 (Denotation function)**.** The function:

$$\Downarrow : \texttt{El}(\texttt{list grp} \times \mathbb{L}\ \texttt{int}) \to \mathbb{Z}$$

defines the semantic interpretation of reified arithmetic expressions of type $\mathbb{G}$.

$$\Downarrow \square \hookrightarrow 0$$
$$\Downarrow (h :: tl, \ell) \hookrightarrow \Downarrow_{\mathbb{G}} (h, \ell) + \Downarrow (tl, \ell)$$

with

$$\Downarrow_{\mathbb{G}} : \texttt{El}(\texttt{grp} \times \mathbb{L}\ \texttt{int}) \to \mathbb{Z}$$
$$\Downarrow_{\mathbb{G}} (\texttt{cst}\ (k), _) \hookrightarrow k$$
$$\Downarrow_{\mathbb{G}} ((\texttt{var}\ i\ k), \ell) \hookrightarrow \ell[i] * k$$

Example 27. Consider the arithmetic expression $x + y + x + 1$, where x and y are $\mathbb{Z}$ variables. When reifying this expression, we start with an empty context $\square$ and apply the $(\Uparrow)$ function recursively.

$$(\Uparrow \square\ (x + 2 * y + x + 1)) \hookrightarrow^* (\mathsf{Var}\ 0\ 1 :: \mathsf{Var}\ 1\ 2 :: \mathsf{Var}\ 0\ 1 :: (\texttt{cst}\ 1) :: \square,\ \ell)$$

The final context map $\ell = x :: y :: \square$, so that x has index 0 and y has index 1.

10.4.2 *Outer normalisation procedure*

In this definition of merge sort algorithm, we reuse the split function defined in Definition 9.9.3 used for the `contraction` rule.

Definition 10.4.5 (Merge)**.** The function `merge` combines two lists of group elements by folding constants and variables with the same index, and produces a single list that is sorted by variable indices assuming the inputs are themselves sorted.

$$\texttt{merge} : \texttt{list grp} \times \texttt{list grp} \to \texttt{list grp}.$$

It proceeds by structural recursion:

$$\texttt{merge}\ \square\ \square \hookrightarrow \square$$
$$\texttt{merge}\ \square\ \ell_2 \hookrightarrow \ell_2$$
$$\texttt{merge}\ \ell_1\ \square \hookrightarrow \ell_1$$
$$\texttt{merge}\ (\texttt{cst}\ (c_1) :: \ell_1')\ (\texttt{cst}\ (c_2) :: \ell_2') \hookrightarrow$$
$$\quad \texttt{cst}\ (c_1 + c_2) :: \texttt{merge}\ (\ell_1', \ell_2')$$
$$\texttt{merge}\ (\texttt{cst}\ (c) :: \ell_1')\ ((\texttt{var}\ i\ k) :: \ell_2') \hookrightarrow$$
$$\quad \texttt{cst}\ (c) :: \texttt{merge}\ (\ell_1', (\texttt{var}\ i\ k) :: \ell_2')$$
$$\texttt{merge}\ ((\texttt{var}\ i\ k) :: \ell_1')\ (\texttt{cst}\ (c) :: \ell_2') \hookrightarrow$$
$$\quad \texttt{cst}\ (c) :: \texttt{merge}\ ((\texttt{var}\ i\ k) :: \ell_1', \ell_2')$$
$$\texttt{merge}\ ((\texttt{var}\ i\ k_1) :: \ell_1', (\texttt{var}\ j\ k_2) :: \ell_2') \hookrightarrow$$
$$\begin{cases} (\texttt{var}\ i\ (k_1 + k_2)) :: \texttt{merge}\ (\ell_1', \ell_2') & \text{if } i = j, \\ (\texttt{var}\ i\ k_1) :: \texttt{merge}\ (\ell_1', (\texttt{var}\ j\ k_2) :: \ell_2') & \text{if } i < j, \\ (\texttt{var}\ j\ k_2) :: \texttt{merge}\ ((\texttt{var}\ i\ k_1) :: \ell_1', \ell_2') & \text{if } i > j. \end{cases}$$

Definition 10.4.6 (Mergesort)**.** The sorting function `mergesort` is defined by recursion:

$$
\begin{aligned}
&\texttt{mergesort}\ \square \hookrightarrow \square,\\
&\texttt{mergesort}\ (x :: \square) \hookrightarrow x :: \square,\\
&\texttt{mergesort}\ (x :: y :: \ell) \hookrightarrow\\
&\texttt{let}(l_1, l_2) := \texttt{split}\ \ell\ \texttt{in}\\
&\quad \texttt{merge}(\texttt{mergesort}\ l_1, \texttt{mergesort}\ l_2).
\end{aligned}
$$

Normalization will consist of converting a `list grp` expression into a canonical form through a composition of functions:

Definition 10.4.7 (normalisation function)**.** Given $x : \texttt{list grp}$,

$$\texttt{norm}\, x := \texttt{remove0}\ (\texttt{mergesort}\, x)$$

10.4.3 *Correctness of Reified Normalization*

Several helper lemmas establish the soundness of the normalization pipeline, by verifying that each transformation step preserves the semantic interpretation of the term. In the following, we omit the proof modality symbol `Prf` to simplify the notation.

We first establish the correctness of the auxiliary operations that will be used in the normalization procedure. These lemmas ensure that splitting, merging, and removing trivial elements preserve the denotational semantics.

Lemma 24 (Concatenation preserves denotation)**.** For all lists x, y of `grp` and all environments ℓ, we have

$$\Downarrow (x \mathbin{+\!\!+} y,\ \ell) = \Downarrow (x, \ell) + \Downarrow (y, \ell).$$

Proof. Direct by induction on the structure of x. □

Lemma 25 (Split correctness)**.** For all ℓ and gs, given $\texttt{split}\ gs = (gs_1, gs_2)$, we have

$$\Downarrow ((gs_1 \mathbin{+\!\!+} gs_2, \ell) = (\Downarrow\ gs\ \ell).$$

Proof. By two-step induction on the structure of gs.

- (Case $gs = \square$ or singletons), are immediate by definition of `split`.
- (Case $gs = a :: b :: tl$), it is immediate by the induction hypothesis and Lemma 24.

The merge function combines two already-sorted lists of $\mathbb{G}$ elements. The following lemma states that it behaves extensionally like addition of the corresponding denotations.

Lemma 26 (Merge correctness)**.** For all ℓ, g_1, g_2, we have

$$\Downarrow (\texttt{merge}\ g_1\ g_2, \ell) = \Downarrow (g_1, \ell) + \Downarrow (g_2, \ell).$$

Proof. By induction on g_1.

- If $g_1 = \square$, the result follows trivially.
- If g_1 starts with a constant, use the recursive rule and induction hypothesis.
- If g_1 begins with a variable of the form $(\texttt{var}\ i\ k_i)$, we proceed by case analysis on the structure of g_2. In the situation where both g_1 and g_2 begin with variables, say $(\texttt{var}\ i\ k_i)$ and $(\texttt{var}\ j\ k_j)$ respectively, we distinguish cases according to the comparison between the indices i and j, and establish the desired equality in each branch.

□

To show that mergesort terminates because we defined it as a general recursive function, and is well-behaved, we need to control the size of the sublists obtained by splitting. The following lemma provides the necessary bound.

Lemma 27 (Size bound for split)**.** For all lists ℓ, with $\texttt{split}\ \ell = (\ell_1, \ell_2)$, we have:

$$\texttt{size}\ \ell_1 \text{ and } \texttt{size}\ \ell_2 \leq \texttt{size}\ \ell$$

Proof. By induction on ℓ, using the definition of `split` and the monotonicity of `size`. □

We can now combine the above invariants to show that mergesort preserves denotations.

Lemma 28 (Correctness of mergesort)**.** For all n, ℓ, g, if $\texttt{size}(g) \leq n$, then

$$\Downarrow (\texttt{mergesort}\ g, \ell) = \Downarrow (g, \ell).$$

Proof. By induction on n.

(Case $n = 0$), we then have $\texttt{size}(g) = 0$, then $g = \square$ and thus $\texttt{mergesort}\ g = \square$, so $\Downarrow (\texttt{mergesort}\ \square, \ell) = \Downarrow (\square, \ell)$ holds directly.

(Case $n > 0$), with the definition of `mergesort` and `split`, apply the induction hypothesis to both sublists with the bounds given by Lemma 27, and combine using Lemma 26 and Lemma 25.

□

Next, we show that eliminating trivial constants and variables with zero coefficient does not change the denotation.

Lemma 29 (Removal of zeros)**.** For all context ℓ and sum gs, we have

$$\Downarrow (\texttt{remove0}\ gs, \ell) = \Downarrow (gs, \ell).$$

Proof. We proceed by induction on the list gs. In the inductive step where $gs = x :: tl$, we perform a case analysis on the head element of gs:

(Case $x = \texttt{cst c}$), if $c = 0$, the element is removed and the result follows by the induction hypothesis. If $c \neq 0$, the head is preserved and the induction hypothesis applies to the tail.

(Case $x = (\texttt{var i k})$), if $k = 0$, the element is removed and the result follows by the induction hypothesis. If $k \neq 0$, the head is preserved and the induction hypothesis applies to the tail.

In all cases, the denotation is preserved, hence the lemma holds. □

Finally, we can combine mergesort and zero-removal into a single normalization procedure. The following theorem states that normalization preserves denotation.

Theorem 8 (Correctness of normalization)**.** For all lists x and ℓ,

$$\Downarrow (\texttt{norm}\, x, \ell) = \Downarrow (x, \ell),$$

Proof. Combine Lemma 29 and Lemma 28 with a structural induction on x. □

Example 28 (Application of outer normalisation)**.** The normalisation of inequalities of steps 1-4 from `la_generic` algorithm is done simmilarly than Listing 15.

We conclude this section by establishing the correctness of the reification procedure and an example of reconstruction of `lia_generic`. Reification is designed so that evaluating the syntactic representation it produces recovers the original integer expression.

Theorem 9 (Correctness of reification)**.** For all integers x and all environments ℓ, if

$$(\Uparrow (\, x)) = (g, \ell') \text{ then } \Downarrow (g, \ell') = x.$$

Proof. Direct by induction on x where cases are `ZPos`, `ZNeg` and `Z0`. □

```
1  opaque symbol t2: Prf• ((¬ (3 < x) ∨ ¬ (x = 2)) ∨ ■) {
2      apply ∨ i1;
3      rewrite Zinv_lt_eq;
4      rewrite Z_diff_gt_Z0_eq (− 3) (− x);
5      rewrite Z_diff_eq_Z0_eq (x) 2;
6      rewrite Zgt_le_succ_r_eq ((− 3) - (− x)) 0;
7      rewrite Zmult_ge_compat_eq 1 ((− 3) - (− x)) ((0 + 1));
8      rewrite Zmult_eq_compat_eq (− 1) (x - 2) 0;
9      rewrite imp_eq_or; assume H0 H1;
10     set H0l' ≔ (1 * ((− 3) - (− x))); set H0r' ≔ (1 * (0 + 1));
11     set H1l' ≔ ((− 1) * (x - 2)); set H1r' ≔ ((− 1) * 0);
12     have H1': Prf (H1l' ≥ H1r') { refine Z_eq_implies_ge H1 }; remove H1;
13     set l ≔ H0l' + H1l';
14     set r ≔ H0r' + H1r';
15     have sum : Prf (l ≥ r){
16         apply (Zsum_geq_s H0l' H0r' H1l' H1r' H0 H1');
17     };
18     refine sum _;
19     rewrite left reify_correct l; rewrite left reify_correct r;
20     set l' ≔ reify l; set r' ≔ reify r;
21     rewrite eta_prod l'; rewrite eta_prod r';
22     rewrite left norm_correct (l' 1) (l' 2) _ { refine Ti } { apply Ti };
23 };
```

Listing 16: LIA reconstruction example

To conclude the proof using reflection, both `l` and `r` are reified using the $(\Uparrow)$ function at line 20-21, yielding a pair consisting of a syntactic term and a list of variable values. The correctness of this reification step is ensured by the Theorem 9 (`reify_correct`). The lemma `eta_prod` expresses the η equivalence principle for dependent pairs. It states that any pair $x : a \times b$ is equal to the pair formed by projecting its first and second components. This principle allows one to destruct a pair syntactically into its components while preserving definitional equality. In the context of reification, where a reified expression is represented as a pair (t, ℓ) of a $\mathbb{G}$-term and a context map, `eta_prod` is used to explicitly access the components of the result of $(\Uparrow x)$. Hence we can apply the Theorem 8 (`norm_correct`) at line 26. The final inequality is then reduced to a comparison between the normalized syntactic forms of `l` and `r`, using the theorem `norm_correct`, which guarantees that normalization preserves semantic equivalence. This enables the proof to conclude by purely syntactic comparison of the normalized forms, reducing $\Downarrow[(\Uparrow l)] \geq \Downarrow[(\Uparrow r)]$ to a contradiction.

10.5 RECONSTRUCTION OF LINEAR REAL ARITHMETIC

The reflective reconstruction for Linear Real Arithmetic (**LRA**) follows verbatim the integer case of the previous section (Section 10.4). We only change the ground domain from $\mathbb{Z}$ to $\mathbb{Q}$ defined at Section 7.3, and adapt the typing of lists and constants accordingly. All definitions (reification $\Uparrow$, denotation $\Downarrow$, `merge`, `mergesort`, `remove0`, and `norm`) and all proofs/lemmas carry over *mutatis mutandis*, relying on the same algebraic laws (associativity, commutativity, and distributivity) which hold in the group $\mathbb{Q}$ just as they do in the group $\mathbb{Z}$.

Definition 10.5.1 (Real reflection signature)**.** We replace the integer reflection carrier by its rational counterpart:

$$
\begin{aligned}
&\mathbb{G}_{\mathbb{R}} : \texttt{TYPE}\\
&\mid \texttt{var} : \mathbb{N} \to \mathbb{Q} \to \mathbb{G}_{\mathbb{R}}\\
&\mid \texttt{cst} : \mathbb{Q} \to \mathbb{G}_{\mathbb{R}}\\
&\texttt{rgrp} : \texttt{Set}\\
&\texttt{El rgrp} \hookrightarrow \mathbb{G}_{\mathbb{R}}
\end{aligned}
$$

We continue to write $\ell[\mathtt{i}]$ for $(\texttt{nth}\ \frac{0}{1}\ \mathtt{i}\ \ell)$, now with $\ell : \mathbb{L}\ \texttt{rat}$ and default $\frac{0}{1} \in \mathbb{Q}$ to make indexing total.

REIFICATION AND DENOTATION. The function $\Uparrow_{\mathbb{Q}} : \mathbb{L}\ \texttt{rat} \to \mathbb{Q} \to \texttt{El}(\texttt{list grp} \times \texttt{list rat})$ is obtained from the integer version by replacing integer constants and coefficients with rational counterparts e.g. $\frac{0}{1}$ for 0. The denotation $\Downarrow_{\mathbb{Q}} : \texttt{El}(\texttt{list grp} \times \mathbb{L}\ \texttt{rat}) \to \mathbb{Q}$ is unchanged apart from types: `cst` $(\mathtt{q})$ evaluates to $\mathtt{q} \in \mathbb{Q}$ and $(\texttt{var i q})$ evaluates to $\ell[\mathtt{i}] *_{\mathbb{Q}} \mathtt{q}$ for $\ell : \mathbb{L}\ \texttt{rat}$.

NORMALIZATION PIPELINE. The operators `mul` : $\mathbb{Q} \to \mathbb{L}$ `grp` $\to \mathbb{L}$ `grp`, `merge`, `mergesort`, and `remove0` are identical to the integer case, with coefficients and constants in $\mathbb{Q}$ and the environment of type $\mathbb{L}$ `rat`. The normalization function remains

$$\texttt{norm}\,(x : \mathbb{L}\ \texttt{grp}) \coloneqq \texttt{remove0}\Big(\texttt{mergesort}\ x\Big) \tag{2}$$

cancel and removes neutral elements

sorts a list of `grp`

CORRECTNESS RESULTS. All statements and proofs from Section 10.4 transfer:

- $\Downarrow_{\mathbb{Q}} (\texttt{merge}\ g_1\ g_2, \ell) = \Downarrow_{\mathbb{Q}} (g_1, \ell) +_{\mathbb{Q}} \Downarrow_{\mathbb{Q}} (g_2, \ell)$,
- $\Downarrow_{\mathbb{Q}} (\texttt{mergesort}\ g, \ell) = \Downarrow_{\mathbb{Q}} (g, \ell)$,
- $\Downarrow_{\mathbb{Q}} (\texttt{remove0}\ g, \ell) = \Downarrow_{\mathbb{Q}} (g, \ell)$,
- $\Downarrow_{\mathbb{Q}} (\texttt{norm}\ x, \ell) = \Downarrow_{\mathbb{Q}} (x, \ell)$,
- and *reification correctness*: if $\Uparrow (\ell, x) = (g, \ell')$ then $\Downarrow_{\mathbb{Q}} (g, \ell') = x$.

No new lemmas are required: each proof is the same by type instantiation $\mathbb{Z} \rightsquigarrow \mathbb{Q}$ and $\mathbb{L}$ `int` $\rightsquigarrow \mathbb{L}$ `rat`.

11 TRANSLATION CORRECTNESS

An Alethe proof trace is a non-empty list $[c_1, \ldots, c_n]$ of steps such that c_n derives the empty clause: `(cl)`. Moreover, the justification of every step may only refer to earlier steps, such that the proof trace can be represented as a directed acyclic graph. We recall the definition of a well-formed Alethe proof trace [11, Def. 7.2].

Proving correctness of the translation amounts to showing that any valid Alethe proof trace is translated into a Lambdapi proof of the empty clause: $\texttt{Prf}^{\bullet}\ \blacksquare$.

We first recall that a *valid* Alethe proof (Definition 4.5.4) is a finite trace of proof steps where each clause is justified either by an inference rule or by reference to previously established clauses. Validity requires that the trace is well-formed (Definition 4.5.3), in the sense that the rule applied at each step is among the supported Alethe rules, that all dependencies point to earlier steps in the trace, and that the conclusion clause is exactly the one prescribed by the semantics of the rule. In this way, a valid proof ensures that every clause in the trace is correctly derived from earlier clauses or assumptions, so that the final clause faithfully represents the logical consequence of the whole derivation.

Theorem 10 (Correctness of reconstruction)**.** For any valid Alethe proof trace prefix $P = [c_1, \ldots, c_i]$, if

$$c_i.\Gamma \rhd \varphi_1 \ldots \varphi_m\ (\mathcal{R}\ P)[A]$$

then

$$\vdash_\Sigma c_i : \texttt{Prf}^{\bullet}(\mathcal{E}(\varphi_1) \vee \cdots \vee \mathcal{E}(\varphi_m) \vee \blacksquare) \coloneqq M_i.$$

If M_i is provided it is a proof term inhabiting the type of the translated clause.

Proof. Inductively, assume that the theorem holds for the shorter trace prefix $P = [c_1, \ldots, c_{i-1}]$, which is also a valid proof trace prefix, and consider the rule R that is applied for establishing c_n.

- Case R = `assume`. Then the assertion obviously holds, and no proof term is provided.
- For tautologous and simple deduction rules, the proofs follow a similar approach. We illustrate them with two cases:
  - Case R = `equiv_pos2`. Using the interpretation of clauses as disjunctions (`cl2disj` Definition 7.5.3), the clause

$$\texttt{Prf}^{\bullet}(\neg(\mathcal{E}(\varphi_1) = \mathcal{E}(\varphi_2)) \vee (\neg\mathcal{E}(\varphi_1)) \vee \mathcal{E}(\varphi_2) \vee \blacksquare)$$

corresponds to the disjunction

$$\texttt{Prf}(\neg(\mathcal{E}(\varphi_1) = \mathcal{E}(\varphi_2)) \vee \neg\mathcal{E}(\varphi_1) \vee \mathcal{E}(\varphi_2)).$$

This is classically valid. Indeed, the disjunction rewrites to $\mathcal{E}(\varphi_1) = \mathcal{E}(\varphi_2) \Rightarrow \neg\mathcal{E}(\varphi_1) \vee \mathcal{E}(\varphi_2)$, so we have $\neg\mathcal{E}(\varphi_2) \vee \mathcal{E}(\varphi_2)$ by substituting and then it is a tautology by excluded middle.

– Case `R = cong`. Since `P` is a valid proof trace prefix, the conclusion of the step must be of the form $(f\ t_1 \dots t_m) \approx (f\ u_1 \dots u_m)$ and the justification must refer to previous steps $i_j.\Gamma \rhd t_j \approx u_j$. By assumption, we therefore have in Lambdapi that $\vdash_\Sigma c_j : \texttt{Prf}^\bullet(t_j = u_j)$, and an application of rule $\texttt{cong}_m$ proves the assertion.

The rest of the supported tautologous and simple deductions can be found in Section A.3.

- Case `R = resolution`. The assertion follows from Theorem 3 since the resolution rule is a chain of resolutions of clauses.
- Case `R = subproof`. The assertion follows from Lemma 12, cf. Section 9.5, and the assumption that `P` is a valid Alethe trace prefix.
- Case `R = contraction`. The assertion follows from Theorem 6, since the rule contracts a clause already established in a previous step. By the induction hypothesis, that earlier step is valid, and therefore the contracted clause is also valid under the assumption that `P` is a valid Alethe trace prefix.
- Case `R = bind`. The assertion follows from Lemmas 14 and 13 and the assumption that `P` is a valid Alethe trace prefix.
- Case `R = sko_forall`. Again relying on the assumption that `P` is a valid Alethe trace prefix, the assertion follows from Lemma 11 and the definition of the translation in Figure 16.
- Case `R = la_generic` with the coefficients $[a_1, \dots, a_n]$ given. The assertion that the translated clause:

  $$\texttt{Prf}^\bullet(\neg(\mathcal{E}(\varphi_1^1) \bowtie \mathcal{E}(\varphi_2^1)) \vee \dots \vee \neg(\mathcal{E}(\varphi_1^k) \bowtie \mathcal{E}(\varphi_2^k)) \vee \blacksquare)$$

  with $\bowtie = \{\geq, \leq, <, >, =\}$ leads to a contradiction follows from the normalisation Theorem 8 and reification Theorem 9, which ensures the correctness of the reification of arithmetic expressions in order to prove a contradiction; establishing the normalisation of disjunctions of inequalities and the coefficients $[\mathcal{E}(a_1), \dots, \mathcal{E}(a_n)]$.

The case of `lia_generic` is handled by relying on the Carcara elaboration of linear arithmetic reasoning (see Section 5.1.3).

□

We do not formally assert correctness for the simplification and evaluation proof steps, because the latter rely on transformations that are not entirely known to us, but we provide a best-effort translation for these commands. Further Alethe commands are left for future work.

12 EVALUATION

12.1 BENCHMARK SUITE AND EVALUATION SETTINGS

We evaluate our approach to SMT-proof reconstruction on benchmarks drawn from SMT-LIB and the TLAPS proof assistant. We chose SMT-LIB because it is an actively maintained, community-curated suite used in the SMT competition [17], and because the Carcara checker likewise uses benchmarks drawn from SMT-LIB [3, §4]. We included TLAPS proofs to align with our objective of verifying proofs produced by the TLAPS solver, as outlined in Section 6.5.

Our evaluation targets exactly the fragments supported by our current encoding of uninterpreted functions and linear integer arithmetic (Chapter 9, Chapter 10). Accordingly, we select the SMT-LIB samples `QF_UF`, `UF`, `LIA`, `UFLIA` and `QF_LIA`. These choices ensure that the benchmark suite exercises precisely the reasoning capabilities our reconstruction currently implements. However, we do not yet include `LRA` because the support for real arithmetic (Section 10.5) remains experimental, and we have so far evaluated it only on small illustrative examples.

12.2 BENCHMARK SETTINGS

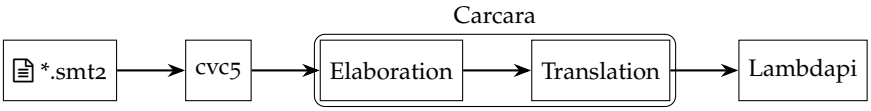


Figure 20: Pipeline of the benchmarks

PIPELINE As illustrated in the Figure 20, each benchmark starts from an SMT-LIB input file (`*.smt2`) that we feed to cvc5 to produce an Alethe proof. The proof is then processed by Carcara in two phases: an *elaboration* pass that checks and enriches the trace, followed by our *translation* pass that reconstructs the proof into Lambdapi terms. We execute *elaboration* and *translation* as two distinct steps so their costs can be measured independently. The resulting Lambdapi file is finally type-checked by the Lambdapi kernel. Each phase is invoked separately and its wall-clock time recorded, allowing us to report elaboration and reconstruction throughput without confounding effects from the other phase. The commands used to run cvc5, Alethe and Lambdapi are as follows:

- Generate an Alethe proof with cvc5:

```
cvc5 --produce-proofs
```

```
  --proof-format-mode=alethe
  --proof-granularity=dsl-rewrite
  --proof-alethe-res-pivots
  --proof-elim-subtypes
  --print-arith-lit-token
  *.smt2
```

- Elaborate with Carcara:

```
carcara elaborate --no-print-with-sharing
  --expand-let-bindings
  -i --log off
  *.alethe
```

- Type-check the reconstructed proof with Lambdapi:

```
lambdapi check -w -v0 *.lp
```

SETTING The experiments presented in this chapter were conducted on the Grid'5000 [9] infrastructure. All experiments were run on a server equipped with:

- 2 CPU AMD EPYC 9754 processor,
- 256 hardware cores (512 threads) at 2.4 GHz,
- 1024 GB RAM,
- 1490 GB SSD,
- configured with a Ubuntu 24.04 LTS.

We selected this configuration to support a multi-stage pipeline that emits numerous intermediate and large final artifacts. The workflow spawns many independent jobs, so a high core count enables broad parallelism to reduce end-to-end runtime. Substantial memory is required to hold large terms, caches, and in-memory indices without excessive swapping. A fast SSD mitigates I/O bottlenecks from frequent reads and writes of intermediate files. In preliminary runs on a smaller machine, we encountered operational limits—including file descriptor exhaustion and sporadic I/O stalls motivating the move to a higher capacity setup. The benchmarks infrastructure repository[1] and the operating system image have been published to ensure reproducibility with a grid'5000 ready for being reproduced by anyone with access to the platform, however, the benchmarks can be reproduced on any platform.

SOFTWARE VERSIONS. Our work is based on the following versions:

- cvc5 version 1.2.2 − dev.277 (commit 8aeaa1938)
- Carcara 1.1.0 (commit 92fd8b4)
- Lambdapi version 3.0.0 (commit 994691d)

1 https://github.com:NotBad4U/benchmarks-reconstruction.git

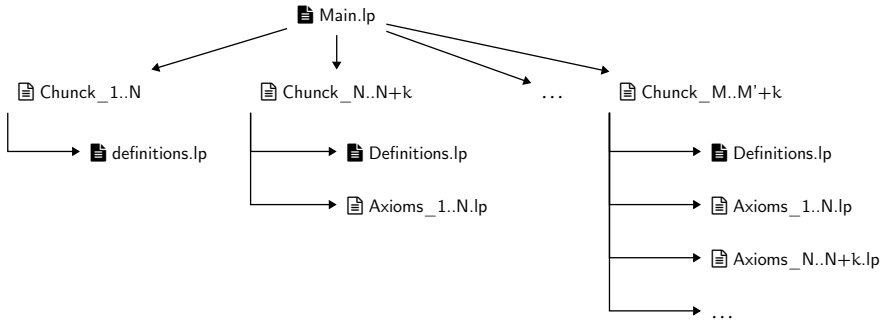


Figure 21: Parallel translation and checking.

MANAGING IDENTIFIERS. Names are an important source of conflicts because the class of identifiers of SMT and Lambdapi language is different. We need to replace identifiers of the SMT input problem that are invalid in the Lambdapi language by valid identifiers. For example, the character '$' is not allowed in Lambdapi identifiers, or some reserved keywords such as "`apply`" in Lambdapi cannot be used as identifiers but are used in some input problems.

12.3 PARALLEL RECONSTRUCTION AND CHECKING OPTIMIZATIONS.

PARALLEL CHECKING. Translating proof traces with several thousand steps can yield Lambdapi files so large that type checking becomes lengthy. Lambdapi is not yet optimized for very large files [23, §4], and checking may take minutes or even hours. To accelerate the verification, we exploit parallelism across CPU cores. The key observation is *locality*: translating step k of Alethe proof does not require knowledge of any future step $k' > k$. The only global constraint is ordering in the final build: the translation that defines the evidence for step k must be available before any file that uses it.

As sketched in Figure 21, we partition the trace into fixed-size chunks of k consecutive steps and translate each chunk independently. For every chunk $[N..M]$ we generate two files: one file with lemmas and their proofs (`Chunk_N..M.lp`), and one with the same statements declared as axioms (`Axioms_N..M.lp`) to be imported by later chunks. Each `Chunk_N..M.lp` depends on `definitions.lp` and on prior `Axioms_*.lp` files as required. A main file (`main.lp`) then includes the chunk files in order.

With this dependency tree in place, the translated proof can be check in parallel using `make -jN`, which checks independent chunk files concurrently. We do not yet perform a final "gluing" pass that eliminates the axioms by substituting the corresponding proved lemmas; however this is unnecessary for checking the validity of the proof. Such a post-processing step would be needed only if one intends to reuse the resulting proof objects for further developments.

TERM SHARING. In addition, to reduce the memory usage and checking time of Lambdapi, we define a constant alias for any closed

Table 6: Summary of benchmark.
The format S − F − T indicates respectively the number of success (S), failed (F) and timeout (T) samples.

Name	Logic	Samples	cvc5	Carcara elaborate
UltimateAutomizer	LIA	153	153 - 0 - 0	146 - 7 - 0
grasshopper	UFLIA	382	330 - 0 - 52	325 - 5 - 0
misc	UFLIA	3	3 - 0 - 0	2 - 1 - 0
simplify	UFLIA	831	797 - 0 - 34	797 - 0 - 0
sledgehammer	UFLIA	1,521	1351 - 0 - 170	1279 - 72 - 1
boogie	UFLIA	877	874 - 0 - 3	853 - 21 - 0
convert	QF_LIA	39	39 - 0 - 0	39 - 0 - 0
mathsat	QF_LIA	100	81 - 0 - 19	59 - 34 - 0
pb2010	QF_LIA	40	22 - 0 - 18	22 - 0 - 0
pidgeons	QF_LIA	19	19 - 0 - 0	19 - 0 - 0
20180326-Bromberger	QF_LIA	175	100 - 0 - 75	100 - 0 - 0
20220307-SMPT	QF_LIA	1,568	1522 - 0 - 46	1520 - 26 - 0
CAV-2009	QF_LIA	85	85 - 0 - 0	85 - 0 - 0
convert	QF_LIA	39	39 - 0 - 0	39 - 0 - 0
cut-lemmas	QF_LIA	93	68 - 0 - 25	68 - 3 - 0
mathsat	QF_LIA	100	81 - 0 - 19	59 - 34 - 0
pb2010	QF_LIA	40	22 - 0 - 18	22 - 0 - 0
20170428-Barrett	UF	1,585	1512 - 0 - 73	1491 - 22 - 0
grasshopper	UF	376	361 - 0 - 15	351 - 10 - 0
sledgehammer	UF	1,403	1269 - 0 - 134	1206 - 68 - 0
20170829-Rodin	QF_UF	20	20 - 0 - 0	20 - 0 - 0
2018-Goel-hwbench	QF_UF	229	222 - 0 - 7	221 - 8 - 0
QG-classification	QF_UF	3,859	3802 - 0 - 57	3718 - 133 - 0
SEQ	QF_UF	44	34 - 0 - 10	34 - 5 - 0
TypeSafe	QF_UF	3	3 - 0 - 0	3 - 0 - 0
eq-diamond	QF_UF	100	100 - 0 - 0	100 - 0 - 0

subterm that is used more than three times in the Alethe proof trace. For example, in the code below, we introduce aliases for frequently occurring subterms:

```
1 symbol p2 ≔ (p b);
2 symbol p4 ≔ (p a);
3 symbol p6 ≔ (p a = p b);
4 opaque symbol t1 : Prf• (¬ p6 ∨ ¬ p4 ∨ p2 ∨ ■) ≔ ...;
5 opaque symbol t2 : Prf• (p_6 ∨ ■) ≔ ...;
```

Here, `p2`, `p4`, and `p6` are used as shorthand for `(p b)`, `(p a)`, and `(p a = p b)`, respectively. This approach minimizes redundancy and improves the efficiency of proof checking. This is similar to the behavior of SMT `:named` attribute of SMT-LIB used by solvers with `--dag-thresh N` dagify common sub expressions appearing more than N times. However, this can make proof reconstruction more difficult because **`rewrite`** tactics may need to unfold these definitions.

12.4 EVALUATION OVER SMT-LIB BENCHMARKS

We evaluated our approach on a benchmark suite drawn from the SMT-LIB repository. This suite comprises six benchmark groups: `LIA`,

`UFLIA`, `QF_UF`, `UF`, `QF_LIA`, each corresponding to a distinct collection of proof obligations expressed in the SMT-LIB format that our encoding supports. In the following, we will only present the results of the 26 samples that have proofs checked by Lambdapi, in order to save space. The complete results for 54 samples, including failed and timed-out samples, are available in Section A.5. After we remove the samples that do not have *unsat* status in the header of the problem, we are left with a total of 13684 samples. Table 6 summarizes the number of samples (column samples) for which cvc5 found a proof and that Carcara could elaborate. The columns `cvc5` and `Carcara` use the format S − F − T indicating respectively the number of success (S), failed (F), and timeout (T) samples. Among the samples instances retained in this experiment, `cvc5` produces a proof for 12,909 out of 13,684 samples, with no reported failures and 775 timeouts in the `cvc5` column. Carcara is then able to elaborate 12,578 of these proofs, while 449 elaborations fail and only 1 times out. In other words, Carcara successfully elaborated about 97.4% of the proofs produced by `cvc5` that reach the elaboration phase. While we report the number of failures and timeouts for both `cvc5` and Carcara, we do not undertake a detailed analysis of their causes. Understanding these behaviours in depth would require SMT solver and Carcara kernel specific expertise that lies beyond the scope of this work, whose focus is on the translation pipeline and proof checking in Lambdapi.

Overall, for the subset of proof obligations whose *translate / check* stage succeeds and that are summarised in Table 7 and Table 8, our translator is able to produce Lambdapi scripts for 9898 of them, that is, about 78% of the elaborated samples. In the table, the column suffixed by `XL` corresponds to the proofs where we apply our *parallel checking* technique to split the proof into several chunks and type check them independently as described in Section 12.3. In Table 8, the groups of columns `translate XL / check XL` refer to two distinct sets of proof obligation than the one of Table 7. The `translate / check` columns in Table 7 report results for obligations whose Alethe proof is at most 10 MB and are processed in the standard mode outputting a single file. By contrast, the `translate XL / check XL` columns in Table 8 report results for obligations whose Alethe proof exceeds 10 MB; for these large proofs we systematically apply our *parallel checking* technique, splitting the proof into chunks and checking them independently. Thus, the XL columns are in addition to the `translate / check` results and do not correspond to cases where `translate / check` has failed. The remaining 2680 failures have several causes. In the arithmetic benchmarks, some proofs cannot be translated because the underlying cvc5 proofs use the SMT-LIB operator `to_real`, so that a problem stated over `LIA` is ultimately solved in the `LRA`; since our current encoding does not support `to_real`, these samples are rejected. Other failures are due to missing support for a small number of Alethe and RARE rules that we have not yet encoded in Lambdapi. We should be able to increase the number of proofs reconstructed with little effort by supporting these rules. Finally, the timeouts we observe, particularly in the parallel checking

Table 7: Translation and checking results.
The format $S - F - T$ indicates respectively the number of success (S), failed (F) and timeout (T) samples.

Name	Logic	Translate	Check
UltimateAutomizer	LIA	90 - 51 - 0	71 - 19 - 0
grasshopper	UFLIA	247 - 0 - 0	14 - 233 - 0
misc	UFLIA	1 - 0 - 0	1 - 0 - 0
simplify	UFLIA	116 - 0 - 0	103 - 7 - 6
sledgehammer	UFLIA	1081 - 52 - 0	897 - 170 - 14
boogie	UFLIA	239 - 0 - 0	0 - 239 - 0
convert	QF_LIA	1 - 0 - 0	1 - 0 - 0
mathsat	QF_LIA	0 - 0 - 0	0 - 0 - 0
pb2010	QF_LIA	20 - 2 - 0	20 - 0 - 0
pidgeons	QF_LIA	19 - 0 - 0	6 - 0 - 13
20180326-Bromberger	QF_LIA	15 - 65 - 0	10 - 4 - 1
20220307-SMPT	QF_LIA	1397 - 7 - 0	702 - 691 - 5
CAV-2009	QF_LIA	16 - 62 - 0	12 - 4 - 0
convert	QF_LIA	1 - 0 - 0	1 - 0 - 0
cut-lemmas	QF_LIA	4 - 20 - 0	1 - 3 - 0
mathsat	QF_LIA	0 - 0 - 0	0 - 0 - 0
pb2010	QF_LIA	20 - 2 - 0	20 - 0 - 0
20170428-Barrett	UF	1419 - 0 - 0	1284 - 121 - 14
grasshopper	UF	301 - 0 - 0	18 - 283 - 0
sledgehammer	UF	1099 - 7 - 0	959 - 117 - 23
20170829-Rodin	QF_UF	20 - 0 - 0	16 - 4 - 0
2018-Goel-hwbench	QF_UF	189 - 3 - 0	167 - 22 - 0
QG-classification	QF_UF	37 - 0 - 0	30 - 0 - 7
SEQ	QF_UF	10 - 0 - 0	4 - 0 - 6
TypeSafe	QF_UF	3 - 0 - 0	3 - 0 - 0
eq-diamond	QF_UF	66 - 0 - 0	15 - 0 - 51

mode, are mainly caused by I/O bottlenecks in our current buffered writer. This configuration produces a large number of output files, and without further tuning, the file writes become the dominant cost, even on our comparatively powerful server. This issue is likely to become even more pronounced when running the toolchain on a laptop with more limited I/O bandwidth, but in such settings it can largely be mitigated in practice by simply reducing the degree of parallelism and avoiding too many simultaneous translations.

Compared to our previous work [32] for first order logic, where we were not able to reconstruct any proof from the SMT-LIB benchmark samples `20170428-Barrett` and `grasshopper`, our current reconstruction pipeline is now able to handle these samples and successfully reconstruct elaborated proofs from them. Among the instances that time out, most of those in the `UF` and `QF_UF` logics are due to the use of the Alethe rule `contraction`. In particular, we uncovered a performance bug in Lambdapi's `rewrite` tactic: when applying our normalisation theorems (e.g. Theorem 6), the tactic becomes extremely slow, whereas directly normalising the same term with the Lambdapi command **`compute`** succeeds, albeit still taking a noticeable amount of time. If we admit the `contraction` rule as an axiom instead of recon-

Table 8: Translation and checking results.
The format S − F − T indicates respectively the number of success (S), failed (F) and timeout (T) samples.

Name	Logic	Translate XL	Check XL
UltimateAutomizer	LIA	0 - 5 - 0	0 - 0 - 0
grasshopper	UFLIA	78 - 0 - 0	0 - 60 - 18
misc	UFLIA	1 - 0 - 0	0 - 1 - 0
simplify	UFLIA	662 - 0 - 19	0 - 486 - 0
sledgehammer	UFLIA	121 - 21 - 4	27 - 63 - 31
boogie	UFLIA	389 - 3 - 222	3 - 403 - 0
convert	QF_LIA	15 - 0 - 23	0 - 6 - 9
mathsat	QF_LIA	10 - 0 - 49	3 - 4 - 3
pb2010	QF_LIA	0 - 0 - 0	0 - 0 - 0
pidgeons	QF_LIA	0 - 0 - 0	0 - 0 - 0
20180326-Bromberger	QF_LIA	0 - 20 - 0	0 - 0 - 0
20220307-SMPT	QF_LIA	83 - 6 - 27	1 - 84 - 16
CAV-2009	QF_LIA	0 - 7 - 0	0 - 0 - 0
convert	QF_LIA	15 - 0 - 23	0 - 0 - 15
cut-lemmas	QF_LIA	0 - 44 - 0	0 - 0 - 0
mathsat	QF_LIA	10 - 0 - 49	3 - 4 - 3
pb2010	QF_LIA	0 - 0 - 0	0 - 0 - 0
20170428-Barrett	UF	69 - 0 - 3	24 - 10 - 35
grasshopper	UF	50 - 0 - 0	0 - 34 - 16
sledgehammer	UF	87 - 9 - 4	36 - 22 - 29
20170829-Rodin	QF_UF	0 - 0 - 0	0 - 0 - 0
2018-Goel-hwbench	QF_UF	5 - 19 - 5	2 - 4 - 1
QG-classification	QF_UF	1842 - 258 - 1581	6 - 103 - 905
SEQ	QF_UF	16 - 0 - 8	0 - 3 - 16
TypeSafe	QF_UF	0 - 0 - 0	0 - 0 - 0
eq-diamond	QF_UF	34 - 0 - 0	0 - 0 - 34

structing them, then a lot of the proofs such as the ones in `eq_diamond` that previously timed out can be all checked successfully.

Similarly, compared to our earlier work on linear integer arithmetic [31], our current reconstruction pipeline is able to handle new benchmark samples such as `mathsat` and successfully reconstruct additional LIA proof obligations. However, we also observe a regression: some proof obligations from the `tokeneer` benchmark family that we previously reconstructed can no longer be handled, because the more recent versions of cvc5 and Alethe that we now use make pervasive use of newly introduced Alethe rules that are not yet supported in our reconstruction pipeline. With a modest amount of additional effort to encode these rules in Lambdapi, we expect to recover these missing results and further improve our overall coverage of LIA benchmarks.

The benchmark Table 9 belows reports, for each benchmark samples, the distribution of checking times over all problems in that samples. The reconstruction for linear arithmetic samples use the outer normalisation technique described in Section 10.3 without the associative commutative optimisation. For every samples, we record in milliseconds the minimum (column `min`), first quartile (column q1), median, third quartile (column q3), and maximum checking time (col-

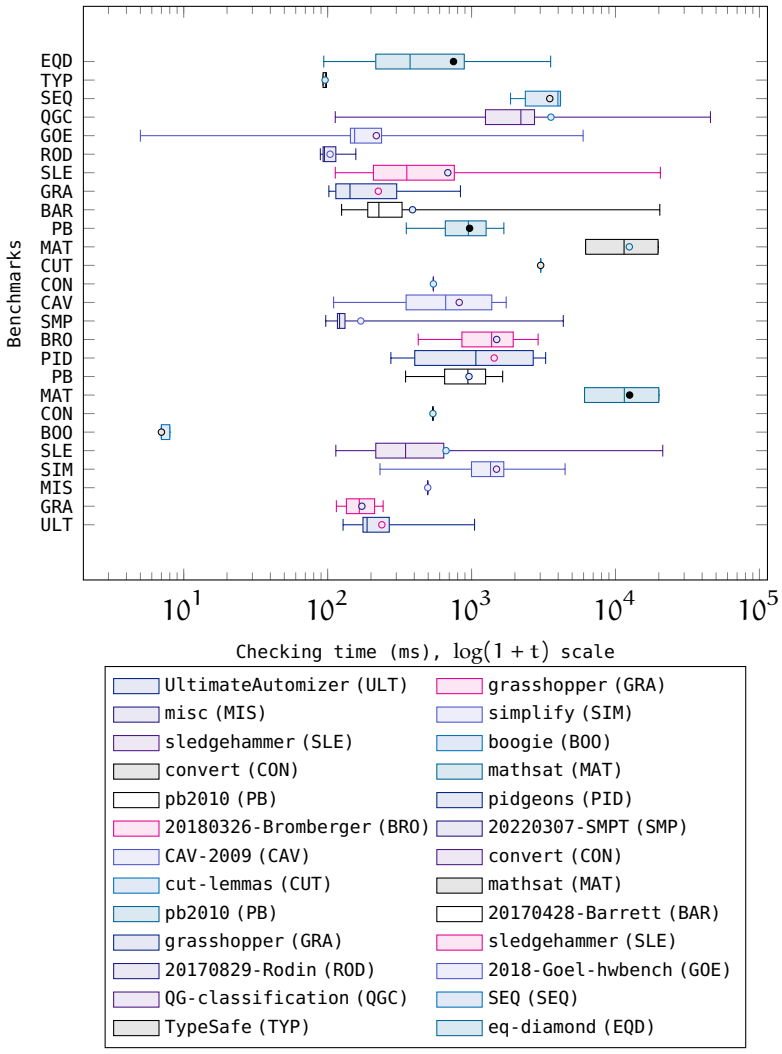


Figure 22: Lambdapi checking time variations between samples

umn `max`), together with the weighted arithmetic mean. The Figure 22 below presents the values of Table 9 with a box plot. In Figure 22, we represent the *mean* by a small colored circle (•) in each box. These summary statistics make it possible to compare not only typical performance (via the median and mean) but also variability and outliers across families, since the maximum values range from a few units up to several tens of thousands, indicating a heavy-tailed distribution. Most benchmark families have median checking times well below one thousand milliseconds, while a few contain significantly harder instances with much larger maxima, highlighting the heterogeneous difficulty of the underlying proof obligations. Among the instances that time out, most of those in the `UF` and `QF_UF` logics are due to the use of the Alethe rule `contraction`, whereas in `LIA` the main culprit is the rule `la_generic`.

When enabling our *inner normalisation* based on the **`associative commutative`** normalisation technique described in Section 10.3, we observe a substantial improvement in checking times across all three benchmark samples. Without this optimisation, the minimum times for `sledgehammer`, `20220307-SMPT`, and `CAV-2009` are 665, 170, and 821

Table 9: Lambdapi checking times (in ms).

name	Mean	Min	Q1	Median	Q3	Max
UltimateAutomizer	238	128	176	188	268	1,049
grasshopper	173	115	135	166	212	243
misc	496	496	496	496	496	496
simplify	1,491	231	999	1,355	1,675	4,466
sledgehammer	665	114	216	348	641	21,301
boogie	7	7	7	7	8	8
convert	539	539	539	539	539	539
mathsat	12,539	6,097	6,097	11,520	19,999	19,999
pb2010	960	348	650	943	1,252	1,645
pidgeons	1,437	275	403	1,071	2,682	3,272
20180326-Bromberger	1,495	426	857	1,379	1,948	2,900
20220307-SMPT	170	97	117	121	132	4,339
CAV-2009	821	110	351	661	1,382	1,740
convert	543	543	543	543	543	543
cut-lemmas	3,021	3,021	3,021	3,021	3,021	3,021
mathsat	12,478	6,205	6,205	11,471	19,758	19,758
pb2010	969	352	657	950	1,262	1,677
20170428-Barrett	389	125	190	227	329	20,323
grasshopper	225	102	114	143	302	837
sledgehammer	684	113	208	355	759	20,511
20170829-Rodin	104	89	93	95	114	157
2018-Goel-hwbench	218	5	144	154	237	5,959
QG-classification	3,561	113	1,249	2,201	2,738	45,695
SEQ	3,495	1,865	2,364	3,982	4,141	4,153
TypeSafe	96	93	93	97	98	98
eq-diamond	749	94	216	374	890	3,544

ms, respectively, whereas with inner AC normalisation they drop to 53, 52, and 54 ms. The upper tails of the distributions also shrink noticeably: for instance, the maximum time for `CAV-2009` decreases from 1740 ms to 683 ms. The quartiles shift towards lower values as well, in particular for `20220307-SMPT`, where times become tightly concentrated around 55 ms. Overall, enabling our inner AC-based normalisation makes proof checking faster. For the moment, however, we only apply this optimisation to a fragment of our benchmarks, since this feature still suffers in the presence of `la_generic` terms with hundreds of uninterpreted variables: the current normalisation mechanism relies on a built-in reordering algorithm in the Lambdapi kernel that is not sufficiently efficient in this setting. We are currently investigating how to modify and improve the implementation of the **`associative commutative`** algorithm. Our latest development and testing suggest that the modifications effectively resolve the issue at hand.

12.5 RECONSTRUCTION OF TLA+ PROOFS

As we described in Section 6.5, one of our main motivations for reconstructing SMT proofs in Lambdapi is to verify proof obligations

Table 10: Lambdapi checking times (ms) with `associative commutative`.

name	Min	Q1	Median	Q3	Max
sledgehammer	53	166	354	552	1,594
20220307-SMPT	52	55	55	56	293
CAV-2009	54	296	403	498	683

Table 11: Summary of benchmark.
The format S − F − T indicates respectively the number of success (S), failed (F) and timeout (T) samples.

Name	Logic	Samples	cvc5	Carcara
Allocator	UF	38	38 - 0 - 0	38 - 0 - 0
EWD840	UFLIA	20	19 - 0 - 1	19 - 0 - 0
Cantor	UF	8	8 - 0 - 0	8 - 0 - 0
Reachability	UFLIA	33	33 - 0 - 0	32 - 1 - 0
BubbleSort	UFLIA	15	14 - 0 - 1	13 - 1 - 0
Bakery	UFLIA	13	13 - 0 - 0	3 - 10 - 0
Boulanger	UFLIA	13	13 - 0 - 0	3 - 10 - 0
LamportMutex	UFLIA	225	201 - 0 - 24	186 - 14 - 1

generated by the TLAPS proof assistant. To evaluate our approach in this context, we collected a benchmark suite of TLA+ proof obligations drawn from examples from the TLA+ community [2]. Similarly to the SMT-LIB benchmarks, Table 11 summarizes the number of samples that cvc5 found a proof for and that Carcara could elaborate.

The TLAPS benchmarks span a variety of typical algorithmic and distributed-system case studies. `Allocator` [70] models a simple resource allocator that enforces exclusive access to a finite pool of resources. `EWD840` [69] encodes Dijkstra's token based termination detection algorithm on a ring of processes. `Cantor` contains proofs of Cantor's theorem stating that the power set of any set has a strictly greater cardinality than the set itself, proving no set can be put into a one-to-one correspondence with its power set. Both `LamportMutex` and `Bakery` (including its priority-based variant `Boulanger`) specify mutual exclusion protocols based on ticket mechanisms and prove both safety and progress properties. Finally, `BubbleSort` is a TLA+ encoding of the classical sorting algorithm with a proof of correctness, and Reachability formalizes a graph reachability algorithm that computes exactly the set of reachable nodes.

In total, the TLAPS suite contributes 365 proof obligations. Among these, cvc5 produces proofs for 339 (92%) obligations, and Carcara successfully elaborates 302 (89%) of them into Alethe. Our pipeline translates 303 proof traces to Lambdapi and we actually attempt to check 293 of the resulting scripts. Overall, 237 proof obligations are successfully reconstructed and checked in Lambdapi, which corresponds to about 65% of all obligations and roughly 70% of the obligations for which cvc5 found a proof. For the `UF` case studies corre-

2 https://github.com/tlaplus/Examples

Table 12: Translation and checking results.
The format S − F − T indicates respectively the number of success (S), failed (F) and timeout (T) samples.

Name	Translate	Check	Translate XL	Check XL
Allocator	34 - 0 - 0	31 - 0 - 0	4 - 0 - 0	4 - 0 - 0
EWD840	19 - 0 - 0	15 - 0 - 0	0 - 0 - 0	0 - 0 - 0
Cantor	8 - 0 - 0	8 - 0 - 0	0 - 0 - 0	0 - 0 - 0
Reachability	29 - 0 - 0	28 - 1 - 0	3 - 0 - 0	1 - 1 - 1
BubbleSort	11 - 0 - 0	11 - 0 - 0	2 - 0 - 0	1 - 0 - 1
Bakery	2 - 0 - 0	2 - 0 - 0	1 - 0 - 0	0 - 0 - 1
Boulanger	2 - 0 - 0	2 - 0 - 0	1 - 0 - 0	0 - 0 - 1
LamportMutex	137 - 0 - 0	109 - 27 - 1	47 - 0 - 3	25 - 16 - 6

Table 13: Lambdapi checking times (in ms).

name	Mean	Min	Q1	Median	Q3	Max
Allocator	505	170	221	349	437	9,463
EWD840	396	94	119	177	697	1,230
Cantor	125	89	94	124	133	205
Reachability	448	89	115	317	673	5,665
BubbleSort	757	95	117	140	979	9,185
Bakery	103	99	96	103	109	107
Boulanger	106	101	98	106	114	111
LamportMutex	849	102	117	269	1,413	15,003

sponding to TLA+ proof using only set theory, reconstruction is now essentially complete: for `Allocator` we reconstruct all proof obligations for which cvc5 and Carcara succeed, and for `EWD840` we reconstruct almost all such obligations, with the few missing ones blocked by current errors or limitations in cvc5 or Carcara rather than in our reconstruction pipeline. Coverage remains lower for the arithmetic `UFLIA` samples, where our current arithmetic support is more restricted.

The checking times presented in Table 11 for the reconstructed TLAPS obligations follow the same qualitative pattern as the SMT-LIB experiments discussed in Section 12.4. Similarly than the previous section, the Section 12.5 represent the times in Table 11 as a box plot. Across all case studies, the weighted mean Lambdapi checking time per obligation lies between about 10ms and 85ms, and the median per benchmark ranges from roughly 10ms to 35ms, so most obligations are reconstructed quickly. We are particularly encouraged that the vast majority of obligations are checked in well under one second, which is important because reconstruction is meant to be integrated into TLAPS and might run while users are interactively developing proofs, so long checking times would directly harm the user experience. All benchmarks nevertheless exhibit a heavy tail: several large arithmetic proof obligations, in particular from LamportMutex, Allocator, BubbleSort, and Reachability, take several seconds to check, with worst-case times around 15s. As in the SMT-LIB setting, these slow and failing cases are almost entirely associated with obligations

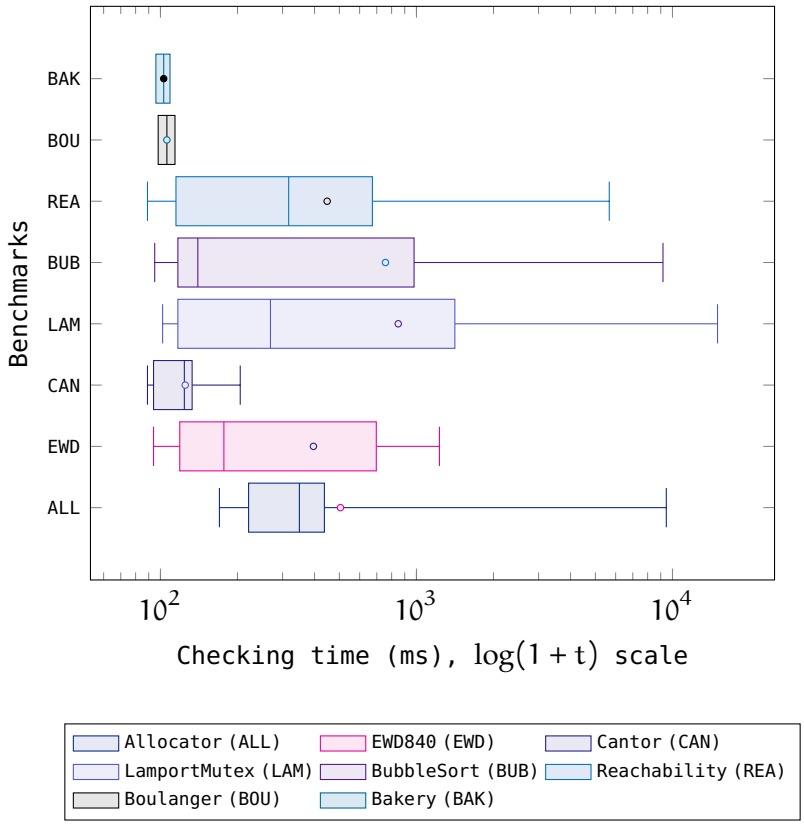


Figure 23: Lambdapi checking time variations between TLAPS samples

that heavily use the `la_generic` and `contraction` rules, which trigger the same weaknesses of our current normalisation mechanism and of the kernel's reordering algorithm, as well as a small number of obligations relying on RARE and Alethe rules that we do not yet support.

Our work revealed that some proof obligations in Allocator initially failed to be reconstructed [32]. In fact, the proof trace found by the SMT solver became incorrect after elaboration by Carcara. The reconstruction failure was due to one of the set theory axioms of TLA$^+$ being incorrectly encoded in SMT [41], and this issue has also been fixed. We also discovered that some proofs produced by cvc5 for the `LamportMutex` samples were incorrect: cvc5 generated proof steps that relied on partially applied functions, whereas the Alethe format does not support currying. For instance, given the membership predicate in set theory (_ ∈ _) encoded in SMT as a binary predicate (`declare-fun Mem (Idv Idv) Bool`), cvc5 would produce terms such as `(! (Mem A) :named M)`, where `Mem` is partially applied to `A`; such terms fall outside the intended setting of Alethe and cannot be faithfully reconstructed.

12.6 EVALUATION SUMMARY

Our evaluation shows that the end-to-end pipeline proposed in this work, from cvc5 through Carcara to Lambdapi, scales to a large and heterogeneous set of SMT-LIB and TLA$^+$ benchmarks in the frag-

ments currently supported by our reconstruction. For almost all proof obligations for which cvc5 and Carcara succeed, the corresponding Lambdapi scripts can be type-checked within a fraction of a second, with the remaining failures explained either by unsupported Alethe or RARE rules or by SMT-LIB features such as `to_real` that we do not yet encode. The timeouts we observe are largely concentrated in `UF/QF_UF` and `LIA` benchmarks that use the Alethe rules `contraction` and `la_generic`, where we hit a performance issue in Lambdapi's **`rewrite`** tactic (while the underlying computations still succeed when using **`compute`**).

The reconstruction effort also uncovered bugs in the TLAPS encoding and even incorrect proofs, which could then be corrected, illustrating the added value of an independent foundational checker. This work was carried out in close collaboration with the Carcara developers: our benchmark suite revealed several bugs in the elaboration phase that we reported, leading to many fixes and thus strengthening the elaborator on which our reconstruction relies. Taken together, these results indicate that Lambdapi can already serve as a practical back-end for independently checking SMT and TLA$^+$ proofs, while also helping us identify missing rules, unsupported features, and performance bottlenecks that guide the next steps in improving both the reconstruction and the checker.

13 CONCLUSION

13.1 CONTRIBUTIONS

This thesis presented a framework to reconstruct Alethe-format SMT proofs into the λΠ-calculus modulo rewriting. The contributions of this work are both conceptual and practical. On the conceptual side, we designed a systematic encoding of SMT logics, including first-order reasoning and linear arithmetic, into a dependently typed setting that includes rewriting logic. This encoding allows us to faithfully represent Alethe inference rules, and to verify them inside the trusted kernel of Lambdapi. On the practical side, we developed a modular extension of Carcara [3] to translate Alethe proof into Lambdapi. This extension leverage elaboration pipeline of Carcara and RARE [78] of cvc5 [12]. Following the approach of *proof logging* [68], this methodology yields independently checkable proof certificates, ensuring that solver outputs can be validated without relying on the internal correctness of the solver itself.

Our approach extends the usability of Alethe by providing a certified reconstruction pipeline that increase trust of SMT proofs. Moreover, we can benefit of the Lambdapi features of import and exporting proof [23] to share SMT proof in order to increase portability, and interoperability across different proof assistants.

ENCODING SMT-LIB IN LAMBDAPI A first contribution is the encoding of the syntax and semantics of SMT-LIB into Lambdapi. We defined universes for sorts, terms, and formulas, together with embeddings that map SMT-LIB constructs into Lambdapi constructions. This encoding supports equality, logical connectives, quantifiers, and arithmetic operators, and it serves as the foundation for the reconstruction of Alethe proofs. By encoding SMT-LIB's many-sorted first-order logic with dependent types and rewriting in Lambdapi, we enable a faithful and extensible representation of SMT solver proof traces that can interact with other formal libraries. Besides, this work also significantly extended the current standard library of Lambdapi.

RECONSTRUCTING FOL We developed a systematic reconstruction of first-order reasoning (logic UF) in Lambdapi. This includes tautological rules, resolution, quantifier instantiation and skolemization, which were encoded as dedicated lemmas and verified within the kernel. Special care was taken to handle Alethe's subproof mechanism and to preserve the structure of natural-deduction style derivations. This reconstruction demonstrates that a wide range of coarse-grained Alethe rules can be faithfully re-expressed in a fine-grained and cer-

tifiable way, thus bridging solver-level reasoning and proof assistant checking.

RECONSTRUCTION VIA REFLECTION For proof rules with complex side conditions or computations (e.g., arithmetic simplifications, AC modulo), we encode the side condition as a reflective decision procedure, formally verified in Lambdapi. The proof rule is then reformulated as a theorem where the side condition appears as an explicit premise.

RECONSTRUCTING ARITHMETIC The reconstruction framework supports proofs in linear arithmetic, both over integers (LIA) and partly over reals (LA). Here we relied on proof by reflection, implementing normalization procedures for arithmetic expressions within Lambdapi and proving their correctness once and for all. This approach allowed us to certify simplification rules such as linear combination, equivalence of normalized forms, and inequality reasoning. In practice, this makes it possible to validate arithmetic reasoning steps from cvc5, including those elaborated by Carcara's `lia_generic` rule, in an efficient and scalable way.

RECONSTRUCTING TLAPS PROOFS A distinctive application of our framework is the reconstruction of proofs produced by the TLA$^+$ Proof System (TLAPS). The proof assistant TLAPS uses external SMT solvers such as veriT and cvc5 as backends to discharge proof obligations that arise in the verification of TLA$^+$ specifications. By translating these Alethe proofs into Lambdapi, our pipeline provides certified validation of the SMT reasoning steps used inside TLAPS. This bridges the gap between high-level system proofs in TLA$^+$ and low-level certified reasoning, and demonstrates that our approach is not limited to isolated SMT benchmarks but also scales to practical verification tasks.

Our empirical evaluation demonstrates that our framework is capable of validating non-trivial SMT traces efficiently, including proofs produced by cvc5 from TLAPS and SMT-LIB benchmarks.

13.2 FUTURE PERSPECTIVES

These findings lay the groundwork for future investigations into:

IMPLEMENTING BITVECTORS A natural extension of this work is the support for the SMT-LIB **BV** theory. Although the Alethe format already mentions bitvectors, only three rules are currently supported, and the remainder of the theory is still under specification. We are considering to encode bitvectors of width w as elements of the finite type $\texttt{Fin}(2^w)$ (Definition 13.2.1), i.e. natural numbers bounded by 2^w. This representation captures the intended semantics of unsigned machine integers and enforces range-safety through dependent typing. Arithmetic operators can be defined by lifting operations on $\mathbb{N}$ mod-

ulo 2^w, while bit-level operators such as conjunction, disjunction, or shifts can be defined either via Boolean decomposition of the underlying representation or through modular arithmetic formulations. An initial work has been started in this direction for extending the Lambdapi standard library with bitvectors. We chose the `Fin` representation over other possible encodings due to its relative efficiency. If, in the future, Lambdapi's kernel adds support for efficient representations of natural numbers e.g., via overriding them with arbitrary-precision arithmetic libraries like GMP[1] bitvectors encoded with `Fin` would likewise benefit from such optimizations. We therefore chose to implement `Fin n`, as we believe it provides the most suitable and future-proof foundation for representing the `BitVec` type, particularly in light of potential enhancements to the kernel.

Definition 13.2.1 (Fin)**.** The type `Fin n` is the type of natural numbers i with the constraint that $i < n$. It is the canonical type with n elements.

$$\texttt{Fin}\ (n : \mathbb{N}) \coloneqq \int (\lambda x, \texttt{istrue}(x < n))$$

$$\texttt{FinMk}\ (n\ m : \mathbb{N})(\texttt{islt} : \texttt{Prf}(m < n)) : \texttt{Fin}\ n$$
$$\coloneqq \texttt{exist}\ (\lambda x, \texttt{istrue}(x < n))\ m\ \texttt{islt}$$

$$\texttt{val}\ [n] : \texttt{Fin}\ n \to \mathbb{N} \coloneqq \lambda e, (e\ s_1)$$

$$\texttt{isLt}\ [n](f : \texttt{Fin}\ n) : \texttt{Prf}((f\ s_1) < n) \coloneqq (f\ s_2)$$

with s_1 and s_2 the first and second projection for dependent pair defined in Definition 2.6.5.

Definition 13.2.2 (Bitvector)**.** We can define bitvectors as natural numbers bounded by powers of two and modular arithmetic representation. We first introduce an infix exponentiation function `_^_` on $\mathbb{N}$. Using this, we define the type `BitVec n` as the finite type `Fin (2 ^n)` (written `Fin` 2^n), representing the set of natural numbers strictly less than 2^n.

$$\hat{}: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \qquad \text{(written infix)}$$
$$0\hat{}1 \hookrightarrow 1$$
$$n\hat{}(m+1) \hookrightarrow n * (n\hat{}m)$$

$$\texttt{BitVec}\ n \coloneqq \texttt{Fin}\ (2^n)$$
$$\texttt{mkBv} : \Pi(n\ m : \mathbb{N})(p : \texttt{Prf}\ (\texttt{istrue}(m < 2^n))) : \texttt{BitVec}\ n$$
$$\coloneqq \texttt{FinMk}\ (2^n)\ m\ p$$

Each bitvector of size n is thus a natural number modulo 2^n, capturing the behaviour of (unsigned) integers with n bits. The function `mkBv` then takes a size $n : \mathbb{N}$, an element m, and a proof p that $m < 2^n$, and returns a bitvector of type `BitVec n`. Internally, this is constructed using `FinMk`, the constructor for the finite type `Fin` (2^n), ensuring that only values within the correct range can inhabit the bitvector type.

1 https://gmplib.org/

EUNOIA COMPATIBILITY Another line of work is to extend our Carcara module to support Alethe proof trace expressed in Eunoia syntax. Because the syntaxes of Alethe and Eunoia are closely related, this adaptation appears feasible and would provide a bridge between the two formats, ensuring compatibility with SMT 3 and promoting interoperability across proof exchange ecosystems.

EXPORTING PROOF TO ROCQ At present, Lambdapi can only translate its λ-terms into Coq, which allows a direct re-use of proof objects inside the Rocq ecosystem. However, this translation only captures the final proof terms and does not reflect the structure of the tactics used in their construction. A complementary approach would therefore be to translate Lambdapi tactics into equivalent Rocq tactics or to define new Ltac tactics mirroring their behaviour. This would preserve higher-level proof strategies and improve readability and maintainability.

Another promising direction arises from the recent experimental addition of user-defined rewrite rules in Rocq [66]. In earlier versions, rewrite rules had to be simulated through `Fixpoint` definitions, which imposed syntactic guards restricting recursion to structural cases. This made it impossible to directly express nonstructural or general recursion, as illustrated by our reflective implementation of merge sort for `lia_generic`. With Rocq's new native support for rewriting rules, it becomes feasible to export Lambdapi rewrite rules directly, avoiding the limitations of `Fixpoint` and enabling a faithful transfer of more complex proof constructions. This opens the path to a deeper integration between Lambdapi and Rocq, where proofs can be shared not only as fully elaborated terms but also as structured proof developments making use of rewriting.

INTEGRATE CARCARA IN TLA$^+$ TOOLBOX We plan to integrate our reconstruction approach within the TLA$^+$ Toolbox, with the aim to automatically reconstruct proof of the SMT backend in Lambdapi.

SHARING TLA$^+$ THEOREMS The work presented in [52] encodes the logic and set theory of Event-B in Lambdapi, enabling proof checking within a type-theoretical framework. Event-B is a formal method for system-level modelling and verification, based on sorted set theory and first-order logic. It shares key similarities with TLA$^+$, particularly in its use of set-theoretical constructs, and its emphasis on proof obligations for correctness. Given this proximity, as Figure 24 suggests, a promising direction for future work would be to explore translations between TLA$^+$ and Event-B, possibly via a common intermediate representation in Lambdapi. One possible approach to enable this transfer could build on the *parametricity* [82, 99] based methodology developed in [96], which provides a framework for connecting two theories encoded in Lambdapi. Such a translation path could allow one to leverage verification tools from both ecosystems while maintaining a shared logical foundation.

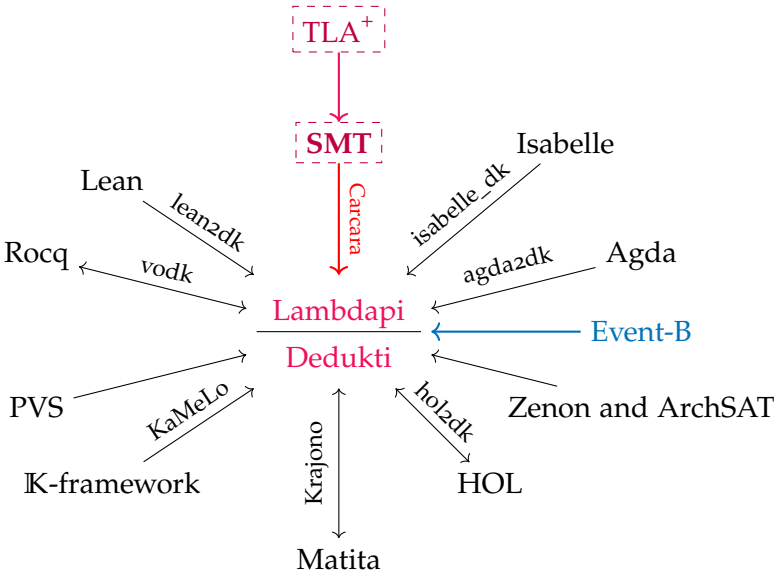


Figure 24: Verifying TLAPS proof with Lambdapi.

LAMBDAPI HAMMER This work lays the foundation for a future *hammer* in Lambdapi similar to CoqHammer or Sledgehammer, to benefit from automatic proof reconstruction, assuming proper encoding of Lambdapi symbols in SMT. A similar strategy to that employed by CoqHammer [39, 40] could be pursued to realise this perspective.

13.3 SUMMARY

In summary, this work establishes Lambdapi as a suitable backend for certifying SMT proofs and contributes a significant step towards making Alethe a widely usable standard for proof exchange in the formal methods community. It opens the way for future extensions, in particular the support for additional theories such as bit-vectors and arrays, as well as tighter integration with interactive proof assistants. Ultimately, the methodology proposed here strengthens the bridge between automated reasoning and interactive theorem proving, fostering greater confidence in the results of modern SMT solvers.

INDEX

⇑, 111
(_ , _), 17
*ℚ, 77
+ℚ, 77
=, 15
$\longrightarrow_{\Sigma}^{AC}$, 112
□, 18
□𝕍, 19
⇓ℚ, 121
⇒, 16
Σ, 10
$\Theta^{\mathcal{S}}$, 27
$\Theta^{\mathcal{X}}$, 27
⇑ℚ, 121
≈, 26
⊥, 16
⇓ _, 100
⇓ _, 116
≐, 74
⊕, 110
⇓G, 117
⇓, 111
𝔹, 18
ℕ, 18
ℚ, 75
bool, 18
cl2disj, 80
cl, 104
cst, 115
false, 18
frac, 75
list, 18
mul, 116
propS, 99
rat, 75
rgrp, 121
true, 18
var, 115
ϵ, 71
ϵ_d, 71
ϵ_i, 71
$\equiv_{\beta\Sigma}$, 11
$\equiv_{\Sigma}$, 11
∃, 16
∀, 16
∫, 18
KIND, 10
Vec, 19
∧, 16
⇝, 15
$\longrightarrow_{\beta}^{*}$, 11
$\longrightarrow_{\Sigma}^{*}$, 11
$\longrightarrow_{\beta\Sigma}^{*}$, 11
$\longrightarrow_{\beta\Sigma}$, 11
∨, 16
λΠ/ ≡, 9
𝔾, 110, 115
$\mathbb{G}_{\mathbb{R}}$, 121
𝕃, 18
ℙ, 72
ℤ, 72
::, 18
::𝕍, 19
𝒞, 104
𝒫, 99
Z0, 72
ZNeg, 72
ZPos, 72
$\longrightarrow_{\Sigma/AC}$, 112
⊨, 30
¬, 16
■, 79
++, 79
$\rightarrow_{\mathbb{Z}}$, 75
$\rightarrow_{\mathbb{P}}$, 75
↪, 10
(⇑ _) , 100, 116
$\rightarrow_{\delta}$, 10
/, 75
Prf•, 80
Clause, 79
H, 72
I, 72
O, 72
comp, 73
cst, 110
em, 72
grp, 110
int, 72
mul, 110

nat, 72
opp, 110
pos, 72
prop_ext, 72
var, 110
×, 17
⊤, 16
Comp, 73
$\twoheadrightarrow^{AC}$, 21
TYPE, 9
⊢, 11, 27
∀, 79
s_1, 19
s_2, 19
1, 17
2, 17
ite, 78
coerce, 22
distinct, 77
headcl, 89
Prop, 15
Set, 15
El, 15
Prf, 15

CDCL, 33

judgment, 11

LF, 9

0, 15, 17

small types, 15

Tarski universes, 15

well-formed, 11
well-type, 11

Part III

APPENDIX

A APPENDIX

A.1 CONFLUENCE OF THE REWRITING RULES OF INTEGERS AND POSITIVE BINARY NUMBER

The rules presented below represent the relations $\rightarrow_{\mathbb{Z}}$ and $\rightarrow_{\mathbb{P}}$ encoded in the TRS[1] format accepted by the [102] tool. These rules can be used to rerun the tool in order to verify the confluence property.

```
(VAR
  a: Z
  b: Z
  x : P
  q : P
  y : P
)
(RULES
  ~(Z0) -> Z0
  ~(Zpos(p)) -> Zneg(p)
  ~(Zneg(p)) -> Zpos(p)
  ~(~(a)) -> a
  ~(add(a,b)) -> add(~(a), ~(b))

  double(Z0) -> Z0
  double(Zpos(p)) -> Zpos(O(p))
  double(Zneg(p)) -> Zneg(O(p))

  succ_double(Z0) -> Zpos(H)
  succ_double(Zpos(p)) -> Zpos(I(p))
  succ_double(Zneg(p)) -> Zneg(pos_pred_double(p))

  pred_double(Z0) -> Zneg(H)
  pred_double(Zpos(p)) -> Zpos(pos_pred_double(p))
  pred_double(Zneg(p)) -> Zneg(I(p))

  sub(I(p), I(q)) -> double(sub(p, q))
  sub(I(p), O(q)) -> succ_double(sub(p, q))
  sub(I(p), H) -> Zpos(O(p))
  sub(O(p), I(q)) -> pred_double(sub(p, q))
  sub(O(p), O(q)) -> double(sub(p, q))
  sub(O(p), H) -> Zpos(pos_pred_double(p))
  sub(H, I(q)) -> Zneg(O(q))
  sub(H, O(q)) -> Zneg(pos_pred_double(q))
  sub(H, H) -> Z0

  +(Z0,a) -> a
  +(a,Z0) -> a
  +(Zpos(x), Zpos(y)) -> Zpos(add(x, y))
  +(Zpos(x), Zneg(y)) -> sub(x, y)
  +(Zneg(x), Zpos(y)) -> sub(y, x)
  +(Zneg(x), Zneg(y)) -> Zneg(add(x, y))

  mult(Z0, a) -> Z0
  mult(a, Z0) -> Z0
  mult(Zpos(x), Zpos(y)) -> Zpos(mul(x, y))
  mult(Zpos(x), Zneg(y)) -> Zneg(mul(x, y))
  mult(Zneg(x), Zpos(y)) -> Zneg(mul(x, y))
  mult(Zneg(x), Zneg(y)) -> Zpos(mul(x, y))
```

1 http://www.lri.fr/~marche/tpdb/format.html

```
  succ(I(x)) -> O(succ(x))
  succ(O(x)) -> I(x)
  succ(H) -> O(H)
  add(I(x), I(q)) -> O(addcarry(x, q))
  add(I(x), O(q)) -> I(add(x, q))
  add(O(x), I(q)) -> I(add(x, q))
  add(O(x), O(q)) -> O(add(x, q))
  add(x, H) -> succ(x)
  add(H, y) -> succ(y)

  addcarry(I(x), I(q)) -> I(addcarry(x, q))
  addcarry(I(x), O(q)) -> O(addcarry(x, q))
  addcarry(O(x), I(q)) -> O(addcarry(x, q))
  addcarry(O(x), O(q)) -> I(add(x, q))
  addcarry(x, H) -> add(x, O(H))
  addcarry(H, y) -> add(O(H), y)

  pos_pred_double(I(x)) -> I(O(x))
  pos_pred_double(O(x)) -> I(pos_pred_double(x))
  pos_pred_double(H) -> H

  mul(I(x), y) -> add(x, O(mul(x,y)))
  mul(O(x), y) -> O(mul(x, y))
  mul(H, y) -> y
)
```

Listing 17: Rewriting rule of $\mathbb{Z}$ and $\mathbb{P}$ in the TRS format

A.2 ARITHMETIC DEFINITIONS

Definition A.2.1 (Integer arithmetic functions)**.**

$$
\begin{aligned}
&+ : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}\\
&\mathtt{Z0} + y \hookrightarrow y\\
&x + \mathtt{Z0} \hookrightarrow x\\
&(\mathtt{Zpos}\ x) + (\mathtt{Zpos}\ y) \hookrightarrow (\mathtt{ZPos}\ (\mathtt{add}\ x\ y))\\
&(\mathtt{Zpos}\ x) + (\mathtt{Zneg}\ y) \hookrightarrow (\mathtt{sub}\ x\ y)\\
&(\mathtt{Zneg}\ x) + (\mathtt{Zpos}\ y) \hookrightarrow (\mathtt{sub}\ y\ x)\\
&(\mathtt{Zneg}\ x) + (\mathtt{Zneg}\ y) \hookrightarrow \mathtt{ZNeg}\ (\mathtt{add}\ x\ y)
\end{aligned}
$$

$$
\begin{aligned}
&\mathtt{succ} : \mathbb{P} \to \mathbb{P}\\
&\mathtt{succ}\ (\mathtt{I}\ x) \hookrightarrow \mathtt{O}(\mathtt{succ}\ x)\\
&\mathtt{succ}\ (\mathtt{O}\ x) \hookrightarrow \mathtt{I}(\mathtt{succ}\ x)\\
&\mathtt{succ}\ (\mathtt{H}) \hookrightarrow \mathtt{OH}
\end{aligned}
$$

$$
\begin{aligned}
&\mathtt{add} : \mathbb{P} \to \mathbb{P} \to \mathbb{P}\\
&\mathtt{add}\ (\mathtt{I}\ x)\ (\mathtt{I}\ q) \hookrightarrow \mathtt{O}\ (\mathtt{addc}\ x\ q)\\
&\mathtt{add}\ (\mathtt{I}\ x)\ (\mathtt{O}\ q) \hookrightarrow \mathtt{I}\ (\mathtt{add}\ x\ q)\\
&\mathtt{add}\ (\mathtt{O}\ x)\ (\mathtt{I}\ q) \hookrightarrow \mathtt{I}\ (\mathtt{add}\ x\ q)\\
&\mathtt{add}\ (\mathtt{O}\ x)\ (\mathtt{O}\ q) \hookrightarrow \mathtt{O}\ (\mathtt{add}\ x\ q)\\
&\mathtt{add}\ x\ \mathtt{H} \hookrightarrow \mathtt{succ}\ x\\
&\mathtt{add}\ \mathtt{H}\ y \hookrightarrow \mathtt{succ}\ y
\end{aligned}
$$

$$
\begin{aligned}
&\mathtt{mul} : \mathbb{P} \to \mathbb{P} \to \mathbb{P}\\
&\mathtt{mul}\ \mathtt{H}\ y \hookrightarrow y\\
&\mathtt{mul}\ y\ \mathtt{H} \hookrightarrow y\\
&\mathtt{mul}\ (\mathtt{O}\ x)\ y \hookrightarrow \mathtt{O}\ (\mathtt{mul}\ x\ y)\\
&\mathtt{mul}\ (\mathtt{I}\ x)\ y \hookrightarrow\\
&\quad \mathtt{add}\ y\ (\mathtt{O}\ (\mathtt{mul}\ x\ y))
\end{aligned}
$$

```
addc : ℙ → ℙ → ℙ
addc (I x) (I q) ↪ I (addc x q)
addc (I x) (O q) ↪ O (addc x q)
addc (O x) (I q) ↪ O (addc x q)
addc (O x) (O q) ↪ I (add x q)
addc x H ↪ add x (O H)
addc H y ↪ add (O H) y
```

```
* : ℤ → ℤ → ℤ
Z0 * _ ↪ Z0
_ * Z0 ↪ Z0
ZPos x * ZPos y ↪ ZPos(mul x y)
ZPos x * ZNeg y ↪ ZNeg(mul x y)
ZNeg x * ZPos y ↪ ZNeg(mul x y)
ZNeg x * ZNeg y ↪ ZPos(mul x y)
```

Definition A.2.2 (Complete definition of sub function)**.**

```
sub : ℙ → ℙ → ℤ
sub (I p) (I q) ↪ double (sub p q)
sub (I p) (O q) ↪ succ_double (sub p q)
sub (I p) H ↪ Zpos (O p)
sub (O p) (I q) ↪ pred_double (sub p q)
sub (O p) (O q) ↪ double (sub p q)
sub (O p) H ↪ Zpos (pos_pred_double p)
sub H (I q) ↪ Zneg (O q)
sub H (O q) ↪ Zneg (pos_pred_double q)
sub H H ↪ Z0

pos_pred_double : ℙ → ℙ
pos_pred_double (I x) ↪ I (O x)
pos_pred_double (O x) ↪ I (pos_pred_double x)
pos_pred_double H ↪ H

double : ℤ → ℤ
double Z0 ↪ Z0
double (Zpos p) ↪ Zpos (O p)
double (Zneg p) ↪ Zneg (O p)

succ_double : ℤ → ℤ
succ_double Z0 ↪ Zpos H
succ_double (Zpos p) ↪ Zpos (I p)
succ_double (Zneg p) ↪ Zneg (pos_pred_double p)

pred_double : ℤ → ℤ
pred_double Z0 ↪ Zneg H
pred_double (Zpos p) ↪ Zpos (pos_pred_double p)
pred_double (Zneg p) ↪ Zneg (I p)
```

Definition A.2.3 (cmp)**.**

```
cmp_acc : ℙ → Comp → ℙ → Comp

cmp_acc (I x) c (I q) ↪ cmp_acc x c q
cmp_acc (I x) _ (O q) ↪ cmp_acc x Gt q
cmp_acc (I _) _ H ↪ Gt
cmp_acc (O x) _ (I q) ↪ cmp_acc x Lt q
cmp_acc (O x) c (O q) ↪ cmp_acc x c q
cmp_acc (O _) _ H ↪ Gt
cmp_acc H _ (I _) ↪ Lt
cmp_acc H _ (O _) ↪ Lt
cmp_acc H c H ↪ c

cmp x y ≔ cmp_acc x Eq y
```

A.3 SUPPORTED ALETHE RULES

ac_simp	and	and_neg
and_pos	assume	bind
contraction	distinct_elim	eq_congruent
eq_congruent_pred	eq_reflexive	eq_symmetric
eq_transitive	equiv_neg1	equiv_neg2
equiv_pos1	equiv_pos2	equiv_simplify
false	forall_inst	implies_neg1
implies_neg2	implies_pos	implies_simplify
ite_neg1	ite_neg2	ite_pos1
ite_pos2	ite_simplify	la_generic
lia_generic	not_and	not_not
not_or	not_simplify	not_symm
or	or_neg	or_pos
refl	reordering	resolution
sko_exist	sko_forall	subproof
symm	th_resolution	true
xor_neg1	xor_neg2	xor_pos1
xor_pos2		

Figure 25: List of Alethe UF rules supported in our reconstruction (alphabetical order).

`th_resolution` and `resolution` denote the same rule. The difference only distinguishes if the rule was introduced by the SAT solver or a theory solver. The Alethe formats include 102 rules (excluding the

rule `hole`), and we currently support 52 of them. However, some rules in the format are used exclusively by veriT, and do not appear in cvc5 proof traces. Among the unsupported rules, 14 are for linear arithmetic and 3 for bitvectors.

`comp_simplify`	`evaluate`
`la_disequality`	`la_generic`
`la_tautology`	`la_totality`
`lia_generic`	

Figure 26: List of LIA/LA rules supported in our reconstruction (alphabetical order).

`arith-elim-gt`	`arith-elim-leq`
`arith-elim-lt`	`arith-geq-norm`
`arith-geq-norm1`	`arith-geq-norm2`
`arith-geq-tighten`	`arith-int-eq-elim`
`arith-leq-norm`	`arith-poly-norm`
`arith-refl-geq`	`arith-refl-leq`
`arith-refl-lt`	`bool-and-de-morgan`
`bool-and-false`	`bool-and-flatten`
`bool-and-true`	`bool-double-not-elim`
`bool-eq-false`	`bool-eq-true`
`bool-impl-elim`	`bool-impl-false1`
`bool-impl-false2`	`bool-impl-true2`
`bool-or-de-morgan`	`bool-or-false`
`bool-or-flatten`	`distinct-binary-elim`
`eq-refl`	`eq-symm`

Figure 27: List of RARE rules supported in our reconstruction (alphabetical order).

A.4 TRANSLATION EXAMPLE INVOLVING MORE ALETHE FEATURES

This section present an extended SMT input problem example below that illustrates a more complex case of proof reconstruction involving several advanced features of the Alethe format and our translation framework. Compared to Listings 1 and 6 examples presented in previous chapters, this instance contains nested *subproofs* (Section 9.5) that demonstrate the use of local assumptions and their discharge within the Alethe structure. It also makes use of the `contraction` rule (Section 9.9) and of RARE simplification steps that normalize arithmetic expressions through rule-based rewriting. Furthermore, the example features *N-ary* rules (Section 9.7), which operate over variable-length connectives.

```
(set-logic UF)

(declare-sort S1 0)
(declare-sort S3 0)
(declare-sort S4 0)
(declare-sort S5 0)
(declare-sort S6 0)
(declare-sort S39 0)
(declare-sort S46 0)

(declare-fun f1 () S1)
(declare-fun f2 () S1)
(declare-fun f5 (S3 S4) S1)
(declare-fun f6 (S5 S3) S3)
(declare-fun f7 () S5)
(declare-fun f8 (S6 S4) S5)
(declare-fun f63 (S39 S3) S4)
(declare-fun f72 () S39)
(declare-fun f80 (S46 S4) S39)
(declare-fun f81 () S46)

(assert (not (= f1 f2)))
(assert (forall ((?v0 S4) (?v1 S3))
    (=
        (f63 (f80 f81 ?v0) ?v1)
        (ite
            (exists ((?v0 S4))
                (and
                    (= (f5 ?v1 ?v0) f1)
                    (forall ((?v3 S4)) (=> (= (f5 ?v1 ?v3) f1) (= ?v3 ?v0)))))
            (f63 f72 ?v1)
            ?v0
        )
    )
))
(assert (forall ((x S1) (y S1)) (= x y)))
(check-sat)
(get-proof)
```

Listing 18: SMT Input Problem

The corresponding proof generated by cvc5 for this problem is shown below in the Alethe format.

```
(assume a0 (not (! (= f1 f2) :named @p_1)))
(assume a1 (forall ((?v0 S4) (?v1 S3))
    (=
        (f63 (f80 f81 ?v0) ?v1)
        (ite
            (exists ((?v0 S4))
                (and
                    (= (f5 ?v1 ?v0) f1)
                    (forall ((?v3 S4)) (=> (= (f5 ?v1 ?v3) f1) (= ?v3 ?v0)))))
            (f63 f72 ?v1)
            ?v0
        )
    )
))
(assume a2 (forall ((x S1) (y S1)) (= x y)))
(step t0 (cl
    (not (!
      (= (!
        (=> (forall ((x S1) (y S1)) (= x y)) (! (= f2 f1) :named @p_4)) :named
@p_5)
        (! (=> (forall ((x S1) (y S1)) (= x y)) @p_1) :named @p_3)) :named @p_6
    ))
    (not @p_5) @p_3) :rule equiv_pos2
)
(step t1 (cl
  (=
```

```
    (forall ((x S1) (y S1)) (= x y))
    (forall ((x S1) (y S1)) (= x y))))
  :rule refl)
(step t2 (cl (= @p_4 @p_1)) :rule rare_rewrite :args ("eq-symm" f2 f1))
(step t3 (cl @p_6) :rule cong :premises (t1 t2))
(step t4 (cl @p_5 (forall ((x S1) (y S1)) (= x y))) :rule implies_neg1)
(anchor :step t5)
(assume t5.a0 (forall ((x S1) (y S1)) (= x y)))
(step t5.t0 (cl (or (! (not (forall ((x S1) (y S1)) (= x y))) :named @p_2) @p_4))
  :rule forall_inst :args (f2 f1))
(step t5.t1 (cl @p_2 @p_4) :rule or :premises (t5.t0))
(step t5.t2 (cl @p_4) :rule resolution
  :premises (t5.t1 t5.a0) :args ((forall ((x S1) (y S1)) (= x y)) false))
(step t5 (cl @p_2 @p_4) :rule subproof :discharge (t5.a0))
(step t6 (cl @p_5 @p_4) :rule resolution
  :premises (t4 t5) :args ((forall ((x S1) (y S1)) (= x y)) true))
(step t7 (cl @p_5 (not @p_4)) :rule implies_neg2)
(step t8 (cl @p_5 @p_5) :rule resolution :premises (t6 t7) :args (@p_4 true))
(step t9 (cl @p_5) :rule contraction :premises (t8))
(step t10 (cl @p_3) :rule resolution
  :premises (t0 t3 t9) :args (@p_6 false @p_5 false))
(step t11 (cl @p_2 @p_1) :rule implies :premises (t10))
(step t12 (cl @p_1 @p_2) :rule reordering :premises (t11))
(step t13 (cl) :rule resolution
  :premises (t12 a2 a0) :args ((forall ((x S1) (y S1)) (= x y)) false @p_1 true))
```

Listing 19: Alethe Proof of Listing 18 Found by cvc5

Finally, the Alethe trace is translated into Lambdapi as shown in the listing below.

```
require open lambdapi.Alethe;
require open lambdapi.Simplify;
require open lambdapi.Rare;
require open lambdapi.Clause;

symbol S1 : Set;
symbol S3 : Set;
symbol S4 : Set;
symbol S5 : Set;
symbol S6 : Set;
symbol S39 : Set;
symbol S46 : Set;

symbol f1 : El S1;
symbol f2 : El S1;
symbol f5 : El (S3 ↪ S4 ↪ S1);
symbol f6 : El (S5 ↪ S3 ↪ S3);
symbol f7 : El S5;
symbol f8 : El (S6 ↪ S4 ↪ S5);
symbol f63 : El (S39 ↪ S3 ↪ S4);
symbol f72 : El S39;
symbol f80 : El (S46 ↪ S4 ↪ S39);
symbol f81 : El S46;

symbol p2 ≔
  ((∀ x : El S1, ∀ y : El S1, x = y) ⇒ f2 = f1) = (∀ x : El S1, ∀ y : El S1, x = y)
    ⇒ f1 = f2;
symbol p3 ≔ ((∀ x : El S1, ∀ y : El S1, x = y) ⇒ f2 = f1);
symbol p4 ≔ (∀ x : El S1, ∀ y : El S1, x = y);
symbol p5 ≔ (f2 = f1);
symbol p6 ≔ (∀ x : El S1, ∀ y : El S1, x = y) ⇒ f1 = f2;
symbol p7 ≔ f1 = f2;
symbol p11 ≔ ¬ (∀ x : El S1, ∀ y : El S1, x = y);

opaque symbol t0 : Prf• (¬ (p3 = p6)  ∨ ¬ p3  ∨ p6 ∨ ■) ≔
begin apply equiv_pos2 end;

opaque symbol t1 : Prf• (p4 = p4 ∨ ■) ≔
begin
  apply ∨ᵢ1;
```

```
  reflexivity
end;

opaque symbol t2 : Prf• (p5 = p7 ∨ ■) ≔
begin
  apply ∨ᵢ1;
  apply eq-symm f2 f1
end;

opaque symbol t3 : Prf• (p2 ∨ ■) ≔
begin apply ∨ᵢ1; apply (cong2 (⟹) (headcl t1) (headcl t2)); end;

opaque symbol t4 : Prf• (p3 ∨ p4 ∨ ■) ≔
begin apply implies_neg1; end;

symbol t5_a0 : Prf• (p4 ∨ ■);

opaque symbol t5_t0 : Prf• ((p11 ∨ p5) ∨ ■) ≔
begin
  apply ∨ᵢ1;
  apply imply_to_or;
  assume H;
  refine (H f2 f1);
end;

opaque symbol t5_t1 : Prf• (p11 ∨ p5 ∨ ■) ≔
begin
  refine (¬ (∀ x : El S1, ∀ y : El S1, x = y) :: f2 = f1 :: □) _;
  simplify;
  eval #repeat_or_id_r ;
  refine (headcl t5_t0);
end;

opaque symbol t5_t2 : Prf• (p5 ∨ ■) ≔
begin
  have t5_t1_t5_a0 : Prf• (p5 ∨ ■) {
    apply resolutionL 0 0 t5_t1 t5_a0 ⊤ᵢ ⊤ᵢ (eq_refl _);
  };
  refine t5_t1_t5_a0;
end;

opaque symbol t5 : Prf• (¬ p4 ∨ p5 ∨ ■) ≔
begin
  apply ∨ᵢ2;
  apply ∨ᵢ1;
  apply (headcl t5_t2);
end;

opaque symbol t6 : Prf• (p3 ∨ p5 ∨ ■) ≔
begin
  have t4_t5 : Prf• (p3  ∨ p5 ∨ ■) {
    apply resolutionR 1 0 t4 t5 ⊤ᵢ ⊤ᵢ (eq_refl _);
  };
  refine t4_t5;
end;

opaque symbol t7 : Prf• (p3 ∨ ¬ p5 ∨ ■) ≔
begin apply implies_neg2; end;

opaque symbol t8 : Prf• (p3 ∨ p3 ∨ ■) ≔
begin
  have t6_t7 : Prf• (p3 ∨ p3 ∨ ■) {
    apply resolutionR 1 1 t6 t7 ⊤ᵢ ⊤ᵢ (eq_refl _);
  };
  refine t6_t7;
end;

opaque symbol t9 : Prf• (p3 ∨ ■) ≔
begin
set ir ≔ reify_cl ((∀ (x : El S1), ∀ (y : El S1), x = y) ⟹ f2 = f1
  ∨ ((∀ (x : El S1), ∀ (y : El S1), x = y)) ⟹ f2 = f1
```

```
112   ∨ ■
113 );
114 have H : Prf (
115   cl2disj (((∀ x : El S1, ∀ y : El S1, x = y) ⇒ f2 = f1)
116     ∨ (∀ x : El S1, ∀ y : El S1, x = y) ⇒ f2 = f1
117     ∨ ■)
118   =
119   cl2disj (((∀ x : El S1, ∀ y : El S1, x = y) ⇒ f2 = f1) ∨ ■)
120 ) {
121   change Prf ((⇑ (ir 2) (ir 1))
122     = (cl2disj ((∀ x : El S1, ∀ y : El S1, x = y) ⇒ f2 = f1 ∨ ■)));
123   rewrite left  contraction_correct;
124   reflexivity;
125 };
126 refine subst_cl H t8;
127 end;
128
129 opaque symbol t10 : Prf• (p6 ∨ ■) ≔
130 begin
131   have t0_t3 : Prf• (¬ p3 ∨ p6 ∨ ■) {
132       apply resolutionL 0 0 t0 t3 ⊤ᵢ ⊤ᵢ (eq_refl _);
133   };
134   have t0_t3_t9 : Prf• (p6 ∨ ■) {
135     apply resolutionL 0 0  t0_t3 t9 ⊤ᵢ ⊤ᵢ (eq_refl _);
136   };
137   refine t0_t3_t9;
138 end;
139
140 opaque symbol t11 : Prf• (p11 ∨ p7 ∨ ■) ≔
141 begin
142   apply implies (headcl t10);
143 end;
144
145 symbol a2 : Prf• (p4 ∨ ■);
146
147 symbol a0 : Prf• (¬ p7 ∨ ■);
148
149 opaque symbol t13 : Prf• ■ ≔
150 begin
151   have t11_a2 : Prf• (p7 ∨ ■) {
152     apply resolutionL 0 0 t11 a2 ⊤ᵢ ⊤ᵢ (eq_refl _);
153   };
154   have t11_a2_a0 : Prf• ■ {
155     apply resolutionR 0 0 t11_a2 a0 ⊤ᵢ ⊤ᵢ (eq_refl _);
156   };
157   refine t11_a2_a0;
158 end;
```

Listing 20: Lambdapi Translation of Listing 19

The module imports `Alethe`, `Simplify`, `Rare`, and `Clause`, which provide the prelude and the encodings for first-order reasoning, clause manipulation, and simplification used throughout the reconstruction. The declarations of sorts `S1, S3, ...` and function symbols `f1, f2, ...` correspond to the SMT symbols defined in Listing 18, while the abbreviations `p2, p3, ...` (lines 25-33) denote shared subterms introduced to factorize repeated expressions and optimize memory usage. Steps `t0`–`t4` apply tautological Alethe rules (e.g., `implies_neg1`), equality reflexivity and symmetry, and congruence (`cong2`), illustrating our translation of first-order logical reasoning and equality handling (Chapter 9). The fragment `t5_t0`–`t5_t2`, spanning lines 56–87, reconstructs the nested subproof from Listing 19 (lines 32–39). A concrete example of a contraction step (Section 9.9) appears in `t9`, starting at line 108. Finally, the `resolutionL` (e.g., line 77) and `resolutionR` (e.g., line 92) lemmas correspond to two instances of Theorem 3: the

former operates when the negated pivot appears on the left of the clause, and the latter when it occurs on the right. Overall, this example demonstrates how our framework integrates advanced Alethe features such as subproofs, contraction, RARE-based simplification, and N-ary reasoning into a unified, checkable proof reconstruction using Lambdapi.

A.5 COMPLETE BENCHMARK RESULTS

We collect the complete benchmark results for all configurations and benchmark suites considered in this work. We start with a global summary of the benchmark samples in Table 14 below. For each step of our pipeline, SMT solving with cvc5, proof elaboration with Carcara, and translation to Lambdapi, we first report the numbers of successful runs, internal errors, and timeouts in Tables 15 and 18. We then present detailed timing measurements for these steps in Tables 16 and 17, followed by box plots that visualize the distribution of checking times for cvc5 and Carcara in Figures 28 and 29, and for the Lambdapi translation on small and large proofs in Figures 30 and 31. To make some plots fit more on the page, we abbreviate benchmark samples using the acronyms defined in the *acronym* column of Table 14. We do not comment on these figures individually, since the qualitative observations about timeouts, reconstruction failures, and performance trends are the same as those already discussed in Chapter 12.

Table 14: Summary of benchmark.

name	acronym	logic	number of proofs
Svcomp2019	SVC	LIA	27
UltimateAutomizer	ULT	LIA	153
psyco	PSY	LIA	50
tptp	TPT	LIA	36
20180326-Bromberger	BRO	QF_LIA	175
20210219-Dartagnan	DAR	QF_LIA	5
20220307-SMPT	SMP	QF_LIA	1,568
20231117-c_inference	CIN	QF_LIA	17
Averest	AVE	QF_LIA	10
CAV_2009_benchmarks	CAV	QF_LIA	85
CIRC	CIR	QF_LIA	33
bofill-scheduling	BOF	QF_LIA	245
calypto	CAL	QF_LIA	13
check	CHK	QF_LIA	4
convert	CON	QF_LIA	39
cut_lemmas	CUT	QF_LIA	93
dillig	DIL	QF_LIA	3
fft	FFT	QF_LIA	4
mathsat	MAT	QF_LIA	100
nec-smt	NEC	QF_LIA	1,524
pb2010	PB2	QF_LIA	40
pidgeons	PID	QF_LIA	19
prime-cone	PRI	QF_LIA	18
rings	RNG	QF_LIA	294
rings_preprocessed	RNP	QF_LIA	294
slacks	SLA	QF_LIA	2
tightrhombus	TIG	QF_LIA	22
wisa	WIS	QF_LIA	2
20170829-Rodin	ROD	QF_UF	20
2018-Goel-hwbench	GOE	QF_UF	229
20190906-CLEARSY	CLE	QF_UF	11
NEQ	NEQ	QF_UF	48
PEQ	PEQ	QF_UF	47
QG-classification	QGC	QF_UF	3,859
SEQ	SEQ	QF_UF	44
TypeSafe	TYP	QF_UF	3
eq_diamond	EQD	QF_UF	100
20170428-Barrett	BAR	UF	1,585
20190906-CLEARSY	CLE	UF	16
grasshopper	GRA	UF	376
misc	MSC	UF	14
sledgehammer	SLH	UF	1,403
boogie	BOG	UFLIA	877
grasshopper	GRA	UFLIA	382
misc	MSC	UFLIA	3
psyco	PSY	UFLIA	2
sexpr	SXP	UFLIA	32
simplify	SIM	UFLIA	831
simplify2	SI2	UFLIA	2,338
sledgehammer	SLH	UFLIA	1,521
spec_sharp	SPS	UFLIA	3
tokeneer	TOK	UFLIA	1,732
tptp	TPT	UFLIA	10

Table 15: cvc5 and Alethe elaboration results.

acronym	logic	number	cvc5	alethe
SVC	LIA	27	27 - 0 - 0	27 - 0 - 0
ULT	LIA	153	153 - 0 - 0	146 - 7 - 0
PSY	LIA	50	49 - 0 - 1	49 - 1 - 0
TPT	LIA	36	36 - 0 - 0	0 - 36 - 0
BRO	QF_LIA	175	100 - 0 - 75	100
DAR	QF_LIA	5	4 - 0 - 1	4 - 1 - 0
SMP	QF_LIA	1,568	1523 - 0 - 45	1521 - 27 - 0
CIN	QF_LIA	17	3 - 0 - 14	3 - 2 - 0
AVE	QF_LIA	10	8 - 0 - 2	7 - 1 - 2
CAV	QF_LIA	85	85 - 0 - 0	85 - 0 - 0
CIR	QF_LIA	33	8 - 0 - 25	8 - 4 - 0
BOF	QF_LIA	245	245 - 0 - 0	245 - 0 - 0
CAL	QF_LIA	13	7 - 0 - 6	7 - 6 - 0
CHK	QF_LIA	4	4 - 0 - 0	4 - 0 - 0
CON	QF_LIA	39	39 - 0 - 0	39 - 0 - 0
CUT	QF_LIA	93	68 - 0 - 25	68 - 3 - 0
DIL	QF_LIA	3	3 - 0 - 0	3 - 0 - 0
FFT	QF_LIA	4	1 - 0 - 3	1 - 1 - 0
MAT	QF_LIA	100	81 - 0 - 19	59 - 34 - 0
NEC	QF_LIA	1,524	0 - 0 - 1524	0 - 72 - 0
PB2	QF_LIA	40	22 - 0 - 18	22 - 0 - 0
PID	QF_LIA	19	19 - 0 - 0	19 - 0 - 0
PRI	QF_LIA	18	18 - 0 - 0	18 - 0 - 0
RNG	QF_LIA	294	77 - 0 - 217	77 - 11 - 0
RNP	QF_LIA	294	90 - 0 - 204	90 - 4 - 0
SLA	QF_LIA	2	0 - 0 - 2	0
TIG	QF_LIA	22	6 - 0 - 16	6 - 0 - 0
WIS	QF_LIA	2	1 - 0 - 1	1 - 1 - 0
ROD	QF_UF	20	20	20
GOE	QF_UF	229	222 - 0 - 7	221 - 8 - 0
CLE	QF_UF	11	11 - 0 - 0	11 - 0 - 0
NEQ	QF_UF	48	37 - 0 - 11	35 - 8 - 1
PEQ	QF_UF	47	24 - 0 - 23	24 - 8 - 0
QGC	QF_UF	3,859	3801 - 0 - 58	3717 - 134 - 0
SEQ	QF_UF	44	34 - 0 - 10	34 - 5 - 0
TYP	QF_UF	3	3 - 0 - 0	3 - 0 - 0
EQD	QF_UF	100	100	100
BAR	UF	1,585	1512 - 0 - 73	1491 - 22 - 0
CLE	UF	16	8 - 8 - 0	8 - 8 - 0
GRA	UF	376	361 - 0 - 15	351 - 10 - 0
MSC	UF	14	14 - 0 - 0	14 - 0 - 0
SLH	UF	1,403	1269 - 0 - 134	1206 - 68 - 0
BOG	UFLIA	877	874 - 0 - 3	853 - 21 - 0
GRA	UFLIA	382	330 - 0 - 52	325 - 5 - 0
MSC	UFLIA	3	3 - 0 - 0	2 - 1 - 0
PSY	UFLIA	2	2 - 0 - 0	0 - 2 - 0
SXP	UFLIA	32	17 - 0 - 15	17 - 4 - 0
SIM	UFLIA	831	797 - 0 - 34	797 - 0 - 0
SI2	UFLIA	2,338	2326 - 0 - 12	2325 - 5 - 1
SLH	UFLIA	1,521	1351 - 0 - 170	1279 - 72 - 1
SPS	UFLIA	3	3 - 0 - 0	3 - 0 - 0
TOK	UFLIA	1,732	1732 - 0 - 0	1689 - 43 - 0
TPT	UFLIA	10	10	0 - 10 - 0

Table 16: Time measurements for cvc5 in seconds.

acronym	mean	min	Q1	median	Q3	max
SVC	0.08	0.04	0.05	0.07	0.09	0.19
ULT	0.04	0.02	0.02	0.04	0.05	0.11
PSY	1.01	0.04	0.08	0.20	0.72	13.24
TPT	0.02	0.02	0.02	0.02	0.03	0.05
BRO	0.13	0.03	0.04	0.07	0.13	7.30
DAR	2.46	1.05	1.28	2.03	5.91	7.18
SMP	0.79	0.02	0.02	0.03	0.06	37.18
CIN	0.74	3.31	3.31	3.75	5.48	5.48
AVE	7.64	2.00	4.12	7.93	10.49	29.70
CAV	0.22	0.02	0.04	0.06	0.11	10.62
CIR	1.81	0.16	0.29	2.20	14.77	31.01
BOF	1.00	0.37	0.76	0.79	0.82	22.55
CAL	1.08	0.40	0.40	3.19	3.21	3.23
CHK	0.04	0.03	0.03	0.03	0.05	0.06
CON	1.34	0.04	0.27	1.25	1.37	18.26
CUT	2.87	0.03	0.06	0.54	5.15	30.08
DIL	3.37	0.20	0.20	0.50	9.40	9.40
FFT	3.68	14.70	14.70	14.70	14.70	14.70
MAT	5.57	0.05	0.95	3.65	11.19	30.53
NEC	0.00	0.00	0.00	0.00	0.00	0.00
PB2	1.80	0.45	2.02	2.90	3.82	9.20
PID	0.06	0.02	0.03	0.05	0.08	0.12
PRI	0.54	0.06	0.15	0.45	0.88	1.36
RNG	1.16	0.04	0.07	0.11	1.14	25.59
RNP	1.26	0.04	0.07	0.11	0.78	18.66
SLA	0.00	0.00	0.00	0.00	0.00	0.00
TIG	0.54	0.06	0.07	0.44	5.37	5.40
WIS	1.07	2.14	2.14	2.14	2.14	2.14
ROD	0.02	0.02	0.02	0.02	0.03	0.03
GOE	1.28	0.02	0.02	0.04	0.13	25.68
CLE	0.09	0.03	0.03	0.03	0.11	0.42
NEQ	2.97	0.17	0.52	1.58	5.07	26.14
PEQ	2.98	0.14	0.76	2.42	8.00	27.45
QGC	2.07	0.03	0.44	0.89	2.89	30.63
SEQ	3.09	0.04	0.12	0.64	2.62	27.42
TYP	0.03	0.02	0.02	0.03	0.03	0.03
EQD	0.16	0.02	0.04	0.07	0.15	0.89
BAR	0.22	0.02	0.07	0.09	0.11	23.99
CLE	0.03	0.02	0.03	0.03	0.05	0.05
GRA	0.52	0.02	0.04	0.05	0.12	22.16
MSC	0.04	0.03	0.03	0.04	0.04	0.05
SLH	0.38	0.02	0.07	0.12	0.21	28.19
BOG	1.05	0.02	0.13	0.57	1.14	21.34
GRA	0.71	0.03	0.05	0.08	0.28	26.59
MSC	2.56	0.03	0.03	0.45	7.20	7.20
PSY	0.03	0.02	0.02	0.03	0.03	0.03
SXP	4.44	0.12	3.47	7.06	13.30	23.46
SIM	0.84	0.05	0.18	0.46	0.83	28.31
SI2	0.88	0.02	0.10	0.24	0.73	30.37
SLH	0.49	0.02	0.05	0.11	0.20	30.36
SPS	0.20	0.04	0.04	0.07	0.50	0.50
TOK	0.04	0.02	0.03	0.03	0.04	0.36
TPT	0.08	0.02	0.03	0.03	0.07	0.41

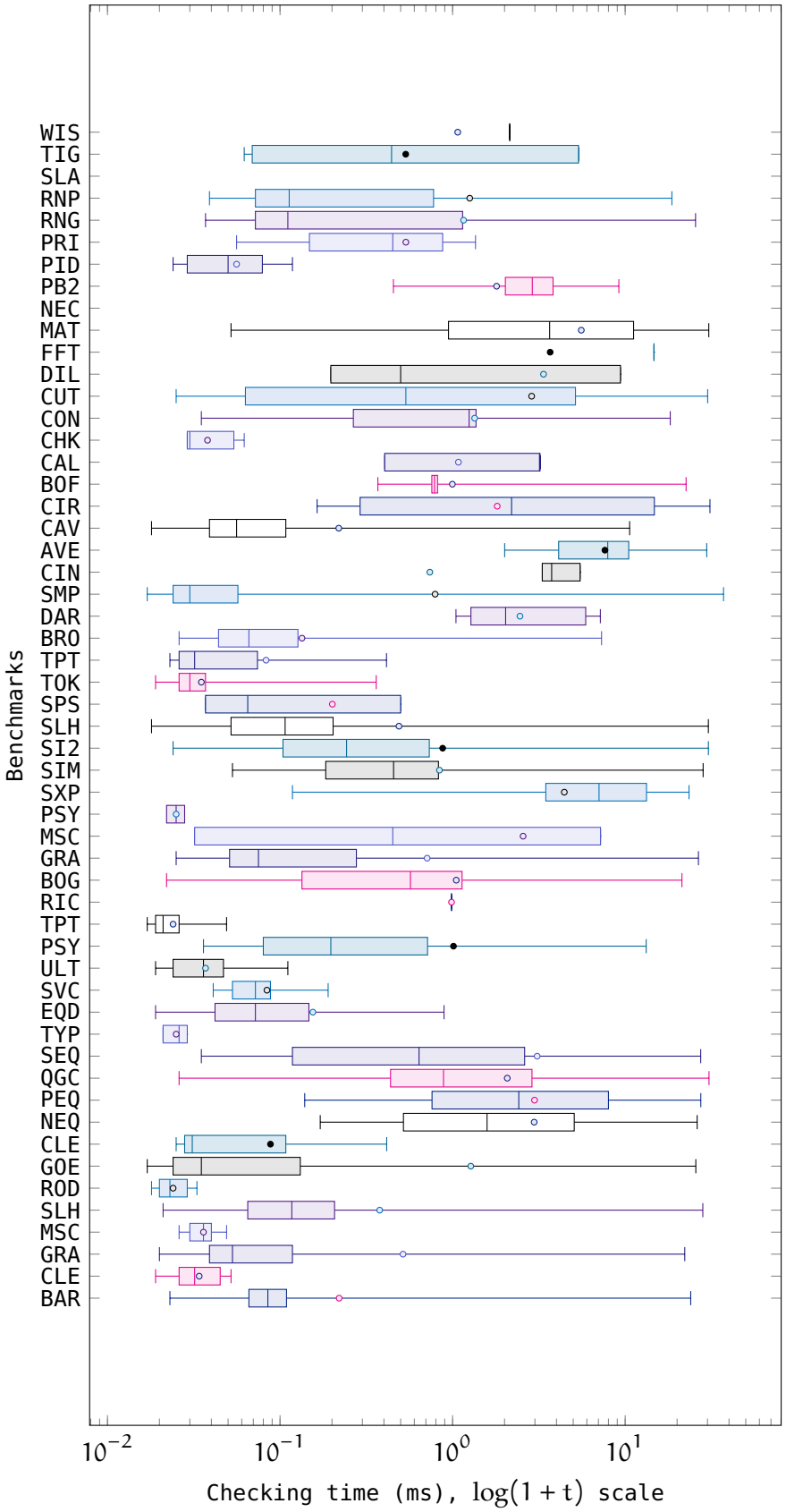


Figure 28: cvc5 times.

Table 17: Time measurements for Carcara elaboration.

acronym	logic	number	mean	min	Q1	median	Q3	max
SVC	LIA	27	0.10	0.03	0.04	0.07	0.11	0.30
ULT	LIA	153	0.03	0.01	0.01	0.02	0.03	0.18
PSY	LIA	50	2.50	0.02	0.08	0.33	1.49	38.01
TPT	LIA	36	0.01	0.00	0.01	0.01	0.01	0.01
BRO	QF_LIA	100	0.06	0.01	0.02	0.04	0.07	0.33
DAR	QF_LIA	5	1.98	0.53	0.59	2.01	3.37	4.41
SMP	QF_LIA	1,548	0.41	0.00	0.01	0.01	0.02	50.20
CIN	QF_LIA	5	0.88	0.01	0.02	0.48	1.93	2.19
AVE	QF_LIA	10	16.13	5.43	11.20	23.42	27.02	32.44
CAV	QF_LIA	85	0.18	0.01	0.02	0.03	0.07	10.07
CIR	QF_LIA	12	9.19	0.01	0.01	0.71	9.01	53.24
BOF	QF_LIA	245	0.82	0.07	0.52	0.70	0.73	43.58
CAL	QF_LIA	13	2.96	0.01	0.01	0.83	8.93	8.99
CHK	QF_LIA	4	0.02	0.01	0.01	0.01	0.04	0.05
CON	QF_LIA	39	3.82	0.02	0.59	3.79	3.89	51.34
CUT	QF_LIA	71	3.16	0.01	0.04	0.56	5.09	18.64
DIL	QF_LIA	3	3.34	0.18	0.18	0.45	9.38	9.38
FFT	QF_LIA	2	17.56	0.03	0.03	17.56	35.09	35.09
MAT	QF_LIA	93	5.85	0.02	0.21	1.31	9.18	30.31
NEC	QF_LIA	72	0.22	0.01	0.07	0.21	0.32	0.93
PB2	QF_LIA	22	0.35	0.07	0.22	0.31	0.41	0.93
PID	QF_LIA	19	0.02	0.01	0.01	0.02	0.03	0.03
PRI	QF_LIA	18	0.86	0.04	0.15	0.66	1.42	2.49
RNG	QF_LIA	88	6.35	0.01	0.04	0.11	1.59	43.53
RNP	QF_LIA	94	5.97	0.01	0.07	0.15	1.15	28.67
SLA	QF_LIA	0	0.00	0.00	0.00	0.00	0.00	0.00
TIG	QF_LIA	6	1.82	0.06	0.06	0.43	4.93	5.01
WIS	QF_LIA	2	1.49	0.05	0.05	1.49	2.94	2.94
ROD	QF_UF	20	0.01	0.01	0.01	0.01	0.01	0.01
GOE	QF_UF	229	0.62	0.00	0.01	0.01	0.05	19.11
CLE	QF_UF	11	0.03	0.01	0.01	0.01	0.03	0.11
NEQ	QF_UF	44	5.95	0.02	0.37	2.23	8.54	38.41
PEQ	QF_UF	32	7.23	0.02	0.12	2.05	10.34	48.22
QGC	QF_UF	3,851	3.14	0.01	0.64	1.32	4.10	45.94
SEQ	QF_UF	39	5.65	0.01	0.05	0.48	2.68	49.03
TYP	QF_UF	3	0.01	0.01	0.01	0.01	0.01	0.01
EQD	QF_UF	100	0.28	0.01	0.03	0.06	0.17	2.34
BAR	UF	1,513	0.05	0.01	0.02	0.02	0.03	8.06
CLE	UF	16	0.02	0.01	0.01	0.02	0.02	0.04
GRA	UF	361	0.06	0.01	0.01	0.02	0.05	2.92
MSC	UF	14	0.02	0.01	0.01	0.02	0.03	0.07
SLH	UF	1,274	0.14	0.00	0.02	0.03	0.05	59.13
BOG	UFLIA	874	1.45	0.01	0.04	0.76	1.43	48.97
GRA	UFLIA	330	0.08	0.01	0.02	0.03	0.07	2.32
MSC	UFLIA	3	4.96	0.02	0.02	1.06	13.79	13.79
PSY	UFLIA	2	0.01	0.01	0.01	0.01	0.01	0.01
SXP	UFLIA	21	2.29	0.02	0.28	2.33	3.26	6.10
SIM	UFLIA	797	0.36	0.02	0.12	0.21	0.43	9.12
SI2	UFLIA	2,331	0.51	0.01	0.04	0.09	0.38	59.22
SLH	UFLIA	1,352	0.09	0.00	0.02	0.03	0.05	4.08
SPS	UFLIA	3	0.28	0.01	0.01	0.05	0.77	0.77
TOK	UFLIA	1,732	0.02	0.00	0.01	0.01	0.02	0.68
TPT	UFLIA	10	0.01	0.01	0.01	0.01	0.01	0.01

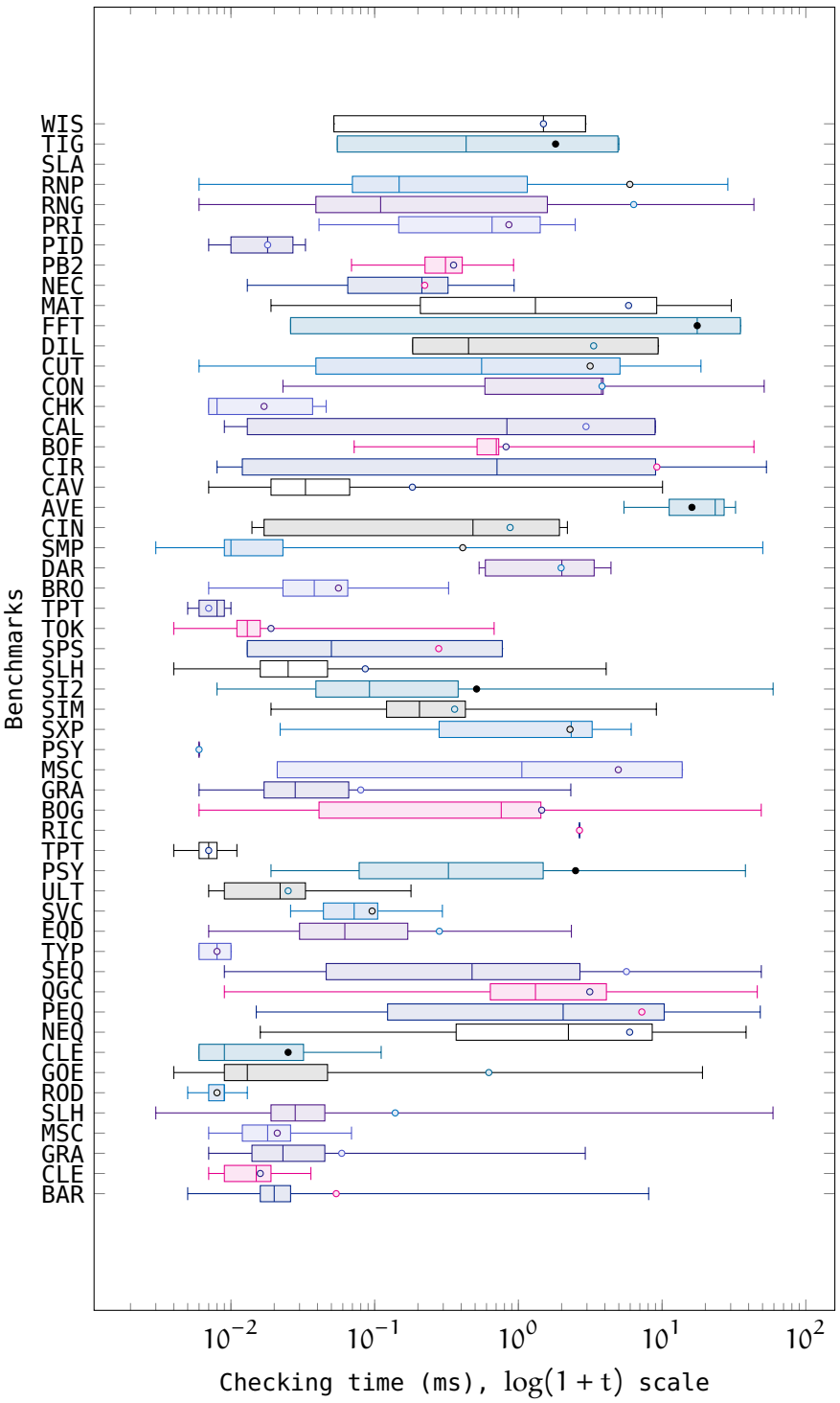


Figure 29: Carcara elaboration times.

Table 18: Translation results.

name	logic	translate	translate XL
Svcomp2019	LIA	0 - 13 - 0	0 - 14 - 0
UltimateAutomizer	LIA	90 - 51 - 0	0 - 5 - 0
psyco	LIA	12 - 0 - 0	36 - 0 - 1
tptp	LIA	0	0
20180326-Bromberger	QF_LIA	15 - 65 - 0	0 - 20 - 0
20210219-Dartagnan	QF_LIA	0	1 - 0 - 3
20220307-SMPT	QF_LIA	1397 - 7 - 0	83 - 6 - 28
20231117-c_inference	QF_LIA	0	3 - 0 - 0
Averest	QF_LIA	0	0 - 0 - 7
CAV_2009_benchmarks	QF_LIA	16 - 62 - 0	0 - 7 - 0
CIRC	QF_LIA	0	0 - 8 - 0
bofill-scheduling	QF_LIA	11 - 0 - 0	233 - 0 - 1
calypto	QF_LIA	0	0 - 7 - 0
check	QF_LIA	0 - 4 - 0	0
convert	QF_LIA	1 - 0 - 0	15 - 0 - 23
cut_lemmas	QF_LIA	4 - 20 - 0	0 - 44 - 0
dillig	QF_LIA	0	0 - 3 - 0
fft	QF_LIA	0	0 - 0 - 1
mathsat	QF_LIA	0	10 - 0 - 49
nec-smt	QF_LIA	0	0
pb2010	QF_LIA	20 - 2 - 0	0
pidgeons	QF_LIA	19 - 0 - 0	0
prime-cone	QF_LIA	0 - 4 - 0	0 - 14 - 0
rings	QF_LIA	28 - 0 - 0	21 - 28 - 0
rings_preprocessed	QF_LIA	29 - 0 - 0	26 - 35 - 0
slacks	QF_LIA	0	0
tightrhombus	QF_LIA	0 - 2 - 0	0 - 4 - 0
wisa	QF_LIA	0	0 - 1 - 0
20170829-Rodin	QF_UF	20	0
2018-Goel-hwbench	QF_UF	189 - 3 - 0	5 - 19 - 5
20190906-CLEARSY	QF_UF	10	1 - 0 - 0
NEQ	QF_UF	3 - 0 - 0	19 - 0 - 13
PEQ	QF_UF	0	14 - 0 - 10
QG-classification	QF_UF	37 - 0 - 0	1840 - 258 - 1582
SEQ	QF_UF	10	16 - 0 - 8
TypeSafe	QF_UF	3 - 0 - 0	0
eq_diamond	QF_UF	66 - 0 - 0	34 - 0 - 0
20170428-Barrett	UF	1419 - 0 - 0	69 - 0 - 3
20190906-CLEARSY	UF	8 - 0 - 0	0
grasshopper	UF	301 - 0 - 0	50
misc	UF	14 - 0 - 0	0
sledgehammer	UF	1099 - 7 - 0	87 - 9 - 4
boogie	UFLIA	239 - 0 - 0	387 - 3 - 224
grasshopper	UFLIA	247 - 0 - 0	78 - 0 - 0
misc	UFLIA	1 - 0 - 0	1 - 0 - 0
psyco	UFLIA	0	0
sexpr	UFLIA	1 - 0 - 0	3 - 12 - 1
simplify	UFLIA	116 - 0 - 0	662 - 0 - 19
simplify2	UFLIA	1037 - 0 - 0	1142 - 1 - 146
sledgehammer	UFLIA	1081 - 52 - 0	121 - 21 - 4
spec_sharp	UFLIA	2 - 0 - 0	1 - 0 - 0
tokeneer	UFLIA	1662 - 2 - 0	25 - 0 - 0
tptp	UFLIA	0	0

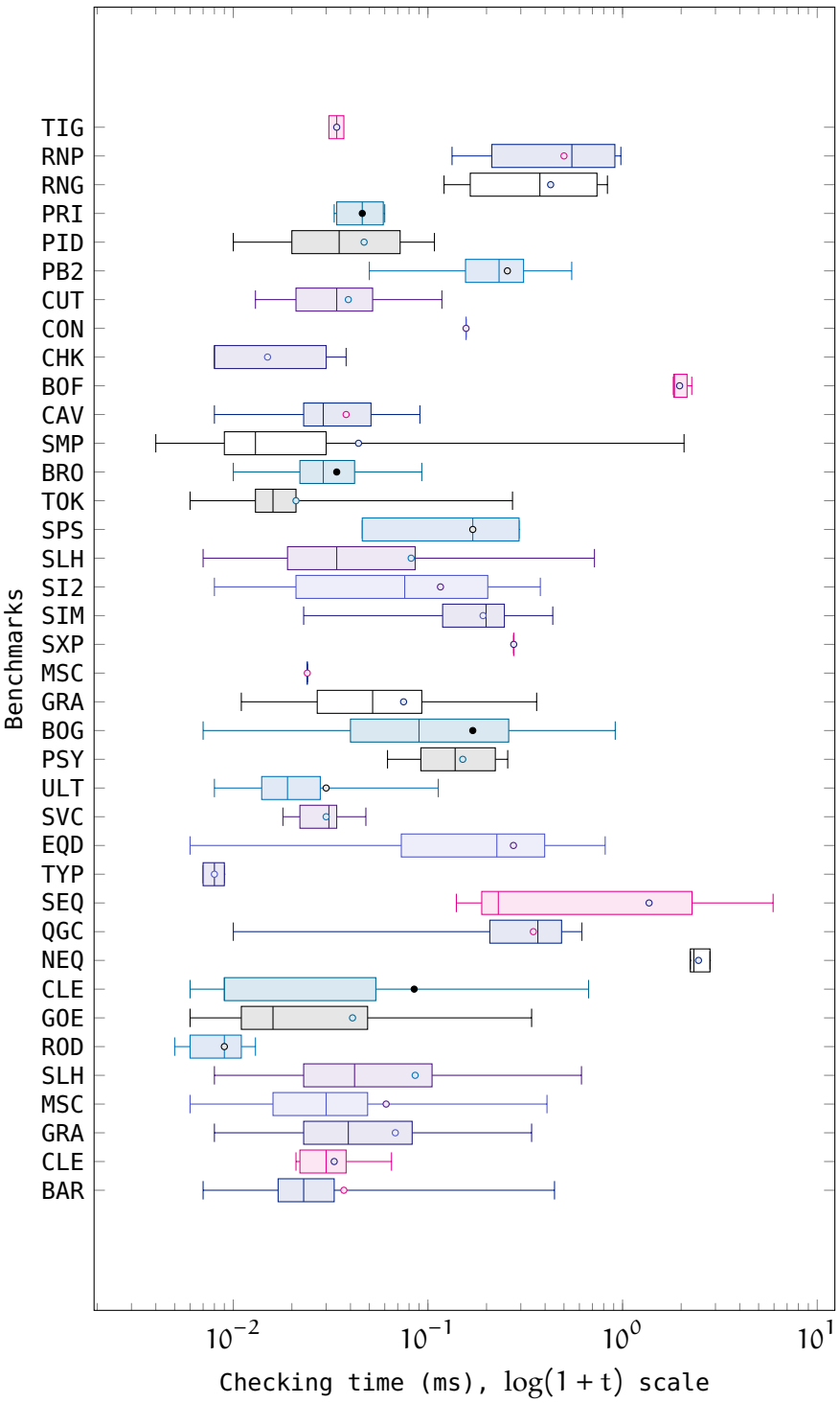


Figure 30: Lambdapi translation times for small problems.

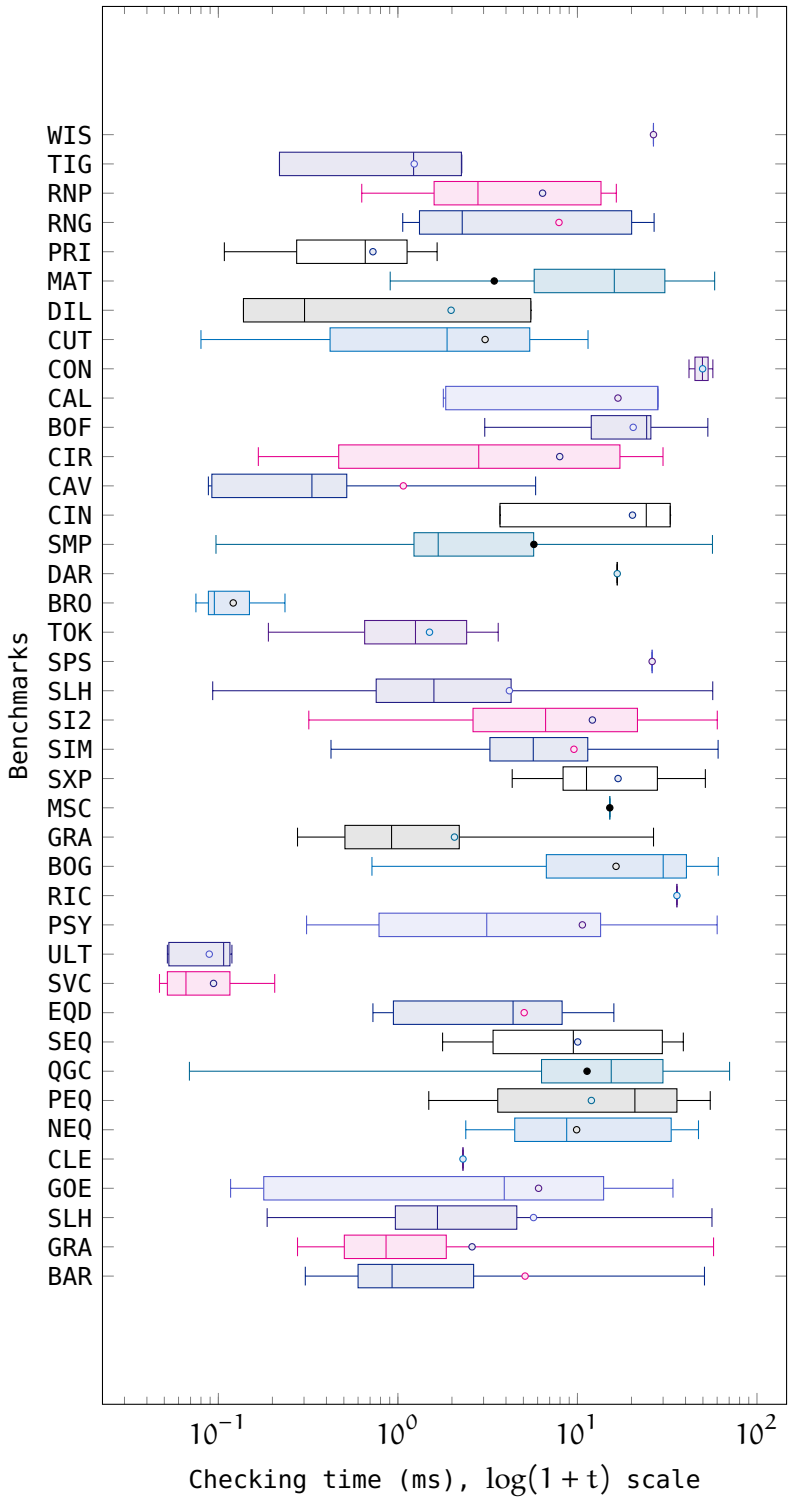


Figure 31: Lambdapi translation times for large proofs.

BIBLIOGRAPHY

[1] J. R. Abrial, M. K. O. Lee, D. S. Neilson, P. N. Scharbach, and I. H. Sørensen. "The B-method." In: *VDM '91 Formal Software Development Methods*. Ed. by Søren Prehn and Hans Toetenel. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 398–405. ISBN: 978-3-540-46456-3.

[2] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. "Rodin: an open toolset for modelling and reasoning in Event-B." In: *International Journal on Software Tools for Technology Transfer* 12.6 (Nov. 2010), pp. 447–466. ISSN: 1433-2787. DOI: 10.1007/s10009-010-0145-y.

[3] Bruno Andreotti, Hanna Lachnitt, and Haniel Barbosa. "Carcara: An Efficient Proof Checker and Elaborator for SMT Proofs in the Alethe Format." In: *Tools and Algorithms for the Construction and Analysis of Systems: 29th International Conference, TACAS 2023*. Cham: Springer-Verlag, 2023, 367–386. ISBN: 978-3-031-30822-2. DOI: 10.1007/978-3-031-30823-9_19.

[4] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Thery, and Benjamin Werner. "A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses." In: *First Intl. Conf. Certified Programs and Proofs (CPP 2011)*. Ed. by Jouannaud, Jean-Pierre, Shao, and Zhong. Vol. 7086. LNCS. Lenting, Taiwan: Springer, 2011, pp. 135–150. DOI: 10.1007/978-3-642-25379-9_12.

[5] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. "Dedukti: a Logical Framework based on the λΠ-Calculus Modulo Theory." 2016. URL: https://inria.hal.science/hal-04281492.

[6] Anne Baanen. "Formalizing Fundamental Algebraic Number Theory." English. PhD-Thesis - Research and graduation internal. Vrije Universiteit Amsterdam, Jan. 2024. DOI: 10.5463/thesis.541.

[7] Matthias Baaz, Uwe Egly, Alexander Leitsch, Jean Goubault-Larrecq, and David Plaisted. "Chapter 5 - Normal Form Transformations." In: *Handbook of Automated Reasoning*. Ed. by Alan Robinson and Andrei Voronkov. Handbook of Automated Reasoning. Amsterdam: North-Holland, 2001, pp. 273–333. ISBN: 978-0-444-50813-3.

[8] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. "Semantic-based Automated Reasoning for AWS Access Policies using SMT." In:

2018 Formal Methods in Computer Aided Design (FMCAD). 2018, pp. 1–9. DOI: 10.23919/FMCAD.2018.8602994.

[9] Daniel Balouek et al. "Adding Virtualization Capabilities to the Grid'5000 Testbed." In: *Cloud Computing and Services Science*. Ed. by Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan. Vol. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, pp. 3–20. ISBN: 978-3-319-04518-4. DOI: 10.1007/978-3-319-04519-1_1.

[10] Haniel Barbosa, Jasmin Christian Blanchette, Mathias Fleury, and Pascal Fontaine. "Scalable Fine-Grained Proofs for Formula Processing." In: *Journal of Automated Reasoning* 64.3 (Mar. 2020), pp. 485–510.

[11] Haniel Barbosa, Mathias Fleury, Pascal Fontaine, and Hans-Jörg Schurr. *The Alethe Proof Format An Evolving Specification and Reference*. https://verit.gitlabpages.uliege.be/alethe/specification.pdf. Mar. 2024.

[12] Haniel Barbosa et al. "cvc5: A Versatile and Industrial-Strength SMT Solver." In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24.

[13] Haniel Barbosa et al. "Generating and Exploiting Automated Reasoning Proof Certificates." In: 66.10 (2023).

[14] Haniel Barbosa et al. "Generating and Exploiting Automated Reasoning Proof Certificates." In: *Commun. ACM* 66.10 (Sept. 2023), 86–95. ISSN: 0001-0782.

[15] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. "Satisfiability Modulo Theories." In: *Handbook of Satisfiability*. 2021.

[16] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. "CVC4." In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 171–177.

[17] Clark Barrett, Morgan Deters, Leonardo de Moura, Albert Oliveras, and Aaron Stump. "6 Years of SMT-COMP." In: *Journal of Automated Reasoning* 50.3 (2013), pp. 243–277. URL: http://dx.doi.org/10.1007/s10817-012-9246-5.

[18] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Available at www.SMT-LIB.org. Department of Computer Science, The University of Iowa, 2017–2024. URL: https://smt-lib.org/papers/smt-lib-reference-v2.6-r2024-09-20.pdf.

[19] Frédéric Besson. "Fast Reflexive Arithmetic Tactics the Linear Case and Beyond." In: *Intl. Workshop Types for Proofs and Programs (TYPES 2006)*. Ed. by Thorsten Altenkirch and Conor McBride. LNCS. Berlin, Heidelberg: Springer, 2006, pp. 48–62.

[20] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. "Extending Sledgehammer with SMT Solvers." In: *23rd Intl. Conf. Automated Deduction (CADE-23)*. Ed. by Nikolaj S. Bjørner and Viorica Sofronie-Stokkermans. Vol. 6803. LNCS. Wroclaw, Poland: Springer, 2011, pp. 116–130.

[21] Frédéric Blanqui. "Type Safety of Rewrite Rules in Dependent Types." In: *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*. Vol. 167. Leibniz International Proceedings in Informatics (LIPIcs). 2020, 13:1–13:14.

[22] Frédéric Blanqui. "Encoding Type Universes Without Using Matching Modulo Associativity and Commutativity." In: *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*. Ed. by Amy P. Felty. Vol. 228. Leibniz International Proceedings in Informatics (LIPIcs). 2022, 24:1–24:14.

[23] Frédéric Blanqui. "Translating HOL-Light proofs to Coq." In: *Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning*. Balaclava, Mauritius, May 2024, pp. 1–18.

[24] Frédéric Blanqui, Thérèse Hardin, and Pierre Weis. "On the Implementation of Construction Functions for Non-free Concrete Data Types." In: *Programming Languages and Systems*. Ed. by Rocco De Nicola. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 95–109.

[25] Martin Blicha, Konstantin Britikov, and Natasha Sharygina. "The Golem Horn Solver." In: *Computer Aided Verification: 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part II*. Berlin, Heidelberg: Springer-Verlag, 2023, 209–223. ISBN: 978-3-031-37702-0.

[26] Sascha Böhme and Tjark Weber. "Fast LCF-Style Proof Reconstruction for Z3." In: *First Intl. Conf. Interactive Theorem Proving (ITP 2010)*. Ed. by Matt Kaufmann and Lawrence C. Paulson. Vol. 6172. LNCS. Berlin, Heidelberg: Springer, 2010, pp. 179–194.

[27] Samuel Boutin. "Using reflection to build efficient and certified decision procedures." In: *Theoretical Aspects of Computer Software*. Ed. by Martín Abadi and Takayasu Ito. Berlin, Hei-

delberg: Springer Berlin Heidelberg, 1997, pp. 515–529. ISBN: 978-3-540-69530-1.

[28] Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. "veriT: An Open, Trustable and Efficient SMT-Solver." In: *22nd Intl. Conf. Automated Deduction (CADE-22)*. Ed. by Renate A. Schmidt. Vol. 5663. Lecture Notes in Computer Science. Montreal, Canada: Springer, 2009, pp. 151–156.

[29] Robert Brummayer and Armin Biere. "Fuzzing and delta-debugging SMT solvers." In: SMT '09 (2009).

[30] Clearsy. *Atelier B*. https://www.atelierb.eu. Version 24.04.2.

[31] Alessio Coltellacci and Stephan Merz. "Checking Linear Integer Arithmetic Proofs in Lambdapi." In: *Frontiers of Combining Systems*. Ed. by René Thiemann and Christoph Weidenbach. Cham: Springer Nature Switzerland, 2026, pp. 367–385.

[32] Alessio Coltellacci, Stephan Merz, and Gilles Dowek. "Reconstruction of SMT Proofs with Lambdapi." In: *Proc. 22nd Intl. Wsh. Satisfiability Modulo Theories co-located with the 36th Intl. Conf. Computer Aided Verification (CAV 2024)*. Vol. 3725. CEUR Workshop Proceedings. Montreal, Canada: CEUR-WS.org, 2024, pp. 13–23.

[33] The mathlib Community. "The lean mathematical library." In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. POPL '20. ACM, Jan. 2020, 367–381. DOI: 10.1145/3372885.3373824.

[34] Evelyne Contejean, Claude Marché, and Landy Rabehasaina. "Rewrite systems for natural, integral, and rational arithmetic." In: *Rewriting Techniques and Applications*. Ed. by Hubert Comon. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 98–112. ISBN: 978-3-540-69051-1.

[35] Th. Coquand and C. Paulin-Mohring. "Inductively defined types." In: *Proceedings of Colog'88*. Ed. by P. Martin-Löf and G. Mints. Vol. 417. Lecture Notes in Computer Science. Springer-Verlag, 1990.

[36] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. "TLA$^+$ Proofs." In: *18th Intl. Symp. Formal Methods (FM 2012)*. Ed. by Dimitra Giannakopoulou and Dominique Méry. Vol. 7436. LNCS. Paris, France: Springer, 2012, '47–154.

[37] Denis Cousineau and Gilles Dowek. "Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo." In: *Typed Lambda Calculi and Applications*. Berlin, Heidelberg: Springer, 2007, pp. 102–117.

[38] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. "Efficient Certified RAT Verification." In: *Automated Deduction – CADE 26*. Ed. by Leonardo de Moura. Cham: Springer International Publishing, 2017, pp. 220–236.

[39] Lukasz Czajka. "A Shallow Embedding of Pure Type Systems into First-Order Logic." In: *22nd International Conference on Types for Proofs and Programs (TYPES 2016)*. Vol. 97. Leibniz International Proceedings in Informatics (LIPIcs). 2018, 9:1–9:39. ISBN: 978-3-95977-065-1.

[40] Łukasz Czajka and Cezary Kaliszyk. "Hammer for Coq: Automation for Dependent Type Theory." In: *Journal of Automated Reasoning* 61.1 (June 2018), pp. 423–453.

[41] Rosalie Defourné. "Encoding TLA$^+$ Proof Obligations Safely for SMT." In: *Lecture Notes in Computer Science*. Vol. 14010. Lecture Notes in Computer Science. Nancy, France: Springer Nature Switzerland, May 2023, pp. 88–106.

[42] David Déharbe. "Integration of SMT-solvers in B and Event-B development environments." In: *Science of Computer Programming* 78 (Mar. 2013), pp. 310–326. DOI: 10.1016/j.scico.2011.03.007.

[43] David Delahaye. "A Tactic Language for the System Coq." In: *Logic for Programming and Automated Reasoning*. Ed. by Michel Parigot and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 85–95. ISBN: 978-3-540-44404-6.

[44] David Delahaye, Damien Doligez, Frédéric Gilbert, Pierre Halmagrand, and Olivier Hermant. "Zenon Modulo: When Achilles Outruns the Tortoise using Deduction Modulo." In: *LPAR - Logic for Programming Artificial Intelligence and Reasoning - 2013*. Vol. 8312. LNCS. Berlin, Heidelberg: Springer, 2013, pp. 274–290. DOI: 10.1007/978-3-642-45221-5_20.

[45] Niklas Eén and Niklas Sörensson. "An Extensible SAT-solver." In: *Theory and Applications of Satisfiability Testing*. Ed. by Enrico Giunchiglia and Armando Tacchella. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 502–518.

[46] M. Eilers, M. Schwerhoff, A. J. Summers, and P. Müller. "Fifteen Years of Viper." In: *Computer Aided Verification (CAV)*. Ed. by Ruzica Piskac and Zvonimir Rakamarić. Cham: Springer Nature Switzerland, 2025, pp. 107–123. DOI: 10.1007/978-3-031-98668-0_5.

[47] Mathias Fleury. "Formalization of Logical Calculi in Isabelle/HOL." Theses. Universität des Saarlandes Saarbrücken, Jan. 2020. URL: https://hal.univ-lorraine.fr/tel-02963301.

[48] Mathias Fleury. "IsaSAT and Kissat entering the EDA Challenge 2021." In: 2021. URL: https://api.semanticscholar.org/CorpusID:236204544.

[49] Mathias Fleury, Jasmin Christian Blanchette, and Peter Lammich. "A verified SAT solver with watched literals using imperative HOL." In: *CPP 2018 - The 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. Los Angeles, United States: ACM Press, Jan. 2018. DOI: 10.1145/3167080. URL: https://inria.hal.science/hal-01904647.

[50] Mathias Fleury and Hans-Jörg Schurr. "Reconstructing veriT Proofs in Isabelle/HOL." In: *Electronic Proceedings in Theoretical Computer Science* 301 (Aug. 2019), 36–50. ISSN: 2075-2180. DOI: 10.4204/eptcs.301.6. URL: http://dx.doi.org/10.4204/EPTCS.301.6.

[51] Jürgen Giesl et al. "Proving Termination of Programs Automatically with AProVE." In: *Automated Reasoning*. Cham: Springer International Publishing, 2014, pp. 184–191.

[52] Anne Grieu. "From Event-B to Lambdapi." In: *Rigorous State-Based Methods - 10th International Conference, ABZ 2024, Bergamo, Italy, June 25-28, 2024, Proceedings*. Ed. by Silvia Bonfanti, Angelo Gargantini, Michael Leuschel, Elvinia Riccobene, and Patrizia Scandurra. Vol. 14759. Lecture Notes in Computer Science. Springer, 2024, pp. 387–391.

[53] Robert Harper, Furio Honsell, and Gordon Plotkin. "A framework for defining logics." In: *J. ACM* 40.1 (Jan. 1993), 143–184. ISSN: 0004-5411. DOI: 10.1145/138027.138060.

[54] Marijn J. H. Heule. *The DRAT format and DRAT-trim checker*. 2016. arXiv: 1610.06229.

[55] Marijn Heule, Warren Hunt, Matt Kaufmann, and Nathan Wetzler. "Efficient, Verified Checking of Propositional Proofs." In: *Interactive Theorem Proving*. Ed. by Mauricio Ayala-Rincón and César A. Muñoz. Cham: Springer International Publishing, 2017, pp. 269–284.

[56] Guy Katz, Clark Barrett, Cesare Tinelli, Andrew Reynolds, and Liana Hadarean. "Lazy proofs for DPLL(T)-based SMT solvers." In: *2016 Formal Methods in Computer-Aided Design (FMCAD)*. 2016, pp. 93–100.

[57] Benjamin Kiesl, Adrián Rebola-Pardo, and Marijn J. H. Heule. "Extended Resolution Simulates DRAT." In: *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*. Ed. by Didier Galmiche, Stephan Schulz, and Roberto Sebastiani. Vol. 10900. Lecture Notes in Computer Science. Springer, 2018, pp. 516–531.

[58] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201896842.

[59] Laura Kovács and Andrei Voronkov. "First-Order Theorem Proving and Vampire." In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–35. ISBN: 978-3-642-39799-8.

[60] Hanna Lachnitt, Mathias Fleury, Leni Aniva, Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. "IsaRare: Automatic Verification of SMT Rewrites in Isabelle/HOL." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Bernd Finkbeiner and Laura Kovács. Cham: Springer Nature Switzerland, 2024, pp. 311–330. ISBN: 978-3-031-57246-3.

[61] Hanna Lachnitt, Mathias Fleury, Haniel Barbosa, Andrew Reynolds, Jibiana Jakpor, Bruno Andreotti, Clark Barrett, Hans-Joerg Schurr, and Cesare Tinelli. "Improving the SMT Proof Reconstruction Pipeline in Isabelle/HOL." In: *Interactive Theorem Proving (ITP)*. Vol. 1. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2025, pp. 100–100.

[62] Peter Lammich. "Efficient Verified (UN)SAT Certificate Checking." In: *Automated Deduction – CADE 26*. Ed. by Leonardo de Moura. Cham: Springer International Publishing, 2017, pp. 237–254.

[63] Leslie Lamport. "The temporal logic of actions." In: *ACM Trans. Program. Lang. Syst.* 16.3 (May 1994), 872–923. ISSN: 0164-0925.

[64] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 032114306X.

[65] K. Rustan M. Leino. "Dafny: an automatic program verifier for functional correctness." In: *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. LPAR'10. Berlin, Heidelberg: Springer-Verlag, 2010, 348–370.

[66] Yann Leray, Gaëtan Gilbert, Nicolas Tabareau, and Théo Winterhalter. "The Rewster: Type Preserving Rewrite Rules for the Coq Proof Assistant." working paper or preprint. Mar. 2024. URL: https://inria.hal.science/hal-04511667.

[67] Per Martin-Löf. "Intuitionistic Type Theory." 1980.

[68] Oliver Marx, Markus Wedler, Dominik Stoffel, Wolfgang Kunz, and Alexander Dreyer. "Proof logging for computer algebra based SMT solving." In: *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2013, pp. 677–684. DOI: 10.1109/ICCAD.2013.6691188.

[69] Stephan Merz. *EWD840: Termination Detection in a Ring*. URL: https://github.com/tlaplus/Examples/tree/master/specifications/ewd840 (visited on 04/25/2026).

[70] Stephan Merz. "The Specification Language TLA$^+$." In: *Logics of Specification Languages*. Ed. by Dines Bjørner and Martin C. Henson. Monographs in Theoretical Computer Science. Berlin-Heidelberg: Springer, 2008, pp. 401–451.

[71] Abdalrhman Mohamed, Tomaz Mascarenhas, Harun Khan, Haniel Barbosa, Andrew Reynolds, Yicheng Qian, Cesare Tinelli, and Clark Barrett. *Lean-SMT: An SMT tactic for discharging proof goals in Lean*. 2025. arXiv: 2505.15796 [cs.LO]. URL: https://arxiv.org/abs/2505.15796.

[72] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. "Chaff: engineering an efficient SAT solver." In: *Proceedings of the 38th Annual Design Automation Conference*. New York, NY, USA: Association for Computing Machinery, 2001, 530–535.

[73] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Springer Berlin Heidelberg, 2008, pp. 337–340.

[74] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 Theorem Prover and Programming Language." In: *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, 625–635.

[75] Rajdeep Mukherjee, Daniel Kroening, and Tom Melham. "Hardware Verification Using Software Analyzers." In: *2015 IEEE Computer Society Annual Symposium on VLSI*. 2015, pp. 7–12. DOI: 10.1109/ISVLSI.2015.107.

[76] 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), 937–977. ISSN: 0004-5411.

[77] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Berlin, Heidelberg: Springer-Verlag, 2002. ISBN: 3540433767.

[78] Andres Nötzli, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark Barrett, and Cesare Tinelli. "Reconstructing Fine-Grained Proofs of Rewrites Using a Domain-Specific Language." In: *FMCAD 2022*. 2022, pp. 65–74. DOI: 10.34727/2022/isbn.978-3-85448-053-2_12.

[79] Jonathan Protzenko et al. "Verified low-level programming embedded in F*." In: *Proceedings of the ACM on Programming Languages* 1.ICFP (Aug. 2017), 1–29. ISSN: 2475-1421.

[80] William Pugh. "The Omega test: A fast and practical integer programming algorithm for dependence analysis." In: *Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. 1991, pp. 4–13.

[81] Adrián Rebola-Pardo and Luís Cruz-Filipe. "Complete and Efficient DRAT Proof Checking." In: *2018 Formal Methods in Computer Aided Design (FMCAD)*. 2018, pp. 1–9.

[82] John C. Reynolds. "Types, Abstraction and Parametric Polymorphism." In: *IFIP Congress*. 1983.

[83] John Rushby. "Formal Methods and their Role in the Certification of Critical Systems." In: *Safety and Reliability of Software Based Systems*. Ed. by Roger Shaw. London: Springer London, 1997, pp. 1–42. ISBN: 978-1-4471-0921-1.

[84] Ronan Saillard. "Typechecking in the lambda-Pi-Calculus Modulo : Theory and Practice." Theses. Ecole Nationale Supérieure des Mines de Paris, Sept. 2015.

[85] Alexander Schrijver. *Theory of linear and integer programming*. USA: John Wiley & Sons, Inc., 1986. ISBN: 0471908541.

[86] Stephan Schulz. "System Description: E 0.81." In: *Automated Reasoning*. Ed. by David Basin and Michaël Rusinowitch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 223–228. ISBN: 978-3-540-25984-8.

[87] Hans-Jörg Schurr, Mathias Fleury, Haniel Barbosa, and Pascal Fontaine. "Alethe: Towards a Generic SMT Proof Format (extended abstract)." In: *PxTP 2021*, pp. 49–54.

[88] Hans-Jörg Schurr, Mathias Fleury, and Martin Desharnais. "Reliable Reconstruction of Fine-grained Proofs in a Proof Assistant." In: *Automated Deduction – CADE 28*. Cham: Springer International Publishing, 2021, pp. 450–467.

[89] Hans-Jörg Schurr, Mathias Fleury, and Martin Desharnais. "Reliable Reconstruction of Fine-grained Proofs in a Proof Assistant." In: *Automated Deduction – CADE 28*. Ed. by André Platzer and Geoff Sutcliffe. Cham: Springer International Publishing, 2021, pp. 450–467.

[90] Hans-Jörg Schurr, Mathias Fleury, Haniel Barbosa, and Pascal Fontaine. "Alethe: Towards a Generic SMT Proof Format (extended abstract)." In: *Electronic Proceedings in Theoretical Computer Science* 336 (July 2021), pp. 49–54.

[91] Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. "SMT proof checking using a logical framework." In: *Form. Methods Syst. Des.* 42.1 (Feb. 2013), 91–118. ISSN: 0925-9856.

[92] "Chapter 4 - The Curry-Howard isomorphism." In: *Lectures on the Curry-Howard Isomorphism*. Ed. by Morten Heine Sørensen and Paweł Urzyczyin. Vol. 149. Studies in Logic and the Foundations of Mathematics. Elsevier, 2006, pp. 77–101.

[93] The Rocq Development Team. *The Rocq Reference Manual – Release 9.0.0*. https://rocq-prover.org/doc/V9.0.0/refman/index.html. 2025.

[94] The Rust Team. *The Rust Programming Language*. https://rust-lang.org. 2025.

[95] François Thiré. "Interoperability between proof systems using the logical framework Dedukti." Theses. Université Paris-Saclay, Dec. 2020.

[96] Thomas Traversié. "Proofs for Free in the λΠ-Calculus Modulo Theory." In: *Electronic Proceedings in Theoretical Computer Science*. Vol. 404. Tallinn, Estonia, July 2024, pp. 49 –63.

[97] Vincent Trélat. "Safely Encoding B Proof Obligations in SMT-LIB." In: *Rigorous State-Based Methods*. Ed. by Michael Leuschel and Fuyuki Ishikawa. Cham: Springer Nature Switzerland, 2026, pp. 52–69. ISBN: 978-3-031-94533-5.

[98] Paweł Urzyczyn. "Intuitionistic Games: Determinacy, Completeness, and Normalization." In: *Studia Logica* 104.5 (Oct. 2016), pp. 957–1001. ISSN: 1572-8730. DOI: 10.1007/s11225-016-9661-4. URL: https://doi.org/10.1007/s11225-016-9661-4.

[99] Philip Wadler. "Theorems for free!" In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. New York, NY, USA: Association for Computing Machinery, 1989, 347–359. ISBN: 0897913280.

[100] Lucas Wagner, Alain Mebsout, Cesare Tinelli, Darren Cofer, and Konrad Slind. "Qualification of a Model Checker for Avionics Software Verification." In: Apr. 2017, pp. 404–419.

[101] Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz, and Giles Reger. "The SMT Competition 2015-2018." In: *J. Satisf. Boolean Model. Comput.* 11.1 (2019), pp. 221–259. URL: https://doi.org/10.3233/SAT190123.

[102] Harald Zankl, Bertram Felgenhauer, and Aart Middeldorp. "CSI - A Confluence Tool." In: *23rd Intl. Conf. Automated Deduction (CADE-23)*. Ed. by Nikolaj S. Bjørner and Viorica Sofronie-Stokkermans. Vol. 6803. LNCS. Wroclaw, Poland: Springer, 2011, pp. 499–505.

[103] Mathematical Components team. *Mathematical Components Library*. https://github.com/math-comp/mathcomp. Available at: https://math-comp.github.io/. 2025.

DECLARATION

This dissertation is my own work and includes nothing which is the outcome of work done in collaboration with others except where specifically indicated in the text.

I further declare that no AI tools have been used in the development of the code for the reconstruction pipeline presented in this work.

France, 2026

Coltellacci Alessio