Theory of sequences tailored for program verification

Hichem Rami Ait El Hara

université PARIS-SACLAY

Theory of Sequences Tailored for Program Verification

Théorie des séquences adaptée à la vérification des programmes

Thèse de doctorat de l'université Paris-Saclay

École doctorale n°580 : Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat: Informatique
Graduate School : Informatique et sciences du numérique.
Référent : Faculté des Sciences d'Orsay

Thèse préparée dans l'unité de recherche **Institut LIST (Université Paris-Saclay, CEA)**, sous la direction de **François BOBOT**, Ingénieur-Chercheur, et le co-encadrement de **Guillaume BURY**, Ingénieur R&D à OCamlPro

Thèse soutenue à Paris-Saclay, le 15 octobre 2025, par

Hichem Rami AIT EL HARA

Composition du jury

Membres du jury avec voix délibérative

Sylvain CONCHON Professeur, Université Paris-Saclay	Président
Pascal FONTAINE Professeur, Université de Liège	Rapporteur & Examinateur
Maria Paola BONACINA Professeure, Università degli Studi di Verona	Rapporteuse & Examinatrice
Arnaud GOTLIEB Professeur, Simula Research Laboratory	Examinateur
Claire DROSS Docteure, Adacore	Examinatrice

THESE DE DOCTORAT

NNT : 2025UPASG067

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Théorie des séquences adaptée à la vérification des programmes

Mots clés: Vérification de programmes, Théorie des séquences, Solveur SMT, Satisfibilité Modulo Théories, Solveur de contraintes, Programmation par contraintes.

Résumé: Les choix de modèles sémantiques d'un langage de programmation ont un effet important sur l'efficacité de la vérification des programmes dans ce langage. En effet, de nombreuses techniques de vérification génèrent des formules mathématiques en utilisant ces modèles. Les théories mathématiques utilisées dans ces formules et leur forme ont un impact direct sur leur solvabilité par le solveur utilisé.

La modélisation de la mémoire et des structures de données utilise souvent la théorie SMT (Satisfiabilité Modulo Théories) des tableaux, qui est bien établie et utilisée dans le domaine des solveurs SMT. Dans cette théorie, les tableaux permettent d'associer des valeurs à des indices, quel que soit le type des indices ou des valeurs. La théorie permet également des opérations permettant d'écrire et de lire le contenu des tableaux. Cependant, dans les programmes concrets d'où proviennent les formules à prouver, la mémoire et les structures de données sont généralement limitées. Par exemple, les tableaux dans les langages de programmation sont généralement indexés de 0 à une constante n. Bien qu'il soit envisageable d'encoder les tableaux finis dans la théorie SMT des tableaux, cela n'est pas toujours une solution satisfaisante, une raison étant que l'égalité extensionnelle sur un tableau fini de 0 à n ne peut pas être directement modélisée en utilisant l'égalité extensionnelle sur des tableaux infinis, qui considère tous les entiers. Une théorie SMT des séquences finies, dans laquelle les séquences sont des collections de valeurs indexées sur un ensemble contigu d'entiers, simplifierait la résolution des formules qui modélisent de telles structures de données. De plus, les séquences finies avec des opérateurs de concaténation et d'extraction peuvent également être utilisées pour exprimer des langages de spécification particuliers tels que la logique de séparation. Une difficulté est de choisir l'ensemble des opérations sur les séquences à supporter, puisque la décidabilité de la théorie en dépend. D'un autre côté, la décidabilité complète n'est pas toujours requise, car les formules obtenues par la vérification de programmes peuvent avoir une forme ou une utilisation spécifique des opérations.

L'objectif de la thèse est d'étudier quelle théorie des séquences est appropriée pour la vérification de programmes. La thèse s'intéresse notamment au cas des séquences n-indexées, qui sont des séquences qui peuvent commencer à n'importe quel indice n et se terminer à n'importe quel indice m. Ces séquences sont notamment présentes dans le langage de programmation Ada, mais étant une généralisation des séquences 0-indexées, elles devraient permettre de représenter et de raisonner sur ces dernières également. Dans cette thèse, une théorie des séquences n-indexées est proposée, et différentes façons de raisonner sur cette théorie sont explorées, que ce soit à travers des axiomatisations, en utilisant des théories existantes ou en développant différents calculi dédiés à cette théorie. L'évaluation de ces approches de raisonnement est effectuée à travers des implémentations en OCaml dans le solveur de contraintes Colibri2. Divers détails de ces implémentations sont présentés, ainsi que des contributions supplémentaires apportées au raisonnement arithmétique.

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Title: Theory of Sequences Tailored for Program Verification

Keywords: Program verification, Theory of sequences, SMT solver, Satisfiability Modulo Theories, Constraint solver, Constraint programming.

Abstract: The choices of semantic models for a programming language have a significant effect on the efficiency of the verification of programs in that language. Indeed, many verification techniques generate mathematical formulas using those models. The mathematical theories used in these formulas and their shape have a direct impact on their solvability by the used solver.

The modelization of memory and data structures often uses the SMT (Satisfiability Modulo Theories) theory of arrays, which is well established and used in SMT solvers. In this theory, arrays associate values with indices, both of which can be of any type. The theory also allows for operations that enable the writing and reading of the stored data. However, in the concrete programs from which the formulas that need to be solved are produced, memory and data structures are usually limited. For example, arrays in programming languages are usually indexed from 0 to a constant n. Although it is possible to encode finite arrays in the SMT theory of arrays, that is not always a satisfying solution, one reason being that extensional equality on a finite array from 0 to n cannot be directly modeled using the extensional equality on infinite arrays, which considers all integers. An SMT theory of finite sequences, in which the sequences are collections of values indexed over a set of contiguous integers, would simplify the solving of formulas that model such data structures. Moreover, finite sequences with concatenation and extraction operators can also be used to express particular specification languages such as separation logic. One difficulty is choosing the set of operations on the sequences to support, since the decidability of the theory depends on them. On the other hand, complete decidability is not always required, since the formulas obtained from program verification can have specific shapes or uses of the operations.

The goal of the thesis is to study which theory of sequences is suitable for program verification. The thesis focuses particularly on the case of n-indexed sequences, which are sequences that can start at any index n and end at any index m. These sequence appear in the Ada programming language, but since they are a generalization of 0-indexed sequences, they should also make it possible to represent and reason about the latter. In this thesis, a theory of n-indexed sequences is proposed, and different ways of reasoning about this theory are explored, whether through axiomatizations, by using existing theories, or by developing various calculi dedicated to this theory. The evaluation of these reasoning approaches is carried out through OCaml implementations in the Colibri2 CP (Constraint Programming) solver. Various details of these implementations are presented, as well as additional contributions to arithmetic reasoning.

ᕮ ᕽᕮᑕₒᒪᐃₒlᐨᕮᒪ, ᕮ ᕂᒪₒᑫᕾᐃᕂᐨᕮᒪ, ᕮ ᒪᕾᕉᐱᕾᐱᐨᕮᒪ.

Acknowledgements

This PhD involved a significant investment of time and effort on my part. A lot of which I spent working alone from home, reading papers, writing drafts, preparing this dissertation, running experiments, and writing far too many lines of code, a good chunk of which were eventually abandoned when they did not produce the hoped-for results. Yet, like any research project, this thesis is not the result of my work alone. For it to be possible, and for me to be able to successfully carry it out and complete it, depended on the direct and indirect contributions of many people. I would like here to express my sincere gratitude to all of them for making this journey possible.

First, I want to thank my PhD supervisors, François at the CEA and Guillaume at OCamlPro, for accepting me as a PhD student, for everything I learned from them, and for all the scientific and technical discussions we shared during these last three years.

I also want to thank the members of my jury. Thank you to Pascal and Maria Paola for agreeing to be rapporteurs on my manuscript and for their helpful and insightful remarks. Thank you to Sylvain for accepting to preside over the jury. And thank you to Arnaud and Claire for accepting to be my examiners.

I am grateful to Fabrice and Muriel at OCamlPro for creating a place where scientific rigor meets real-world pragmatism and for funding this PhD. OCamlPro was also where I had my first professional experience after my master's degree. I first joined as an intern for six months, then as a Research and Development Engineer for one year before starting my PhD. This allowed me to deepen and expand my understanding of both software engineering and formal methods.

I also want to thank all my colleagues at OCamlPro, both those who are still there and those who have since moved on to "fly under other skies". I am grateful to have met so many talented people and to have taken part in many technical and non-technical discussions, whether at the "coin café" in the Alesia office or during seminars and conferences. I would also like to thank the many interns who passed through over the years, whether they went on to work in the industry or pursued a PhD, and whether we still cross paths or not. I especially want to thank Félix, whom I had the pleasure of supervising with my colleague Leo during my final year as a PhD student.

I am also thankful to my colleagues at the LSL lab at the CEA. Although I spent less time there, I still had the chance to meet many talented people and to attend interesting talks and discussions. I want to thank in particular the Colibri and Frama-C/WP teams, with whom I interacted the most, as well as the many PhD students I met over the years.

My thanks also go to everyone I met at conferences, workshops, and summer schools. Thank you for the many discussions and enriching exchanges. You were all part of this PhD experience for me, and I am glad that our paths crossed. I wish the best of luck to you all, especially those of you who are still pursuing your doctorate.

Finally, I want to thank my parents, who encouraged my curiosity and my love of learning and seeking knowledge, and who instilled in me the discipline that allowed me to reach this point. I also want to thank my brother, who paved the way for me to move abroad and to pursue a Bachelor's, a Master's then a PhD, as well as my other siblings and my whole family for their support and encouragement, even though most of them have no idea what my work is about.

Résumé en Français

Les méthodes formelles sont un domaine d'études à l'intersection de la logique, des mathématiques et de l'informatique. Elles consistent à utiliser des techniques rigoureuses suivant des règles mathématiques et logiques strictes et non ambiguës afin de vérifier les programmes et de garantir la sûreté des logiciels.

Aujourd'hui, les logiciels sont présents dans presque tous les aspects de la vie humaine moderne, y compris dans des domaines hautement critiques tels que les transports, l'industrie, le secteur énergétique, les institutions financières, et bien d'autres encore. Assurer la sûreté des logiciels est donc plus important que jamais, et son importance continue de croître à mesure que les logiciels occupent des rôles de plus en plus significatifs.

Derrière ces logiciels se trouvent des programmes. Un programme correspond concrètement à une suite d'instructions fournies aux ordinateurs afin d'être exécutées et de réaliser certaines tâches. Parfois, l'exécution d'un programme peut différer de celle décrite dans sa spécification, la spécification étant une description du comportement prévu du programme. Lorsque le programme se comporte différemment de ce comportement prévu, on dit qu'il est défectueux ou buggé.

Une méthode formelle spécifique, appelée la vérification déductive, cible précisément ce problème. Son rôle est de prendre un programme et sa spécification et de vérifier formellement que le programme respecte cette spécification. En d'autres mots, le but est de prouver qu'un programme se comportera toujours comme il est prévu de se comporter. Pour ce faire, la spécification doit être formelle, c'est-à-dire précise et exprimée en termes mathématiques et logiques. Une fois qu'une spécification formelle et un programme correspondant existent, les outils de vérification déductive traduisent ces programmes et ces spécifications en formules mathématiques et logiques, de sorte que prouver la validité de ces formules garantit que le programme respecte sa spécification.

L'une des façons les plus utilisées pour prouver automatiquement ces formules, notamment dans l'industrie, consiste à utiliser des solveurs SMT (Satisfiabilité Modulo Theories). En effet, les solveurs prennent en entrée des formules dans lesquelles sont combinées des expressions logiques avec des théories en logique du premier ordre, et permettent soit de prouver leur validité, soit de générer un contre-exemple lorsque ces formules ne sont pas prouvables. Les théories en logique du premier ordre supportées par les solveurs SMT peuvent être classées en deux grandes catégories : celles représentant des valeurs mathématiques, telles que les entiers, les réels et les tableaux fonctionnels (qui sont en fait des mappings), et celles représentant les types et structures de données présentes dans des langages de programmation, comme les théories des vecteurs de bits, de l'arithmétique flottante, des types de données algébriques, des chaînes de caractères et des séquences. Ces dernières sont particulièrement utiles pour vérifier des programmes manipulant de tels types de données, car elles simplifient considérablement la traduction des programmes et des spécifications en formules logiques.

Cette thèse s'inscrit dans ce contexte. L'objectif est de contribuer à réduire l'écart entre ce qui peut être exprimé et prouvé en SMT et les structures de données et opérations réellement utilisées dans les langages de programmation. Elle le fait à travers une étude de cas spécifique : les séquences n-indexées, qui sont des structures de données représentant des collections de valeurs d'un même type, indexées à partir d'un entier n jusqu'à un autre entier m. Comme aucune théorie SMT n'existe

actuellement pour représenter et raisonner sur de telles structures de données, cette thèse propose une nouvelle théorie des séquences n-indexées et explore différentes manières de raisonner sur cette théorie. Elle s'intéresse également à des sujets connexes, notamment d'autres théories dont ce travail dépend, comme l'arithmétique linéaire.

Le reste de ce manuscrit est réparti en 7 chapitres. Le Chapter 1 sert d'introduction et présente les contributions de la thèse. Le Chapter 2 donne une vue d'ensemble du contexte scientifique de ce travail, ainsi que l'état de l'art des travaux connexes au sujet de la thèse. Le Chapter 3 présente le fonctionnement du solveur de contraintes Colibri2, qui est l'outil dans lequel les implémentations des contributions de cette thèse ont été réalisées. Le chapitre se concentre notamment sur le raisonnement arithmétique dans Colibri2, qui sera très utilisé pour raisonner sur les séquences n-indexées. Il présente également la structure de données "labeled union-find", utilisée aussi bien dans l'implémentation du raisonnement arithmétique que pour le raisonnement sur les séquences n-indexées.

Le Chapter 4 présente la théorie des séquences n-indexées ainsi que les différentes façons développées pour raisonner dessus, que ce soit en utilisant des théories déjà existantes, en réadaptant des calculs développés pour la théorie des séquences, ou à travers un nouveau calcul original développé dans le cadre de cette thèse. Ensuite, le Chapter 5 traite de l'implémentation des calculs décrits dans Chapter 4, en donnant certains détails et en expliquant plusieurs choix d'implémentation. Le chapitre présente également les résultats expérimentaux sur les performances de ces implémentations et les compare aux résultats obtenus par d'autres outils.

Le Chapter 6 présente différentes contributions faites au raisonnement arithmétique dans Colibri2 afin d'en améliorer les performances, notamment pour le raisonnement sur les séquences. Enfin, le manuscrit se conclut avec le Chapter 7, qui présente notamment des perspectives et des applications potentielles supplémentaires de ce travail.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

At the intersection of logic, mathematics, and computer science lies the field of formal methods, dedicated to employing rigorous techniques following strict and unambiguous mathematical and logical rules to ensure that software is safe.

Today, software is involved in nearly every aspect of modern human life, including highly critical domains such as transportation, healthcare systems, financial institutions, industrial production, energy facilities, and more. Ensuring that software is safe is therefore more important than ever, and its importance continues to grow as software takes on increasingly significant roles.

The IEEE standard glossary of software engineering terminology [68] defines software as "computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system", and a program as "a combination of computer instructions and data definitions that enable computer hardware to perform computational or control functions".

In other words, a program is what software developers tell the computer to do concretely. This may differ from the specification of a program, which describes the intended behavior of the program. Differences between a program's specification and its actual behavior are a significant source of bugs in software engineering.

A specific formal method called deductive verification targets this problem. Its role is to take a program and its specification and formally verify that the program complies with what the specification states. The specification must be a formal specification, i.e. it must be precise and expressed in mathematical and logical terms. However, in practice, most specifications exist only in natural language (if they exist at all), making this task even harder.

Once a formal specification and a corresponding program exist, there are tools that can transform the program and its specification into mathematical and logical formulas. Proving the validity of these formulas ensures that the program respects its specification.

One such tool is the Why3 deductive verification framework, which offers its own programming and formal specification language. Why3 relies on Hoare logic, a formal system that verifies that a program satisfies its specification. In Hoare logic, each (part of a) program s is associated with a precondition p and a postcondition q, together forming

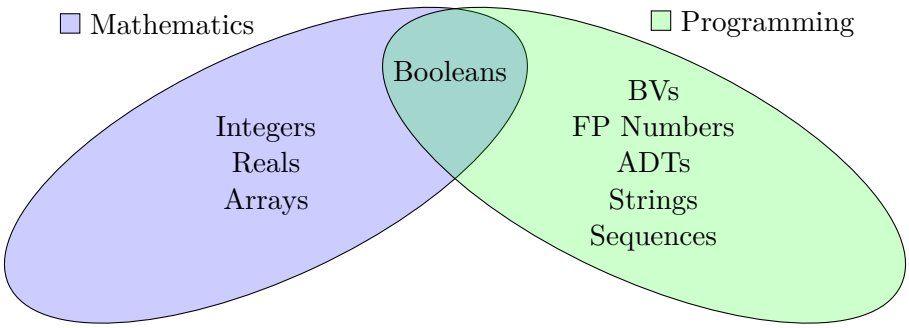

Figure 1.1: An illustration of some popular SMT theories classified into two categories: Mathematics and Programming

a Hoare triple, denoted $\{p\}s\{q\}$. The triplet $\{p\}s\{q\}$ means that if the precondition p holds, then after executing s, the postcondition q should also hold.

Why3 translates these Hoare triples into Satisfiability Modulo Theories (SMT) formulas, which are then checked by SMT solvers. SMT formulas combine logical expressions with first-order logic theories. As illustrated in Figure 1.1, these theories can be classified into two broad categories: those representing mathematical values, such as integers, reals and arrays, and those representing types and data structures from programming languages, such as the theories of bit-vectors (BV), floating-point (FP) arithmetic, algebraic data types (ADTs), strings and sequences. The latter are particularly helpful for verifying programs that manipulate such data types because they significantly simplify the translation of programs and specifications into logical formulas.

This thesis is situated within this context. Its goal is to help bridge the gap between what can be expressed and proven in SMT and the data structures and operations actually used in programming languages. It does so through a specific case study: n-indexed sequences, which are data structures representing collections of values of the same type, indexed from any given integer n to another integer m. Since no SMT theory currently exists for reasoning about such data structures, this thesis explores different ways to achieve this and develops new methods for doing so. It also investigates related topics, notably other theories on which this work depends, such as linear arithmetic.

1.1 Overview of the Thesis and Contributions

This manuscript is divided into seven chapters:

Chapter 1 is the current one, which serves as the introduction.

Chapter 2 provides background and technical preliminaries. It presents, in seven sections, all the necessary technical background to understand the remainder of the manuscript. It starts with Boolean satisfiability in Section 2.1, then moves on to many-sorted first-order logic in Section 2.2. Based on these two, it introduces Satisfiability Modulo Theories (SMT) and Constraint Programming (CP) in Sections 2.3 and 2.4, respectively. It then discusses the theories of arrays and sequences, along with common reasoning techniques over them, in Sections 2.5 and 2.6. Finally, Section 2.7 presents Colibri2, a CP solver used for reasoning over SMT formulas, in which the implementations

described in the manuscript were made, and which was used for experimental evaluation.

Chapter 3 is the first chapter on arithmetic reasoning. It starts in Section 3.1 with a formalization of the arithmetic reasoning in Colibri2. In particular, Section 3.1.1 describes four important domains used in Colibri2 for arithmetic reasoning: the domains of interval unions, polynomials, products, and modular arithmetic. The first three domains were developed independently of this thesis, but their formalization is one of this thesis's contributions. The fourth domain, modular arithmetic, was developed during the work on [80] as part of this thesis. Section 3.1.2 describes the propagation engine used for arithmetic reasoning in Colibri2, which was also developed independently of this thesis.

Section 3.2 introduces the labeled union-find data structure and the constant difference relation. It begins with a reminder on the union-find data structure in Section 3.2.1, then presents the labeled union-find data structure in Section 3.2.2 which was partially developed as part of the work carried out during this thesis. Sections 3.2.3 and 3.2.4 formalize the constant difference relation in Colibri2 and describe its interaction with the Shostak theory of arithmetic. Although these components were implemented independently of this thesis, their formalization had not previously been done and is part of the contributions of this thesis.

Chapter 4 describes one of the main contributions of this thesis: the theory of n-indexed sequences. Section 4.1 introduces the developed theory. Section 4.2 discusses reasoning approaches that use existing theories, while Section 4.3 presents a native calculus developed by adapting calculi from the theory of sequences. Section 4.4 details the original calculus developed specifically for n-indexed sequences.

Chapter 5 builds upon Chapter 4 by presenting the implementations of the calculi described there in Section 5.1, which were done as part of this thesis. Section 5.2 presents the experimental evaluations of these implementations.

Chapter 6 details contributions made during this thesis to arithmetic reasoning in Colibri2, aimed at improving its efficiency for reasoning over n-indexed sequences. In particular, Section 6.1 presents the implementation of difference logic reasoning in Colibri2. Section 6.2 describes how the labeled union-find was employed to enhance the propagation of arithmetic interval constraints.

Chapter 7 concludes the thesis and discusses possible future work and perspectives that can be envisaged based on the results of this research.

1.2 Publications

Part of the contributions described in this manuscript were published in the following papers:

- Hichem Rami Ait-El-Hara, François Bobot, and Guillaume Bury. "On SMT Theory Design: The Case of Sequences". In: *LPAR 2024 Complementary Volume*. Ed. by Nikolaj Bjørner, Marijn Heule, and Andrei Voronkov. Vol. 18. Kalpa Publications in Computing. EasyChair, May 2024, pp. 14–29. DOI: 10.29007/75tl

- Hichem Rami Ait-El-Hara, François Bobot, and Guillaume Bury. "An SMT Theory for n-Indexed Sequences". In: *Proceedings of the 22nd International Workshop on Satisfiability Modulo Theories*. Ed. by Giles Reger and Yoni Zohar. Vol. 3725.

CEUR Workshop Proceedings. Montreal, Canada: CEUR, July 2024, pp. 64–74. URL: https://ceur-ws.org/Vol-3725/#short13

- Dorian Lesbre, Matthieu Lemerre, Hichem Rami Ait-El-Hara, and François Bobot. "Relational Abstractions Based on Labeled Union-Find". In: *Proc. ACM Program. Lang.* 9.PLDI (June 2025). DOI: 10.1145/3729298
- Hichem Rami Ait-El-Hara, François Bobot, and Guillaume Bury. "Reasoning over n-indexed sequences in SMT". in: *Acta Informatica* 62.3 (Aug. 2025), p. 33. ISSN: 1432-0525. DOI: 10.1007/s00236-025-00496-w

Chapter 2

Background

The purpose of this chapter is to introduce the reader to the general topic of automated deduction and how it is used for program verification. Section 2.1 offers a quick presentation of the main ideas behind propositional logic and defines basic concepts such as satisfiability, unsatisfiability, validity, decidability, and completeness. These concepts will be used in the rest of this manuscript. Section 2.2 discusses many-sorted first-order logic, a more expressive logical framework than propositional logic. This section also introduces first-order theories. Section 2.3 presents satisfiability modulo theories and some first-order logic theories that are used in it, and Section 2.4 introduces constraint programming and how it relates to sections 2.1 and 2.3.

2.1 Boolean Satisfiability

Propositional logic, or Boolean logic, is a branch of logic in which formulas are expressed through the boolean constants true and false, boolean variables, and logic connectors. It allows expressing boolean satisfiability problems, or SAT problems [23], in the form of propositional logic formulas.

The syntax of propositional logic formulas is defined by the grammar in Figure 2.1, where V_b denotes the set of propositional variables. Figure 2.2 describes the semantics of logical connectives in propositional logic. From these semantics, additional important logical equivalences can be derived. For example the double negation law, which states that for any formula p:

- $\neg\neg p \equiv p$

As well as De Morgan's laws, which state that for any formulas p and q:

- $\neg(p \vee q) \equiv \neg p \wedge \neg q$
- $\neg(p \wedge q) \equiv \neg p \vee \neg q$

These equivalences are often used to simplify SAT problems and help with solving them. Solving a SAT problem consists in determining whether the formula that expresses it is satisfiable or unsatisfiable.

Definition 2.1 (Interpretation)**.** *An interpretation of a logic formula is a mapping of every variable in the formula to a boolean value.*

$$
\begin{aligned}
\langle \text{formula} \rangle &::= \langle \text{atom} \rangle \\
&\mid \neg \langle \text{formula} \rangle \\
&\mid (\langle \text{formula} \rangle \langle \text{binop} \rangle \langle \text{formula} \rangle) \\
\langle \text{binop} \rangle &::= \wedge \mid \vee \mid \implies \mid \iff \\
\langle \text{atom} \rangle &::= \mathsf{true} \mid \mathsf{false} \mid v \in V_b
\end{aligned}
$$

Figure 2.1: The grammar of propositional logic.

Operation	Symbol	Semantics
Negation	$\neg p$	true if p is false, false if p is true.
Conjunction	$p \wedge q$	true if both p and q are true, false otherwise.
Disjunction	$p \vee q$	true if p or q or both are true, false otherwise.
Implication	$p \implies q$	same as $q \vee \neg p$.
Equivalence	$p \iff q$	same as $(q \implies p) \wedge (p \implies q)$.

Figure 2.2: Semantics of logic operators.

Definition 2.2 (Satisfiability, unsatisfiability and validity)**.** *A logic formula is said to be satisfiable if there exists an interpretation such that the formula evaluates to* true*, unsatisfiable if no such interpretation exists, and valid if it evaluates to* true *for all existing interpretations.*

From Definition 2.2, since an unsatisfiable formula evaluates to false under every interpretation, its negation evaluates to true under every interpretation, i.e. its negation is valid. Conversely, a formula is valid if and only if its negation is unsatisfiable. For this reason, a common way to establish the validity of a formula is to prove that its negation is unsatisfiable.

Figure 2.3 illustrates an example of a propositional logic formula, using the operators described in Figure 2.2 and where p, q, and r are boolean variables. Figure 2.4 shows the satisfiability of the formula for each combination of the truth values of its variables in the form of a truth table.

Example 2.1. *From the truth table in Figure 2.4 of the formula in Figure 2.3, the formula is satisfiable with the interpretation:*

$$I : \{p \mapsto \mathit{true}, q \mapsto \mathit{false}, r \mapsto \mathit{true}\}$$

which corresponds to the third row in the truth table.

A trivial way to solve SAT problems is by enumeration, which consists of assigning truth values to all its boolean variables and checking the satisfiability of the original formula for each possible combination of the truth values of the variables. Truth tables, such as the one illustrated in Figure 2.4, are an example of this approach.

Another popular way to solve SAT problems is by using BDDs (Binary Decision Diagrams) [6]. BDDs encode propositional logic formulas as binary trees. The internal

$$\underbrace{\underbrace{((p \implies q) \wedge (q \implies r))}_{a} \vee (p \implies r)}_{b}$$

Figure 2.3: Example of a propositional logic formula.

p	q	r	$p \implies q$	$q \implies r$	a	$p \implies r$	b
true	true	true	true	true	true	true	true
true	true	false	true	false	false	false	false
true	false	true	false	true	false	true	true
true	false	false	false	true	false	false	false
false	true	true	true	true	true	true	true
false	true	false	true	false	false	true	true
false	false	true	true	true	true	true	true
false	false	false	true	true	true	true	true

Figure 2.4: Truth table of the propositional logic formula in Figure 2.3.

nodes represent boolean variables, and the outgoing edges from a node correspond to the two possible truth values of the variable represented by the node. Each path from the root to a leaf represents an interpretation of the variables, and the leaf stores the resulting truth value of the formula under that interpretation.

Although they were first introduced by Lee [77] in 1959. BDDs became most useful when Brace, Rudell, and Bryant introduced an efficient implementation of BDDs [33] that notably featured subproblem sharing.

In practice BDDs tend to be better than truth tables, as they allow for better solving complexity since they make it possible to avoid redundancy by sharing sub-problems, and avoiding assigning values to variables when doing so is not necessary to determine the satisfiability of the problem. The order in which variables are considered has a significant impact on the efficiency of BDDs. If the satisfiability of a formula does not depend on certain variables, placing them later in the ordering allows the BDD to skip them entirely. In contrast, truth tables always require assigning values to all variables, regardless of their relevance.

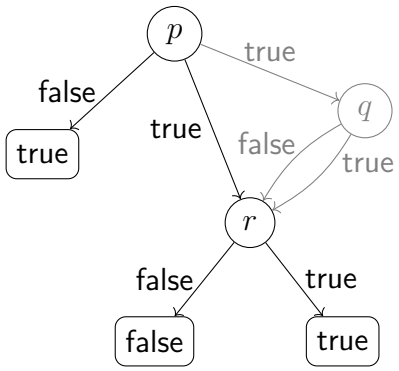

Figure 2.5: Binary decision diagram of the propositional logic formula in Figure 2.3.

Figure 2.5 illustrates a BDD that represents the truth table in Figure 2.4 of the problem in Figure 2.3. The node representing the variable p is at the root of the tree, its left edge, representing the case where p is false, points to the leaf true, which means that if p is false then the formula evaluates to true regardless of the truth values of r and q. The outgoing edge from p pointing to r and the outgoing edges from r show that when p and r are true, the formula is true. When p is true and r is false, then the formula is false. The node q and its outgoing and incoming edges are greyed out because the satisfiability of the formula can be determined from the values of p and r alone as shown in the BDD.

With the simplification illustrated in Figure 2.5, checking the satisfiability of a propositional logic formula expressed as a BDD reduces to checking whether there exists at least one path from the root to a leaf labeled true. If such a path exists, the formula is satisfiable, and the assignments to the variables along that path form a satisfying interpretation. Variables that do not appear on the path can take arbitrary truth values, since they do not affect the evaluation of the formula. Conversely, if the BDD reduces to a single leaf labeled false, then the formula is unsatisfiable.

Definition 2.3 (Soundness)**.** *A satisfiability checking procedure is sound if, whenever it answers that a formula is unsatisfiable, the formula is indeed unsatisfiable*

Definition 2.4 (Completeness)**.** *A satisfiability checking procedure is complete if, whenever a formula is unsatisfiable, the procedure will eventually answer that the formula is unsatisfiable.*

Definition 2.5 (Decision procedure)**.** *A decision procedure is a sound and complete satisfiability checking procedure.*

Definition 2.6 (Decidability)**.** *A logical theory is decidable if there exists a decision procedure for it.*

Propositional logic being a decidable logical theory, decision procedures to automatically solve SAT problems were developed. The first SAT solving decision procedure developed, better than enumeration, was the Davis-Putnam decision algorithm [46], introduced in 1960. The algorithm was soon after, in 1962, improved into the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [45]. Later on, the Conflict Driven Clause Learning (CDCL) algorithm was introduced [83].

Definition 2.7 (Literal)**.** *A literal is either a boolean variable p or its negation $\neg p$.*

Definition 2.8 (Conjunctive Normal Form)**.** *The conjunctive normal form (CNF) is a form in which problems are written as a conjunction of clauses $\bigwedge_i c_i$, where each clause c_i is a disjunction of literals $\bigvee_j l_j$.*

The DPLL and CDCL algorithms work on SAT problems encoded in conjunctive normal form (CNF) [104]. This representation is both simplified and convenient for the algorithms, as it allows them to quickly detect when a formula becomes false after deciding the value of variable, since it takes for one clause to be false for the whole formula to be false. Both algorithms are based on making decisions and backtracking. Making decisions, or case splitting, is the process through which, when the decision procedure does not manage to determine the truth value of a SAT problem by deduction alone, it assigns

values to variables whose values cannot be deduced and tries to solve the problem in each case. Backtracking works by setting a backtracking point before deciding which value to assign to a given variable. Setting a backtracking point consists in saving a snapshot of the current solving state, with all the previously made decisions and deductions, which allows the procedure to come back to the saved state, i.e. backtrack, discarding all the changes made after it.

The DPLL algorithm works by taking the variables of a given formula, one after the other, to decide a truth value to assign to them. If with the assigned values to the variables the formula evaluates to true, then it is satisfiable. Otherwise, if after one of the decisions, the formula evaluates to false, then a contradiction was found, and the procedure backtracks to before making the decision to choose a different value. If that also leads to a contradiction, the procedure backtracks to the previous decision and so on, if all the combinations of decisions lead to contradictions, then the formula is unsatisfiable.

The CDCL algorithm extends the DPLL algorithm by improving the decision-making through the introduction of additional solving strategies such as learning, backjumping, and restarting. Learning is a process through which clauses are added to the formula to decrease the number of unnecessary decisions, these learned clauses are the ones that lead to conflicts. Backjumping, also called non-chronological backtracking, consists in backtracking until a backtracking point of a decision on a variable that is part of the learned clauses from the conflict is encountered, the purpose being to skip making decisions on variables that are irrelevant to the conflict, but in some cases, skipping already made decisions is not helpful [95]. Restarting is the process through which all the previous decisions and variable assignments are discarded, while the learned clauses are kept, allowing the solver to restart with the benefit of the learned clauses and without additional context.

Although the CDCL algorithm is the most popular and the one that has shown some of the best results in practice while being sound and complete, incomplete approaches also exist and have their own advantages and disadvantages. Notably, local search based approaches [65, 111] are often more efficient than complete methods for some randomly generated satisfiable problems, as well as some very large satisfiable problems. The approaches usually work on problems in CNF. They start by choosing a random interpretation for all the boolean variables in the problem. If the problem is not satisfiable with this first interpretation, then a clause that evaluates to false is selected, and the value of one of its boolean variables is flipped from true to false or vice versa, until a satisfying solution is found.

Local search based approaches tend to have a bias for satisfiable problems as they are search based and more efficient for model generation, in fact, they are often not only incomplete but also unable to determine that a problem is unsatisfiable. Therefore, for program verification needs, complete approaches such as DPLL and CDCL are used.

SAT solvers are software that implement SAT solving algorithms and are dedicated to solving SAT formulas. Since the introduction of the CDCL algorithm, their performance has greatly improved [22] as illustrated in Figure 2.6. In fact, the implementations of many modern SAT solvers are based on the CDCL algorithm as it has shown much better practical performance compared to other algorithms [83]. The first CDCL SAT solvers were GRASP [82], Chaff [89], and Berkmin [60], these were superseded by more modern ones such as MiniSat [50], CryptoMiniSAT [115], Glucose [10], CaDiCaL [20],

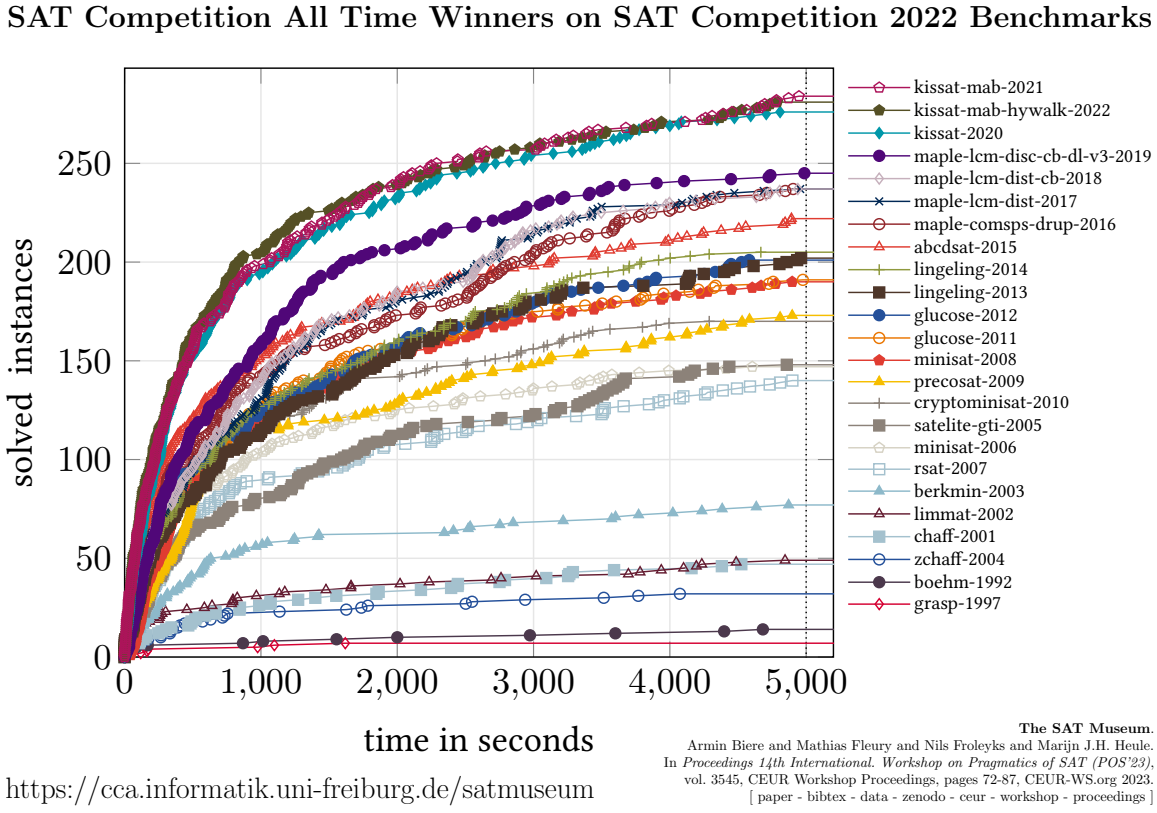

Figure 2.6: Comparison of various versions of state-of-the-art SAT solvers on the benchmarks of the SAT Competition 2022.

$$
\begin{aligned}
\langle \text{formula} \rangle &::= \; p[(\langle \text{term} \rangle_{\langle \text{sort} \rangle}[, \langle \text{term} \rangle_{\langle \text{sort} \rangle}]^*)]? \text{ with } p \in \Sigma^F_{bool} \\
&\mid \neg \langle \text{formula} \rangle \\
&\mid \langle \text{formula} \rangle \langle \text{binop} \rangle \langle \text{formula} \rangle \\
&\mid \forall \langle \text{variable} \rangle [, \langle \text{variable} \rangle]^*. \; \langle \text{formula} \rangle \\
&\mid \exists \langle \text{variable} \rangle [, \langle \text{variable} \rangle]^*. \; \langle \text{formula} \rangle \\
\langle \text{binop} \rangle &::= \; \wedge \mid \vee \mid \implies \mid \iff \\
\langle \text{term} \rangle &::= \; f[(\langle \text{term} \rangle_{\langle \text{sort} \rangle}[, \langle \text{term} \rangle_{\langle \text{sort} \rangle}]^*)]? \text{ with } f \in \Sigma^F \\
&\mid \langle \text{variable} \rangle_{\langle \text{sort} \rangle} \\
&\mid c \in D_{\langle \text{sort} \rangle} \\
\langle \text{variable} \rangle &::= v \in V \\
\langle \text{sort} \rangle &::= \tau \in \Sigma^S
\end{aligned}
$$

Figure 2.7: The grammar of many-sorted first-order logic.

and Kissat [21]. Although they are all based on the CDCL algorithm, each has its own strategies and heuristics that differentiate them from one another in performance.

The improvement in the performance of SAT solvers has been accompanied by the growth of their use cases. Indeed, many tasks that can be encoded as propositional logic problems can be delegated to SAT solvers. Some cases in which SAT solvers are used include: hardware and software verification [38, 66], notably with SAT-Based bounded model checking [24, 69], planning [51, 106], and cryptography [115].

2.2 Many-Sorted First-Order Logic

Propositional logic is not powerful enough to express concepts such as "all objects satisfy some property", or "there exists an object that satisfies property P and not property Q". First-order logic (FOL), also called predicate logic, is necessary to express such statements. This section presents many-sorted first-order logic (MSFOL). It is a more convenient variant of FOL that allows mixing differently sorted terms in the same formulas. It also supports rank-1 polymorphism, i.e. functions can be defined generically to operate over sort variables which can be any sort, with the sort variables universally quantified only at the outermost level of the type signatures of the functions.

In the TPTP world [118], the format in which MSFOL is expressed is called TFF for typed first-order form, or TFF1 [26] for TFF with rank-1 polymorphism. The TPTP world is a standard infrastructure for the research, development, and deployment of automated theorem proving systems. TPTP [117] stands for "Thousands of Problems for Theorem Provers" which is a vast library of theorem proving benchmarks.

Figure 2.7 illustrates the grammar by which MSFOL syntaxes are defined, with Σ^S as the set of sorts, V the set of variable symbols, $D_{\langle \text{sort} \rangle}$ as the sort domain of a given sort, Σ^F the set of function symbols and Σ^F_{bool} the set of predicate symbols. In this grammar,

three main kinds of expressions can be identified, sorts, terms and formulas. Sorts, also called types in programming languages, are a way to identify terms by the domain to which their interpretation must belong.

Definition 2.9 (Sorts)**.** *Sorts are a way to categorize expressions by the domain to which their interpretation belongs. They are also called types in programming languages.*

Definition 2.10 (Sort domain)**.** *To each sort τ is associated a sort domain D_τ, also called a concrete domain, which represents the set of possible values that an expression of that sort can have. Every sort domain D_τ is disjoint from every other sort domain $D_{\tau'}$.*

Example 2.2. *The sort domain of boolean terms is $D_B = \{\mathsf{true}, \mathsf{false}\}$, and the sort domain of integer terms is $D_I = \mathbb{Z}$, with $\mathbb{Z}$ as the set of integers.*

There are three main kinds of sorts:

- Simple sorts: which are either interpreted, i.e. they have a defined associated sort domain, such as booleans, integers or reals, or uninterpreted, in which their sort domain is not predetermined.
- Parametrized sorts $T(\tau_0[, \tau_i]*)$: also called sort constructors, they are sorts that need to take other sorts as parameters to be constructed. For example the sort $\mathsf{Array}(\mathsf{I}, \mathsf{E})$ has two sort parameters I and E.
- Polymorphic sort variables: universally quantified sort variables, that can be any other sort. Since rank-1 polymorphism is supported, sort variables are bound at the top-level of the formula in which they appear.

Definition 2.11 (Function and predicate symbols)**.** *Function symbols represent mappings from an n-tuple of terms into another term, function symbols have sorts of the form: $[\tau_i \rightarrow] * \tau$, with i going from 1 to n, $\tau_1, \ldots, \tau_n$ as the sorts of the arguments of the function, and τ as the sort of the returned term. If the function symbol is of arity 0, i.e. $n = 0$, then the function is an uninterpreted constant symbol of sort τ. When τ is the sort of booleans, then the function symbol is called a predicate symbol.*

Every term has a sort, and there are three kinds of terms:

- Variables: which are symbols that are bound by either universal or existential quantification, and which do not have a fixed interpretation.
- Constants: which are specific elements from a given sort domain.
- Applications of function symbols to as many terms as their arity.

Definition 2.12 (Uninterpreted symbol)**.** *A symbol of which the interpretation is not constrained by any axiom.*

In addition to the logical operations in propositional logic, formulas in first-order logic can also be:

- Applications of predicate symbols to as many terms as their arity.

$$\text{resolution-rule}\frac{C_1 \vee L_1 \qquad C_2 \vee \neg L_2}{\{C_1 \vee C_2\}\theta}(\text{with } \{L_1\}\theta = \{L_2\}\theta)$$

Figure 2.8: The resolution rule in FOL.

- Universal quantifiers: which state that a given formula $\forall x.\ \psi(x)$ is true if $\psi(c)$ is true for any value c in the domain D.
- Existential quantifiers: which state that a given formula $\exists x.\ \psi(x)$ is true if $\psi(c)$ is true for at least one value c in the domain D.

The distinction between predicates and terms in MSFOL is not as relevant as it is in FOL. Formulas can simply be considered as boolean terms, the distinction is there to differentiate between what can be solved, which are formulas for which the satisfiability can be checked, and what is used by the formulas which are other terms. Therefore, in Figure 2.7, $\Sigma^F_{bool} \subseteq \Sigma^F$.

2.2.1 Solving FOL

Contrary to propositional logic where model generation is expected when a problem is satisfiable, in (MS)FOL the goal is usually to prove the validity of a formula. Given a formula ψ, proving that ψ is valid comes down to determining that $\neg\psi$ is unsatisfiable. That can be done in various ways [28, 55], two of which are natural deduction and the resolution method.

Natural deduction consists in applying sound inference rules which introduce or eliminate logic operators and quantifiers until a contradiction is deduced proving validity. Natural deduction can be used through proof assistants such as the Rocq prover [121] (formerly known as the Coq proof assistant), the Lean theorem prover [93] and Isabelle [100].

The resolution method [107] is an automated proof technique for FOL. This technique is mainly based on the resolution rule which is applied repeatedly until a contradiction is found, or the rule cannot be applied anymore, if it was already applied on all clauses on which it could have been applied. This method is complete for validity, but may not terminate if the problem is not valid.

The resolution rule in FOL, presented in Figure 2.8 states that given the formulas $C_1 \vee L_1$ and $C_2 \vee \neg L_2$, such that there exists a substitution θ that makes $\{L_1\}\theta$ and $\{L_2\}\theta$ syntactically equal, then the formula $C_1 \vee C_2$ on which the substitution θ is applied can be deduced.

Before applying the resolution rule, it is necessary to first eliminate quantifiers and then put the problem in CNF. Existential quantifiers are eliminated through Skolemization, which is a method that consists in introduction Skolem functions for each existentially quantified variable. For each formula of the form $\forall x_1, x_2, \ldots, x_n.\ \exists y$, Skolemization consists in:

- Introducing a fresh function symbol f_y (called the Skolem function) that takes as arguments the terms of the sorts of $x_1, x_2, \ldots, x_n$ and returns a term of the same sort as y.

- Substituting all occurrences of y with $f_y(x_1, x_2, \ldots, x_n)$

Skolemization is a way to tie the values of existentially quantified variables to the universal quantifiers that precede the existential quantifier.

Universal quantifiers are eliminated more straightforwardly by removing the quantifiers and keeping the quantified variables as free variables, and considering that all free variables are universally quantified. Which can be done since the goal is to deduce unsatisfiability, i.e. the formula cannot be satisfied with any interpretation of the free variables, which comes down to considering them as universally quantified. Once quantifiers are eliminated, the problem can be put in CNF and the resolution rule can be applied.

The resolution method is particularly well suited for implementation in software. Popular automated theorem provers (ATPs), which are used to automatically prove the validity of first-order logic formulas, like the E theorem prover [110] and Vampire [74], are based on extensions of the resolution method.

2.2.2 First-Order Theories

Problems encoded in FOL can often share many symbols with their corresponding semantics. These symbols with their semantics define some basic foundations of common first-order theories that are often needed, these include integer and real arithmetic theories.

Definition 2.13 (Signature)**.** *A signature Σ in First-Order Logic is the set of defined non-logical symbols composed of:*

- *The set of sorts Σ^S.*
- *The set of constant symbols Σ^C.*
- *The set of functions Σ^F.*

Definition 2.14 ($\Sigma-$structure)**.** *Given a signature Σ, a $\Sigma-$structure is an interpretation of all the symbols in the signature Σ and all the expressions that can be constructed from them without free variables.*

Definition 2.15 (Theory)**.** *A theory $\mathcal{T}$ is defined by the pair $(\Sigma, \mathcal{S})$, where Σ is the signature, and $\mathcal{S}$ is the semantics, expressed as a class of $\Sigma-$structures that interprets the symbols in Σ.*

The semantics of some theories can be defined by a set of axioms that restrict the interpretation of the symbols of the theories, that is notably the case in the theory of equality and uninterpreted function (Figure 2.12), and the theory of arrays (Figure 2.12). One limitation of axiomatizing the semantics of a theory is that it cannot have a unique model, model-generation is in general hard with the presence of axiomatizations which often use quantifiers.

Alternatively, it is possible to also have theories for which the semantics are mathematically defined and built-in allowing them to have a fixed model. The study of these theories and how to efficiently reason over them lead to the development of the field of Satisfiability Modulo Theories.

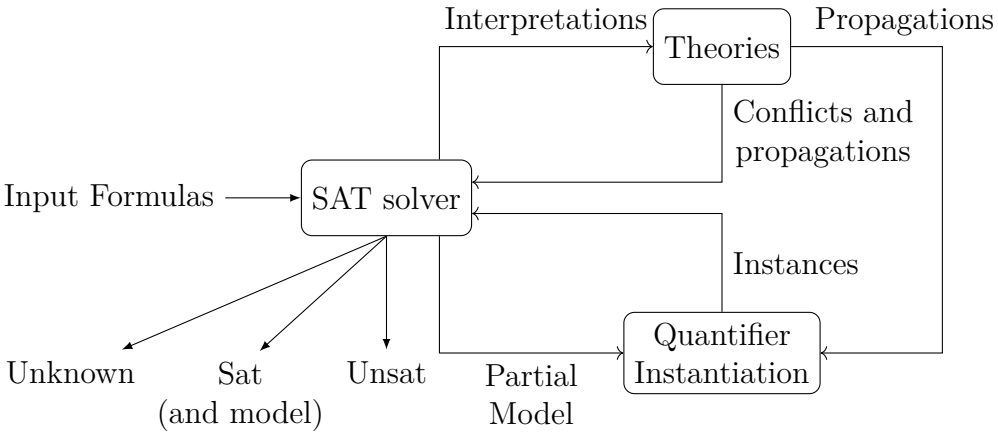

Figure 2.9: A schematic architecture of an SMT solver.

2.3 Satisfiability Modulo Theories

Since the early 2000s, Satisfiability Modulo Theories (SMT) [15] has seen significant growth in popularity and applications, both in academia and industry. SMT is based on MSFOL, as it allows both propositional reasoning and reasoning over first-order theories. The main difference between SMT and MSFOL is that in SMT there are built-in theories with defined interpretations and these theories have dedicated decision procedures to reason over them, while in MSFOL theories are axiomatized.

These decision procedures are combined with one another using theory combination techniques, notably the Shostak [114], Nelson-Oppen [97] and CDSAT [30] theory combination methods, allowing soundness to be maintained through the combined reasoning. The integration of a theory $\mathcal{T}$ solver with a SAT solver is often done through the DPLL($\mathcal{T}$), also called CDCL($\mathcal{T}$), approach on which most state-of-the-art SMT solvers are based [11, 91].

The expressiveness of SMT solvers is one of the main drivers of their industrial and academic uses, since they work on first-order logic formulas, making the encoding of logical formulas in SMT much easier than doing so in propositional logic. In addition, multiple theories, such as fixed-size bit-vectors, floating-point arithmetic, algebraic data types, sequences, and strings, are designed in a way that makes it easier to use them to represent data types and data structures from higher-level programming languages, making the usage of SMT solvers for program verification more convenient.

In fact, SMT solvers like Alt-Ergo [41], bitwuzla [98], cvc5 [11], and Z3 [91] are relied upon by various tools to prove logical formulas or to generate counterexamples, notably program verification tools such as Frama-C [73], Why3 [54], SPARK-2014 [86], TIS-Analyzer [43], and Dafny [79], symbolic execution tools such as KLEE [35], BINSEC [44], and Owi [8], and model checkers such as CBMC [76].

Figure 2.9 illustrates the architecture of an SMT solver and provides a simplified description of its operation. It takes SMT formulas in supported theories as input. The SAT solver makes decisions on boolean variables and provides the interpretation to the theory reasoning module, which, through theory combination, dispatches purified terms based on their corresponding theory to their respective decision procedures.

Definition 2.16 (Purified terms)**.** *Purified terms are produced through purification. Given a theory T and a term t, purifying the term t for the theory T consists in replacing all its sub-terms that don't belong to the theory T with fresh variables while keeping the sub-terms that belong to the theory T, ensuring that the decision procedure for the theory T only sees the terms that belong to the theory.*

The decision procedures perform theory reasoning and either deduce new propagations to share with the rest of the system, typically in the form of new equalities, inequalities, and clause assignments, or detect a conflict. The SAT solver uses this information to learn clauses and backtrack. This part is called the ground reasoning part because it handles the unquantified parts of the input formulas.

The quantifiers are handled through the quantifier instantiation engine, which attempts to instantiate the quantifiers by selecting values for the quantified variables. The resulting instances are then added to the system to be handled by the ground reasoning part. The reasoning process concludes when there are no quantifiers left to instantiate and the SAT solver arrives at a satisfied problem, in which case it answers "sat". Alternatively, if all decisions made by the SAT solver lead to conflicts, it answers "unsat". It can also answer "unknown" since first-order logic is semidecidable, meaning the satisfiability of some problems simply cannot be determined. A solver can also answer "unknown" if it does not use a complete reasoning procedure.

Quantifier instantiation and decision-making by the SAT solver are two commonly costly steps in SMT reasoning. Therefore, decreasing the number of new terms created during reasoning and avoiding the application of redundant rules while maintaining efficiency, as well as completeness in some cases when working on decidable theories, are common optimization approaches [37, 94]. Another optimization involves improving quantifier instantiation, which allows for smarter instantiations that are performed only when needed in ways that make the chosen values for the variables more likely to lead to a result, usually unsatisfiability [34, 90].

2.3.1 The SMT-LIB initiative

The SMT-LIB standard [16] is the standard language for encoding SMT formulas. It was developed and is maintained by the SMT-LIB initiative [13]. The SMT-LIB initiative also defines a set of standardized theories with precise syntaxes and semantics, making it possible to use different SMT solvers that follow the standard language on problems written in it. The SMT-LIB initiative also gathers and maintains benchmarks of SMT problems [103], which originate from academia, industrial use cases, or were synthetically crafted. These benchmarks are used in the SMT-COMP [12], a competition in which SMT solvers are compared based on their performance on SMT problems categorized into different SMT-LIB logics.

Definition 2.17 (Theory Fragment)**.** *A fragment of a theory T is syntactically restricted subset of the formulae of the theory T that is less expressive than the full theory but can be decidable or easier to reason over.*

SMT-LIB logics are syntactic restrictions of theories defined as fragments of one or multiple theories. Some logics, for example, restrict a theory to a quantifier-free fragment,

```
:sorts ((Bool 0))

:funs ( (true Bool)
        (false Bool)
        (not Bool Bool)
        (=> Bool Bool Bool :right-assoc)
        (and Bool Bool Bool :left-assoc)
        (or Bool Bool Bool :left-assoc)
        (xor Bool Bool Bool :left-assoc)
        (par (A) (= A A Bool :chainable))
        (par (A) (distinct A A Bool :pairwise))
        (par (A) (ite Bool A A A))
      )
```

Listing 2.1: The SMT core theory.

such as the QF_AX logic, which corresponds to the quantifier-free fragment of the array theory. Others restrict the symbols that can be used in the theory or the different ways in which they can be used, such as the fragments of linear integer arithmetic LIA and integer difference logic IDL [99].

While expressiveness is one of the main advantages of SMT, it usually also implies more complex decision procedures and reasoning approaches for the theory. It is also possible to change the decidability of a theory or logic by making it more expressive. For example, the theory of Real Difference Logic is decidable but becomes undecidable when combined with uninterpreted unary predicates [27].

2.3.2 The Core Theory

The Core Theory in the SMT-LIB is represented in Listing 2.1. It serves as a common base that is included in all SMT logics. It is used to represent propositional logic and extends it with the following interpreted symbols:

- = (The binary equality predicate): It is applied to two terms of any sort A and evaluates to true if they are equal, and false otherwise.
- distinct (The binary inequality predicate): It is applied to two terms of any sort A and evaluates to true if they are disequal, and false otherwise.
- ite (If-Then-Else): It takes a boolean term called the condition and two terms of any sort A, called respectively the consequence and the alternative. If the condition is true, then the term is equal to the consequence, otherwise, it is equal to the alternative.
- xor (The "exclusive or" operation): evaluates to true when the two boolean arguments have distinct truth values.

$$\forall x.x = x \qquad \text{(reflexivity)}$$

$$\forall x, y.x = y \implies y = x \qquad \text{(symmetry)}$$

$$\forall x, y, z.x = y \wedge y = z \implies x = z \qquad \text{(transitivity)}$$

$$\forall f, x_1, ..., x_n, y_1, y_n.x_1 = y_1 \wedge ... \wedge x_n = y_n \implies f(x_1, ..., x_n) = f(y_1, ..., y_n) \qquad \text{(congruence)}$$

Figure 2.10: Semantics of the EUF logic

2.3.3 Equality and Uninterpreted Functions

The uninterpreted functions logic (UF) is an extension of the Core theory presented in the previous section that allows expressing uninterpreted functions and sorts. The combination of equality, boolean operators and uninterpreted functions and sorts is often referred to as the equality and uninterpreted functions logic (EUF) [75].

Uninterpreted functions, or uninterpreted constant symbols when the functions have no arguments, are declared function symbols that do not have predefined interpretations. And uninterpreted sorts are sorts for which the associated sort domain is not predefined.

Unless uninterpreted functions are explicitly constrained, they can have any interpretation of the appropriate sort, while uninterpreted sorts, in the satisfiable case, need to have an interpretation that associates them with a non-empty sort domain.

Figure 2.10 illustrates the axioms that describe the semantics of EUF. Reasoning over EUF is usually based on the Union-Find [119] data structure, as it is commonly used to represent equivalence relations, which are by definition reflexive, symmetric, and transitive, with equality being an example of them.

In addition to the axioms of equivalence, the congruence axiom is necessary to determine that two applications of the same function or predicate symbol to the same arguments are equal. The resulting decision procedure that includes the axioms in Figure 2.10 is known as Congruence Closure [96, 105].

The union-find data structure is used to group terms that are equal to one another into the same set, these sets are referred to as congruence classes. Each set has a representative term that is usually used to hold information about the congruence classes represented by the set of terms. When two terms that are not in the same set are propagated as equal, then their respective congruence classes (sets of terms) are merged together. If a term was not added to any set, it is considered to be in a singleton set containing only that term.

Inequality is also usually handled by congruence closure. This can be done by simply adding special edges between congruence classes to indicate that they are distinct from one another. When two congruence classes are about to be merged, the existence of such an edge between them signifies that a contradiction has been detected, leading to a conflict.

```
:sorts ((Int 0))

:funs ( (NUMERAL Int)
        (- Int Int)                    ; negation
        (- Int Int Int :left-assoc) ; subtraction
        (+ Int Int Int :left-assoc)
        (* Int Int Int :left-assoc)
        (div Int Int Int :left-assoc)
        (mod Int Int Int)
        (abs Int Int)
        (<= Int Int Bool :chainable)
        (<  Int Int Bool :chainable)
        (>= Int Int Bool :chainable)
        (>  Int Int Bool :chainable)
      )
```

Listing 2.2: The SMT theory of integers.

$$(p \implies x \leq y) \wedge (\neg p \implies x > y) \wedge (p \wedge x > y)$$

Figure 2.11: SMT formula example.

2.3.4 The Theory of Integers

Another example of an SMT theory is presented in Listing 2.2, which illustrates the SMT theory of integers. In this theory, the sort domain of the integer terms is the set of integers $\mathbb{Z}$. The mathematical symbols $-$, $+$, and $*$ have their usual interpretations when working with integers, as do the comparison operations <=, <, >=, and >. The symbols div, mod, and abs represent, respectively, integer division, integer remainder, and integer absolute value.

Figure 2.11 illustrates an SMT formula that uses both boolean operators ($\implies$ and $\wedge$) as well as integer arithmetic operators ($\leq$ and $>$). SMT solvers with a decision procedure for the linear integer arithmetic (LIA) logic can solve such problems, whereas it is not possible to encode such problems in a SAT solver. Listing 2.3 shows the same SMT formula written in the SMT-LIB standard.

2.4 Constraint Programming

Constraint programming (CP) [108] is a paradigm for solving combinatorial problems using automated deduction and search algorithms. Such problems are usually expressed as constraint satisfaction problems (CSPs). CSPs consist of a set of variables, each with its own sort domain, and a set of constraints, each applying over one or multiple variables, which serve to restrict the domains of the variables. CP is used to find a solution that satisfies the constraints over the variables in the form of an interpretation of the variables, if such a solution exists.

Definition 2.18 (Constraint satisfaction problems (CSPs))**.** *A constraint satisfaction*

```
(set-logic QF_LIA)

(declare-const p Bool)
(declare-const x Int)
(declare-const y Int)

(assert (-> p (<= x y)))
(assert (-> (not p) (> x y)))
(assert (and p (> x y)))

(check-sat)
```

Listing 2.3: SMT formula example in the SMT-LIB standard.

problem is defined as a triplet $\langle V, D, C \rangle$*, where* $V = \{v_1, \cdots, v_n\}$ *is a set of variables,* $D = \{D_1, \cdots, D_n\}$ *is the set of domains where each domain* D_i *is the set of possible values of the variable* v_i*, and* $C = \{C_1, \cdots, C_n\}$ *is a set of constraints, where each constraint* C_i *is over a subset of variables in* V *and restricts the combinations of the domains of the variables it involves.*

CP research started in the 1960s and 1970s, stemming from early developments in Artificial Intelligence [56]. The introduction of Prolog [39] was notably a significant milestone, as it was an early form of a constraint programming language, solving constraints over terms and variables using its unification algorithm. Later, in the 90s, in parallel to the introduction of CDCL in the SAT community, CP also progressed with concepts such as global constraints [123] and search heuristics [122]. The emergence of SMT allowed for more bridging between SAT and SMT on one side and CP on the other [5, 53, 102].

Many concepts present in SAT and SMT are also present in CP. For example, decisions in CP work similarly to how they work in SAT and SMT, one difference lies in how they are used. In SAT and CDCL based SMT, decisions are made on boolean literals during satisfiability checking, while in CP, they are also made on terms that have other sorts as a way to prune their domains while searching for a solution. Another difference between is within the conflicts: while in SAT and CDCL based SMT they occur on boolean literals coming from the original problem, in CP they are more general, as they can come from propagations, variable assignments, or global constraints. These techniques, such as decisions and assignments on non-boolean terms, were also brought to SMT in newer SMT techniques than the CDCL based ones such as GDPLL [87], MCSAT [92] and CDSAT [30].

Backtracking in CP works similarly to backtracking in SAT and SMT: after a decision is made, if it leads to a contradiction, it is necessary to backtrack to make another decision. When it comes to learning, while it is ubiquitous in SAT and SMT solving, it is not always present in CP, but there are approaches in CP which allow learning new constraints that prune the search space and the decisions that can be made such as "nogoods" [72] which are learned from contradictions.

Another important technique used in CP is constraint propagation. It is the process through which constraints are used to restrict the domains of variables while maintaining consistency. Constraint propagation consists of ensuring that when a constraint C_1 is

applied to a variable v, the effect of the constraint C_1 on the domain of the variable v is taken into account by all other constraints C_i that are applied to the variable v. Example 2.3 shows a case in which a set of constraints is not satisfiable, as it results in an empty domain for one of the variables.

Example 2.3. *Given the integer variables x, y, and z, and the constraints $c_1 : x \geq 3$, $c_2 : y \geq 2$, $c_3 : x + y = z$, and $c_4 : z < 5$, propagating the constraints results in:*

- *c_1 sets the domain of x to $[3; +\text{inf}]$.*
- *c_2 sets the domain of y to $[2; +\text{inf}]$.*
- *c_3 sets the domain of z to $[5; +\text{inf}]$.*
- *c_4 results in an empty domain for z, which means that the problem is unsatisfiable.*

Although constraint propagation is a notion tied to constraint programming, it is also used in SMT, usually indirectly through theory reasoning and theory combination, but some SMT solvers, notably Alt-Ergo [5], explicitly incorporate domains and constraint propagators in their reasoning.

Many real-world problems can be expressed as CSPs. Therefore, CP has many applications [126]. Such applications include scheduling [32], planning [18], resource allocation [67], data mining and machine learning [47], and software [61] and hardware [125] verification.

2.4.1 Abstract Domains in Constraint Programming

Traditionally in CP, as well as in MSFOL, domains refer to sort domains or concrete domains, i.e. the domains associated with sorts and that represent the set of all possible values terms from a given sort can have. But multiple efforts have been made to extend the notion of domain to abstract domains [101, 128], as the ones that are used in abstract interpretation [42].

Abstract domains allow propagating constraints not only over the concrete set of possible values that a term can have, but also over its properties. When working on arithmetic terms for example, some abstract domains that can be used including the domain of polynomials, which associates arithmetic terms to their polynomial representation and which can allow for more constraint propagations between the term and its sub-terms. Such domains can also allow deducing contradictions earlier and without computing the concrete domains of the terms which is potentially costly [31].

Example 2.4 illustrates an example of the usage of an abstract domain during constraint propagation. In the example, it is used to prune the concrete domain of x and find a precise value for it.

Example 2.4. *Given the integer variable $x \in [1; 3]$ and an abstract domain of parity $\mathcal{D}_p = \{\mathsf{odd}, \mathsf{even}\}$ which associates to each arithmetic term its parity. If the constraint $x \bmod 2 = 0$ is propagated, then x's parity domain is updated to $\mathcal{D}_p(x) = \mathsf{even}$, from it and the fact $x \in [1; 3]$, $x = 2$ can be deduced from.*

For convenience, "domains" is used to refer to both kinds of domains, the nature of the domain is specified only when it is warranted.

```
:sorts ((Array 2))

:funs ( (par (X Y) (select (Array X Y) X Y))
        (par (X Y) (store (Array X Y) X Y (Array X Y)))
      )
```

Listing 2.4: Signature of the theory of arrays.

$$\forall a, i, j, v. i = j \implies \mathsf{select}(\mathsf{store}(a, i, v), j) = v \tag{idx}$$

$$\forall a, i, j, v. \\ i \neq j \implies \mathsf{select}(\mathsf{store}(a, i, v), j) = \mathsf{select}(a, j) \tag{select-over-store}$$

$$\forall a, b. a = b \vee \exists (i : \mathsf{I}). \mathsf{select}(a, i) \neq \mathsf{select}(b, i) \tag{ext}$$

Figure 2.12: Semantics of the theory of Arrays

2.5 The theory of Arrays

The SMT theory of arrays is based on McCarthy's arrays, which were introduced in 1962 [85]. It is a well-studied and explored theory, and many decision procedures have been developed for it [34, 37, 94]. The theory of arrays is frequently used in program verification to represent array-like data structures from programming languages, and is also used to represent and reason about memory models, notably in low-level programming languages. It is worth noting that arrays in SMT are different from arrays in most programming languages, since in SMT they are infinite total mappings of indices to values instead of finite ordered collections of values as is the case in programming languages.

The signature of the SMT theory of arrays, as standardized in the SMT-LIB, is presented in Listing 2.4. The arrays in this theory are represented as mappings of terms from a sort of indices to terms from a sort of values, represented respectively by the X and Y sorts in Listing 2.4. The theory's signature also shows that it supports two functions:

- select: given an array a and an index i, returns the value associated with the index i in the array a.
- store: given an array a, an index i, and a value v, returns a copy of the array a in which the element associated with the index i is the value v.

Figure 2.12 formalizes the semantics of the theory of arrays. The (idx) and (select-over-store) axioms assess the semantics of the store operation by stating how to interpret the application of a select function over a store function. The (ext) axiom, on the other hand, represents the extensionality axiom. The SMT theory of arrays is also referred to as the theory of arrays with extensionality, and in this context, extensionality means that two arrays are equal when they contain the same elements at every index, which is what is stated in the axiom (ext).

The unsatisfiable SMT problem manipulating arrays presented in Listing 2.5 asserts that, for an array a, an index i and a value v, $\mathsf{select}(a, i)$ is different from v. It also

```
(set-logic ALL)
(declare-sort I 0)
(declare-sort E 0)
(declare-const i I)
(declare-const e E)
(declare-const a (Array I E))

(assert (distinct (select a i) e))
(assert (= (select (store a i e) i) (select a i)))

(check-sat)
```

Listing 2.5: Example of an SMT problem using arrays.

asserts that $\mathsf{select}(\mathsf{store}(a, i, v), i)$ is different from v, which, by applying the (idx) rule, is determined to be false. Proving that the problem is unsatisfiable.

2.5.1 Array Property Fragment

The quantifier-free theory of arrays is decidable, but as first-order logic generally is not, neither is the theory of arrays with quantifiers. Bradley et al. explored the topic of the decidability of the theory of arrays with quantifiers and defined the array property fragment [9, 34], which is a decidable fragment of the quantified theory of arrays.

The array property fragment imposes some syntactic restrictions on the theory of arrays. The theory of indices in this fragment has to be the Presburger arithmetic theory, which consists only of the symbols $\{0, 1, +, -, =, <\}$. It also restricts quantification over arrays to the subset $\exists^*\forall^*$, where the universal quantifiers $\forall$ only quantify index variables.

An array property is defined as a formula of the form:

$$\forall \bar{i}.\ G(\bar{i}) \implies F(\bar{i})$$

where $\bar{i}$ are index variables. $G(\bar{i})$ is an index guard that can only contain conjunctions or disjunctions of terms of the forms $\alpha \leq \beta$ and $\alpha = \beta$, where α and β are either index ground terms, universally quantified index variables, or existentially quantified index variables multiplied by a constant. And $F(\bar{i})$ is a constraint on the elements that is also syntactically restricted, such that all occurrences of quantified index variables must be as a second argument to a select application, and select applications cannot be nested, therefore, $\mathsf{select}(\_, \mathsf{select}(\_, \_))$ would not be allowed.

Some interesting properties that can be expressed in this fragment include:

- Equality between two arrays a and b:

$$\forall i.\ \mathsf{select}(a, i) = \mathsf{select}(b, i)$$

- Bounded equality between two arrays a and b within the bounds $[l, u]$:

$$\forall i.\ l \leq i \leq u \implies \mathsf{select}(a, i) = \mathsf{select}(b, i)$$

$$\text{store-elim}\ \frac{a = \mathsf{store}(b, i, v)}{\mathsf{select}(a, i) = v \wedge \forall j.\ j \neq i \implies \mathsf{select}(a, j) = \mathsf{select}(b, j)}$$

$$\text{exists-elim}\ \frac{\exists \bar{i}.\ F[\bar{i}]}{F[\bar{k}]} \quad \text{where } \bar{k} \text{ is a vector of fresh index variables.}$$

$$\text{forall-elim}\ \frac{\forall \bar{i}.\ G[\bar{i}] \implies F[\bar{i}]}{\bigwedge_{i \in I^n} G[\bar{i}] \implies F[\bar{i}]}$$

Figure 2.13: Inference rules for the Array Property Fragment decision procedure.

- Sortedness of an array a within the bounds $[l, u]$:

$$\forall i, j.\ l \leq i \leq j \leq u \implies \mathsf{select}(a, i) \leq \mathsf{select}(a, j)$$

The decision procedure for this fragment of the theory of arrays works by reducing the problem of deciding satisfiability in the theory of arrays to deciding the satisfiability of a problem in EUF, Presburger arithmetic, and the array element theories. It also reduces universal quantifiers over indices into finite conjunctions by constructing a finite set of indices such that examining only the indices in this set is sufficient to decide satisfiability.

Figure 2.13 illustrates the inference rules used in the array property fragment decision procedure. To solve problems in this theory fragment, they first need to be converted to negation normal form, which is a form where negations only appear in literals.

The store-elim rule is then applied exhaustively to eliminate all store terms and replace them with select terms. This is followed by the exhaustive application of the exists-elim rule, which eliminates existentially quantified index terms when the existential quantification is at the top of the formula, by replacing them with fresh index variables.

The reduction of universal quantifiers to finite conjunctions is done in three steps:

- Select a set I of index terms on which to instantiate all universally quantified indices. The set I contains:
  - All index terms t that occur as a second argument of a select application and are not universally quantified.
  - All index terms t that are constants, existentially quantified variables, addition of them or the multiplication of a constant with an existentially quantified variable.
  - A fresh index term λ that is defined as a constant index term different from all the index terms in I.
- Apply the forall-elim rule exhaustively, with n as the size of the vector of quantified variables $\bar{i}$, where $\bar{i} \in I^n$ means that the variables $\bar{i}$ range over all n-tuples of terms in I. This step eliminates all universal quantifiers over indices and replaces them with conjunctions over the set of the n-tuples of index terms in I.

$$(\mathsf{idx})\frac{a = \mathsf{store}(b, i, v)}{\mathsf{select}(a, i) = v}$$

$$\Downarrow\frac{a = \mathsf{store}(b, i, v) \qquad w = \mathsf{select}(a, j)}{i = j \vee \mathsf{select}(a, j) = \mathsf{select}(b, j)}$$

$$\Uparrow\frac{a = \mathsf{store}(b, i, v) \qquad w = \mathsf{select}(b, j)}{i = j \vee \mathsf{select}(a, j) = \mathsf{select}(b, j)}$$

$$(\mathsf{ext})\frac{a \qquad b}{a = b \vee \mathsf{select}(a, k) \neq \mathsf{select}(b, k)}$$

Figure 2.14: Basic decision procedure for the array theory.

- Reduce to the theories of EUF, Presburger arithmetic, and the element theory by associating each n-dimensional array variable a with an n-ary uninterpreted function f_a and replacing nested select applications $\mathsf{select}(\mathsf{select}(\mathsf{select}(a, i), \cdots), j)$ to a with calls to $f_a(i, \cdots, j)$.

2.5.2 Combinatory Array Logic

Another decision procedure for the theory of arrays that relies on the reduction of the array theory to EUF theory is the decision procedure used in Z3, which was developed by de Moura and Bjørner [94]. In their contribution, they present a naive decision procedure for the theory of arrays consisting of the instantiation of the theory's axioms, as well as optimizations that show better performance without losing completeness.

Figure 2.14 illustrates the inference rules of the basic decision procedure for the array theory. The rules idx and ext correspond to the axioms that have the same name in Figure 2.12. ext states that for any pair of arrays with the same sort, they are either equal if they contain the same elements at all indices, or they are distinct, which means that they have at least one index (called the disequality witness and represented by the fresh index variable k) at which the stored elements in the arrays are different. The rules $\Downarrow$ and $\Uparrow$ correspond to the select-over-store axioms in Figure 2.12 by defining two complementary cases for the axiom: $\Downarrow$, where the consequence of the axiom is inferred from $\mathsf{select}(\mathsf{store}(b, i, v), j)$ when $i = j$, and $\Uparrow$, where it is inferred from $\mathsf{store}(b, i, v)$ and $\mathsf{select}(b, j)$ when $i = j$. With this decision procedure, the axioms of the theory are instantiated to saturation, effectively reducing the array theory problems to EUF problems.

The first optimization is achieved by avoiding the redundant application of the inference rules $\Downarrow$, $\Uparrow$, and ext when their consequences are already present in the solver state. Additional optimizations to the array decision procedure are done by replacing the inference rules ext and $\Uparrow$ with restricted versions of them, presented in Figure 2.15.

Instead of the ext rule, two restricted versions are used: $\mathsf{ext}_{\neq}$, which introduces the disequality witness through the fresh index variable k when two arrays are known to be

$$\mathsf{ext}_{\neq}\ \frac{a \neq b}{\mathsf{select}(a,k) \neq \mathsf{select}(b,k)}$$

$$\mathsf{ext}_{r}\ \frac{(a : \mathsf{Array}(\mathsf{I},\mathsf{E})) \quad (b : \mathsf{Array}(\mathsf{I},\mathsf{E})) \quad a,b \subseteq \mathsf{foreign}}{a = b \vee a[k] \neq b[k]}$$

$$\Uparrow_{r}\ \frac{a = \mathsf{store}(b,i,v) \quad w = \mathsf{select}(b,j) \quad b \in \mathsf{non\text{-}linear}}{i = j \vee a[j] = b[j]}$$

Figure 2.15: Restricted extensionality and $\Uparrow$ inference rules for the array decision procedure, with k as a fresh variable.

disequal (instead of applying the ext rule), and ext_r, which restricts the application of the ext to arrays that are part of the foreign set of arrays. The foreign set comprises arrays that appear as the index argument of a select application to another array or appear as an argument of an uninterpreted function. As its name suggests, the foreign set is the set of arrays which are potentially used outside the theory of arrays and which need the application of the extensionality to reason over and propagate their equalities.

The $\Uparrow$ rule is replaced with $\Uparrow_r$, a restricted version that is applied only when the array b is in the non-linear set. An array is in the non-linear set if it is equivalent to two distinct store applications or to a store operation on an array that is already in the non-linear set. This set represents arrays which are not constructed from a simple chain of store applications and that have potentially been constructed from multiple different arrays, and on which it is therefore necessary to apply the $\Uparrow$ rule as it allows propagating elements between those arrays from which the "non-linear" array was constructed.

In addition to the decision procedure and the optimizations, the authors also presented extensions to the theory of arrays that can be reasoned about using the same decision procedure. These extensions include the constant array function, denoted by K, which creates an array where all elements have the same value, and the mapping function, denoted by map_f, which applies a function f element-wise to a set of n arrays that have the same index sort. These functions are formally defined as follows:

$$\begin{aligned}
&\forall v, i.\ \mathsf{select}(K(v), i) = v \\
&\forall a_1 : \mathsf{Array}(\mathsf{I},\mathsf{E}_1), \ldots, a_n : \mathsf{Array}(\mathsf{I},\mathsf{E}_n), i : \mathsf{I}. \\
&\qquad \mathsf{select}(map_f(a_1,\ldots,a_n), i) = f(\mathsf{select}(a_1,i), \ldots, \mathsf{select}(a_n,i))
\end{aligned}$$

The additional functions are notably used to simplify the representation of sets and bags (or multi-sets) using the theory of arrays.

2.5.3 Weakly Equivalent Arrays

Another interesting array decision procedure was developed by Christ and Hoenicke [37]. This decision procedure works by exploiting the notion of weak equivalence over arrays and using it to avoid unnecessary instantiations of the (select-over-store) and (ext) array

axioms described in Figure 2.12. Weak equivalence is represented using a weak equivalence graph that links arrays with indices on which they may differ, which are obtained by the chains of store applications that link the arrays.

Definition 2.19. *A Weak Equivalence Graph $G^W = (V, E)$ is an undirected graph in which the set of vertices V contains array terms that are linked by edges from the set E, which contains either:*

- $\leftrightarrow$*: Unlabeled edges that represent equivalence between array terms.*
- $\overset{i}{\leftrightarrow}$*: Edges labeled with an index i that link two arrays a and b when $a = \mathsf{store}(b, i, \_)$ or $b = \mathsf{store}(a, i, \_)$.*

Definition 2.20 (Weak equivalence)**.** *Two arrays a and b are weakly equivalent if there exists a path P between a and b in G^W, denoted by $a \overset{(P)}{\Leftrightarrow} b$. Consequently, a and b can only be different on a finite set of indices.*

Definition 2.21 (Weak equivalence modulo i)**.** *Weak equivalence modulo i between two arrays a and b, denoted $a \approx_i b$, is defined as:*

$$a \approx_i b := \exists P. a \overset{(P)}{\Leftrightarrow} b \wedge \forall j \in P. i \neq j$$

Definition 2.22 (Weak congruence modulo i)**.** *Weak congruence modulo i between two arrays a and b, denoted $a \sim_i b$, is defined as:*

$$a \sim_i b := a \approx_i b \vee$$
$$\exists a', b', j, k. a \approx_i a' \wedge b \approx_i b' \wedge i = j \wedge i = k \wedge \mathsf{select}(a', j) = \mathsf{select}(b', k)$$

The notions of weak equivalence, weak-equivalence modulo i and weak congruence modulo i are tied to one another. Intuitively, two arrays a and b are:

- Weakly equivalent, denoted $a \overset{(P)}{\Leftrightarrow} b$, if one can be rewritten as a finite chain of store applications over the other, which means that they can only differ on the finite set of indices P that are involved in those store applications.
- Weakly equivalent modulo i, denoted $a \approx_i b$, if $a \overset{(P)}{\Leftrightarrow} b$ and i is distinct from all the indices within the set P.
- Weakly congruent modulo i, denoted $a \sim_i b$, if $a \approx_i b$ and there exists arrays a' and b', such that $a \approx_i a'$ and $b \approx_i b'$ and $\mathsf{select}(a', i) = \mathsf{select}(b', i)$.

Definition 2.23. *$Cond(\_)$ is a function that computes a condition (a conjunction of equalities and inequalities) under which an equivalence, weak equivalence, or weak congruence holds. $Cond_i(\_)$ denotes the condition for a path that does not contain an edge labeled with the index i.*

$$Cond(a \leftrightarrow b) := a = b \qquad Cond_i(a \leftrightarrow b) := a = b$$
$$Cond(a \overset{j}{\leftrightarrow} b) := \mathit{true} \qquad Cond_i(a \overset{j}{\leftrightarrow} b) := i \neq j$$
$$Cond(a \approx_i b) := Cond_i(P) \text{ where } a \overset{(P)}{\Leftrightarrow} b \wedge \forall j \in P. i \neq j$$
$$Cond(a \sim_i b) := \begin{cases} Cond(a \approx_i b) & \text{if } a \approx_i b \\ Cond(a \approx_i a') \wedge i = j \wedge & \text{if } a \approx_i a' \wedge i \sim j \wedge \\ Cond(b \approx_i b') \wedge i = k \wedge & \quad b \approx_i b' \wedge i \sim k \wedge \\ \mathsf{select}(a', j) = \mathsf{select}(b', k) & \quad \mathsf{select}(a', j) \sim \mathsf{select}(b', k) \end{cases}$$

$$\textsf{select-over-weakeq}\ \frac{a \approx_i b \qquad i \sim j \qquad \textsf{select}(a,i) \qquad \textsf{select}(b,j)}{i \neq j \vee \neg Cond(a \approx_i b) \vee \textsf{select}(a,i) = \textsf{select}(b,j)}$$

$$\textsf{weakeq-ext}\ \frac{a \overset{(P)}{\Leftrightarrow} b \qquad \forall i \in P. a \sim_i b}{\neg Cond(P) \vee \bigvee_{i \in P} \neg Cond(a \sim_i b) \vee a = b}$$

Figure 2.16: The weakly equivalent array decision procedure inference rules.

The weakly equivalent array decision procedure consists of the inference rules in Figure 2.16 as well as the idx rule. The select-over-weakeq rule is a version of select-over-store that propagates select applications over chains of store applications between arrays that have select applications on them and on the same index. The weakeq-ext rule is a version of the ext rule that takes into account the weak equivalence graph. It does so by being applied when two arrays are linked and stating that they are either equal or they are not weakly congruent on at least one index from the path that links them together.

It is possible to optimize the decision procedure while maintaining soundness and completeness by decreasing the number of terms managed by it. This can be done if the element theory is stably infinite by not applying the idx rule for every store term, consequently not generating a new select term for every store store.

2.6 The theory of Sequences

The theory of arrays suffers from limitations that make using it to represent, reason about, and verify the properties of more complex data structures harder without significant extensions (additional functions and predicates) or axiomatization, eventually with quantifiers. The first limitation is the fact that arrays from this theory are not dynamically sized, since their size is determined by the number of inhabitants of the sort of indices, while most modern programming languages have dynamically sized arrays. Another limitation comes from the lack of expressiveness in the theory, since it only supports operations for selecting and storing one element at one index.

Thus, when it comes to verifying properties of a given data structure using SMT solvers, it is more convenient to have a tailored theory that clearly and concisely describes the semantics of the higher-level operations on that data structure. Not only does this make verification easier for the user, but it can also pave the way for more dedicated and efficient decision procedures for the theory. Examples of such theories are the theory of strings and the theory of sequences, which have both sparked a lot of interest in recent years.

Sequences are a common data structure in programming languages, although they may be known by different names and have various implementations. Sequences can have fixed sizes, like arrays in C, C++, Rust, OCaml, and Java, or they can be dynamic, like vectors in C++ and Rust, ArrayLists in Java, arrays in JavaScript, and lists in Python. In addition to the common array operations, such as storing and selecting values at an index, some languages support higher-level operations such as concatenation, slicing,

mapping, filtering, and folding.

The difference between the standard SMT theory of arrays and the theory of sequences is that sequences can have different sizes, while arrays have a fixed size determined by the sort of the indices and the number of possible values it has. Sequences are always indexed by integers, whereas arrays do not have such a restriction on their index sort. Additionally, the signature of the theory of sequences is richer than that of the theory of arrays, it includes functions for concatenation, slicing, subsequence extraction, etc. The $\mathsf{nth}(s, i)$ function from the theory of sequences takes a sequence s and an index i, returning the value stored at the i-th index of the sequence, akin to the select function in the array theory. However, the mathematical interpretation of this function on sequences is partial to valid indices, which are those within the bounds of the sequence. As SMT is a total logic, such functions are totalized by considering the returned value when the index is not within the bounds of the sequences as uninterpreted.

The theory of sequences was first formalized by Bjørner et al. [25]. Several works have since explored its syntax and semantics [3, 25] and its decidability [57, 70]. Sheng et al. [113] developed calculi to reason over the variant of the theory of sequences that is implemented in cvc5 [11], the calculi are based on those developed for the theories of strings [19, 81], which have been generalized and adapted for sequence reasoning and combined with array reasoning [37]. The Z3 SMT solver [91] also supports the theory of sequences, although there are no published contributions detailing its reasoning techniques.

2.6.1 Existing theories

The original SMT theory of sequences, proposed by Bjørner et al. [25], is a generalization of the theory of strings to non-character values. State-of-the-art SMT solvers such as cvc5[1] [113] and Z3[2] support their own variations of the theory of sequences. Their signatures share many symbols with the original signature, along with some additions and deductions.

The theories of sequences of cvc5 and Z3 share the following symbols:

- $\mathsf{seq.empty}$: the empty sequence
- $\mathsf{seq.unit}(v)$: a sequence of length 1 containing only the value v
- $\mathsf{seq.len}(s)$: the length of the sequence s, denoted as ℓ_s
- $\mathsf{seq.nth}(s, i)$: the value associated with the i-th index of s if i is within the bounds of s, otherwise, an uninterpreted value
- $\mathsf{seq.extract}(s, i, l)$: the extracted maximal subsequence of s, starting at i of length l if i is within the bounds of s and l is positive, otherwise, the empty sequence
- $\mathsf{seq.}{+}{+}(s_1, ..., s_n)$: the concatenation of the sequences s_1, ..., and s_n

[1] cvc5's sequence theory:
https://cvc5.github.io/docs-ci/docs-main/theories/sequences.html

[2] Z3's sequence theory:
https://microsoft.github.io/z3guide/docs/theories/Sequences

- $\mathsf{seq.at}(s, i)$: a unit sequence containing the i-th value in s if i is within the bounds of s, otherwise, the empty sequence
- $\mathsf{seq.contains}(s_1, s_2)$: true if s_1 is a subsequence of s_2, false otherwise
- $\mathsf{seq.indexof}(s, s', i)$: the first position of s' in s at or after i, -1 if there are no occurrences
- $\mathsf{seq.replace}(s, s_1, s_2)$: the resulting sequence from replacing the first occurrence of s_1 with s_2 in s if s_1 occurs in s, otherwise, s
- $\mathsf{seq.prefixof}(s', s)$: true if s' is a prefix of s, false otherwise
- $\mathsf{seq.suffixof}(s', s)$: true if s' is a suffix of s, false otherwise

The theory of sequences of cvc5 also supports the following symbols:

- $\mathsf{seq.replace\_all}(s, s_1, s_2)$: the resulting sequence from replacing all occurrences of s_1 with s_2 in s, s if s_1 does not occur in s
- $\mathsf{seq.rev}(s)$: the resulting sequence from reversing s
- $\mathsf{seq.update}(s_1, i, s_2)$: a new sequence of the same size as s_1, in which, if i is within the bounds of s_1, then the values from i to $i + \mathsf{seq.len}(s_2) - 1$ are the same values as in s_2, and the other values are the same as in s_1, otherwise, it equals s_1

The theory of sequences of Z3 also supports symbols to map and fold over sequences. The first-order functions used in map and fold are expressed as arrays since higher-order functions were not yet supported in SMT-LIB [14] when these symbols were added. For example a function f of sort $A \to B \to C$ is actually an array of sort $\mathsf{Array}(A, \mathsf{Array}(B, C))$ and $f(a, b)$ where a is a value of sort A and b is a value of sort B is actually $\mathsf{select}(\mathsf{select}(f, a), b)$.

The symbols to map and fold over sequences in Z3 are the following:

- $\mathsf{seq.map}(f, s)$: the sequence of sort $\mathsf{Seq(E')}$ resulting from applying f, which is of sort $\mathsf{E} \to \mathsf{E'}$ with E as the sort of the elements of s, to all the elements of s.
- $\mathsf{seq.mapi}(f, o, s)$: the sequence of sort $\mathsf{Seq(E')}$ resulting from applying f, which is of sort $\mathsf{Int} \to \mathsf{E} \to \mathsf{E'}$ with E as the sort of the elements of s, to all the elements of s and their indices starting from the offset o.
- $\mathsf{seq.fold\_left}(f, b, s)$: the result of folding over s of sort $\mathsf{Seq(E)}$, with an initial value b of sort $\mathsf{E'}$ using the function f of sort $\mathsf{E'} \to \mathsf{E} \to \mathsf{E'}$.

  For example, given $s = \mathsf{seq.{+}{+}}(\mathsf{seq.unit}(v_0), \mathsf{seq.unit}(v_1), \mathsf{seq.unit}(v_2))$:

  $$\mathsf{seq.fold\_left}(f, b, s) = f(f(f(b, v_0), v_1), v_2)$$

- $\mathsf{seq.fold\_lefti}(f, o, b, s)$: the result of folding over the values of s of sort $\mathsf{Seq(E)}$ and their indices, with an initial value b of sort $\mathsf{E'}$ using the function f of sort $\mathsf{Int} \to \mathsf{E'} \to \mathsf{E} \to \mathsf{E'}$, starting from the offset o.

  For example, given $s = \mathsf{seq.{+}{+}}(\mathsf{seq.unit}(v_0), \mathsf{seq.unit}(v_1), \mathsf{seq.unit}(v_2))$:

  $$\mathsf{seq.fold\_lefti}(f, o, b, s) = f(o + 2, f(o + 1, f(o, b, v_0), v_1), v_2)$$

The seq.update function differs in the paper that describes the reasoning implemented in cvc5 [113] from the one that is implemented in cvc5 and described in its documentation[3]. In the paper, it is described as a function that sets only the value of one index and takes that value as a third argument, while in the documentation and implementation, it takes a sequence as a third argument.

In the rest of the document, seq. will be omitted in formulas containing sequence terms for simplicity.

In Wang and Appel's theory of arrays with concatenation [127], the theory of arrays is extended with length, slice, and concatenation functions, providing arrays with properties similar to those of sequences, mainly 0-indexing and length. The theory is composed of the following symbols:

- $\mathsf{length}_S(s)$: the length of s
- $\mathsf{nth}_S(i, s)$: if $0 \geq i < \mathsf{length}_S(s)$ returns the i-th value of s, otherwise, then the value is the default value of the sort of values (the theory assumes that every value sort S has a variable d_S corresponding to the default value of that sort)
- $\mathsf{repeat}_S(v, n)$: a sequence of size n if n is positive, in which all values are v, the empty sequence if n is negative
- $\mathsf{app}_S(s_1, s_2)$: concatenates s_1 and s_2
- $\mathsf{slice}_S(i, j, s)$: a subsequence of s from $\max(i, 0)$ to $\min(j, l)$, with l as the length of s, the empty sequence if such a subsequence does not exist
- $\mathsf{map}_f(s_1, ..., s_k)$: the sequence resulting from applying f element-wise to the n first elements of the sequences s_1, ..., s_k, where $n = \min(\mathsf{length}_S(s_1), ..., \mathsf{length}_S(s_k))$
- $\mathsf{update}_S(i, s, x)$: returns an updated version of s in which i is mapped to x if i is within the bounds of s. It is mentioned that the function update_S is reduced to a concatenation of the sequences $\mathsf{slice}_S(0, i, s)$, $\mathsf{repeat}_S(v, 1)$, and $\mathsf{slice}_S(i+1, \mathsf{length}_S(s), s)$ when i is within the bounds of s.

The map_f symbol is similar to the map function over arrays described in a paper by de Moura and Bjørner presenting the CAL (Combinatory Array Logic) array decision procedure [94], which produces new arrays in which each index i stores the value resulting from the mapped function applied over the i-th elements of the n arrays on which the map function was applied.

There are also works on the theory of arrays that extend the theory with a length function [29] and other functions that operate over regions of arrays, such as the C programming language memset and memcpy functions, as well as lambda terms [52]. These extensions provide the theory of arrays with more expressiveness and properties that are also desired in theories of sequences.

[3] cvc5's sequence theory: https://cvc5.github.io/docs-ci/docs-main/theories/sequences.html

$$\text{R-Nth}\ \frac{x = \mathsf{nth}(y, i)}{\begin{array}{rl} i < 0 \vee i \geq \ell_y & || \\ 0 \leq i < \ell_y \wedge \ell_k = i \wedge y = k{+\!\!+}\mathsf{unit}(x){+\!\!+}k' & \end{array}}$$

$$\text{R-Update}\ \frac{x = \mathsf{update}(y, i, z)}{\begin{array}{rl} i < 0 \vee i \geq \ell_y \wedge x = y & || \\ 0 \leq i < \ell_y \wedge \ell_k = i \wedge \ell_{k'} = 1 \wedge & \\ y = k{+\!\!+}k'{+\!\!+}k'' \wedge x = k{+\!\!+}\mathsf{unit}(z){+\!\!+}k'' & \end{array}}$$

$$\text{R-Split}\ \frac{x = w{+\!\!+}y{+\!\!+}z \quad x = w{+\!\!+}y'{+\!\!+}z'}{\begin{array}{rl} \ell_y > \ell_{y'} \wedge y = y'{+\!\!+}k & || \\ \ell_y < \ell_{y'} \wedge y' = y{+\!\!+}k & || \\ \ell_y = \ell_{y'} \wedge y' = y & \end{array}}$$

Figure 2.17: The reduction rules for the nth and update functions and the R-Split rule used for normalization.

2.6.2 Reasoning approaches

To reason over sequences, two main approaches can be identified. The first is based on string reasoning [19, 81] using word equations [58], which was generalized to support sequences, since sequences can be seen as a generalization of strings to non-character elements. This approach is referred to as the BASE calculus in [113]. The second, referred to as the EXT calculus, is an extension of BASE with dedicated array reasoning for the seq.nth and seq.update sequence functions, which respectively serve the role of the select and store functions in the array theory. The array reasoning in EXT is based on weak-equivalence array reasoning [37].

In [127], the decision procedure is also based on array reasoning since the proposed theory is an extension of the theory of arrays. However, the array reasoning in it is distinct from the one in [37], as it is an extension of the array property fragment decision procedure [34].

This section presents an overview of these reasoning approaches, focusing on their specificities and the implications they have on the efficiency of the reasoning.

BASE calculus

This calculus is based on the one described in [81] for strings. The general idea behind the calculus is to reduce the sequence operations into concatenation operations, effectively reducing the problem to a word equation problem with length constraints. The calculus also uses splitting rules that split sequences appearing in concatenations into smaller subsequences as a way to create normal forms for sequences, making it possible to compare them based on their contents.

Figure 2.17 illustrates how the nth and update functions are reduced to concatenations of sequences. In it k, k' and k'' represent fresh sequence variables. The R-Nth rule states that given $x = \mathsf{nth}(y, i)$, either the index i is not within the bounds of y, in which case the value x is uninterpreted, or y is propagated as equal to a concatenation of fresh sequence

variables with $\mathsf{unit}(x)$ as the i-th element of the sequence. R-Update works similarly: if i in $x = \mathsf{update}(y, i, z)$ is not within the bounds of y, then $x = y$, otherwise, x is propagated as equal to the concatenation of k, $\mathsf{unit}(z)$, and k'', with k of length i, and y is propagated as equal to the concatenation of k, k', and k'', where k, k', and k'' are fresh sequence variables. The R-Split rule normalizes concatenations of sequences, when a sequence is equal to two concatenations with a common prefix w and differ in the rest of the sequence terms, either the two next sequence terms after the prefix in each sequence have the same length, in which case they need to be propagated as equal, or one needs to be split over the other by propagating, for example, that $y = y' +\!\!+ k$ when $\ell_y > \ell'_y$, with k being a fresh sequence variable.

EXT calculus

The EXT calculus extends the BASE calculus by adding dedicated array reasoning for the nth and update functions. Therefore, instead of reducing them to concatenations of sequences, they are handled natively with array reasoning.

The rules of the EXT calculus are shown in figure 2.18, these rules replace the R-Nth and R-Update rules in figure 2.17. The rules use $z_1, \ldots, z_n$ to denote fresh sequence variables and e, e' to denote fresh element variables.

In EXT, reasoning over nth and update is not done through reduction to concatenation as in BASE, instead, the reasoning is inspired by array reasoning and works as follows. The Nth-Concat rule is applied when the nth function is called on a sequence that is a concatenation of other sequences and consists of selecting the appropriate subsequence on which to apply the nth function by choosing the one whose bounds contain the index. Update-Concat and Update-Concat-Inv similarly apply update applications when the resulting sequence, or the sequence on which update is applied, is a concatenation. The Nth-Unit and Update-Unit rules respectively select and store one element at one index in a unit sequence when the index is within the bounds of the sequence, which in this case means that it is equal to 0.

Arrays with concatenation calculus

Appel and Wang introduced an extension of the theory of arrays with length, slice, and concatenation functions [127]. They also defined the array property fragment with concatenation which, contrary to the array property fragment without concatenation [34], is not undecidable in general. The main addition of the array property fragment with concatenation to the one without concatenation is allowing index terms of the form $i + n$ and comparisons between $i + n$ and $j + m$ in index guards, where i and j are quantified variables and n and m are constants. This addition is necessary to reason over concatenation, but it enables index shifting. This phenomenon occurs when two accesses to an array occur in a quantified formula at the indices i and $i+n$, where i is a universally quantified variable. A formula is said to be tangle-free when no index shifting occurs in it. The authors proved that a tangle-free fragment of the array property fragment with concatenation is decidable.

The authors also introduced the index propagation graph (IPG), which is a graph that is used to detect entanglement. The vertices of the IPG are array variables and quantified variables, and each edge is labeled with an integer term representing the weight of the

$$\textsf{Nth-Concat}\ \frac{x = \textsf{nth}(y, i) \qquad y = w_1 +\!\!+ \ldots +\!\!+ w_n}{\begin{array}{ll} i < 0 \vee i \geq \ell_y & || \\ 0 \leq i < \ell_{w_1} \wedge x = \textsf{nth}(w_1, i) & || \ldots || \\ \sum_{j=1}^{n-1} \ell_{w_j} \leq i < \sum_{j=1}^{n} \ell_{w_j} \wedge x = \textsf{nth}(w_n, i - \sum_{j=1}^{n-1} \ell_{w_j}) & \end{array}}$$

$$\textsf{Update-Concat}\ \frac{x = \textsf{update}(y, i, v) \qquad y = w_1 +\!\!+ \ldots +\!\!+ w_n}{\begin{array}{c} x = z_1 +\!\!+ \ldots +\!\!+ z_n \wedge \\ z_1 = \textsf{update}(w_1, i, v) \wedge \cdots \wedge \\ z_n = \textsf{update}(w_n, i - \sum_{j=1}^{n-1} \ell_{w_j}, v) \end{array}}$$

$$\textsf{Update-Concat-Inv}\ \frac{x = \textsf{update}(y, i, v) \qquad x = w_1 +\!\!+ \ldots +\!\!+ w_n}{\begin{array}{c} y = z_1 +\!\!+ \ldots +\!\!+ z_n \wedge \\ w_1 = \textsf{update}(z_1, i, v) \wedge \cdots \wedge \\ w_n = \textsf{update}(z_n, i - \sum_{j=1}^{n-1} \ell_{w_j}, v) \end{array}}$$

$$\textsf{Nth-Unit}\ \frac{x = \textsf{nth}(y, i) \qquad y = \textsf{unit}(u)}{i < 0 \vee i > 0 \quad || \quad i = 0 \wedge x = u}$$

$$\textsf{Update-Unit}\ \frac{x = \textsf{update}(y, i, v) \qquad y = \textsf{unit}(u)}{(i < 0 \vee i > 0) \wedge x = \textsf{unit}(u) \quad || \quad i = 0 \wedge x = \textsf{unit}(v)}$$

$$\textsf{Nth-Update}\ \frac{\textsf{nth}(x, j) \qquad y = \textsf{update}(z, i, v) \qquad x = y \text{ or } x = z}{\begin{array}{ll} j < 0 \vee j \geq \ell_x & || \\ i = j \wedge 0 \leq j < \ell_z \wedge \quad \textsf{nth}(y, j) = v & || \\ i \neq j \wedge 0 \leq j < \ell_z \wedge \quad \textsf{nth}(y, j) = \textsf{nth}(z, j) & \end{array}}$$

Figure 2.18: The inference rules of the EXT calculus.

$$\text{length-app}\ \frac{\Psi(\mathsf{length}_S(\mathsf{app}_S(a,b)))}{\Psi(\mathsf{length}_S(a)+\mathsf{length}_S(b))}$$

$$\text{nth-app}\ \frac{\Psi(\mathsf{nth}_S(\mathsf{app}_S(a,b),i))}{\begin{array}{rl} 0 \leq i < \mathsf{length}_S(a) \implies \Psi(\mathsf{nth}_S(a,i)) & || \\ \mathsf{length}_S(a) \leq i < \mathsf{length}_S(a)+\mathsf{length}_S(b) \implies \Psi(\mathsf{nth}_S(b,i)) & || \\ \neg(0 \leq i < \mathsf{length}_S(a)+\mathsf{length}_S(b)) \implies \Psi(d_S) & \end{array}}$$

Figure 2.19: Reduction rules for ground array terms involving the app_S function. Terms at the top of the rule are replaced by terms at the bottom of the rule in a formula Ψ.

edge. The graph is initialized by adding, for each term of the form $\mathsf{nth}_S(i+n,a)$, an edge from the array a to the index variable i with weight n and an edge from i to a with weight a. For each comparison of the form $i+n \leq j+m$, an edge is added from the index variable i to the index variable j with weight $n-m$ and from j to i with weight $m-n$. It is proven that if the index propagation graph contains a cycle whose weight is not zero, then the formula is not tangle-free, making it undecidable.

The IPG is also used to define the index set I that is used for the instantiation of quantified index variables. A function δ that takes a vertex as an argument and returns a weight is defined by arbitrarily choosing a reference vertex u_c from each connected component in the graph and considering that $\delta(u_c) = 0$. Then, δ is defined for the rest of the vertices as:

$$\delta(u) = w + \delta(v)$$

for each pair of vertices u and v such that there is an edge from u to v with the weight w.

The index set I is then constructed from an empty set for each connected component in the IPG as follows:

- For every array variable a, add $-1-\delta(a)$ to I.
- For every occurrence of $\mathsf{nth}_S(n,a)$, where n is a term that does not contain any universally quantified variables, add $n-\delta(a)$ to I.
- For every occurrence of $i+n \leq m$ or $m \leq i+n$ in index guards, where i is a universally quantified variable and n, m are terms without quantified variables, add $m-n-\delta(i)$ to I.

Similarly to the array property fragment decision procedure, the decision procedure for the array property fragment with concatenation works in steps. The first step consists of applying reduction rules to the theory's ground terms. Figure 2.19 illustrates the rules corresponding to the reduction of ground terms involving app_S applications.

$$\textsf{forall-nth-app}\ \frac{\Psi(\forall \bar{i}.\ G(\bar{i}) \implies F(\textsf{nth}_S(\textsf{app}_S(a,b),i)))}{\Psi\left(\begin{array}{ll} & \forall \bar{i}.\ G(\bar{i}) \wedge 0 \leq i < \textsf{length}_S(a) \\ & \quad \implies F(\textsf{nth}_S(a,i)) \\ \wedge & \forall \bar{i}.\ G(\bar{i}) \wedge \\ & \textsf{length}_S(a) \leq i < \textsf{length}_S(a) + \textsf{length}_S(b) \\ & \quad \implies F(\textsf{nth}_S(b, i - \textsf{length}_S(a))) \\ \wedge & \forall \bar{i}.\ G(\bar{i}) \wedge \\ & \neg(0 \leq i < \textsf{length}_S(a) + \textsf{length}_S(b)) \\ & \quad \implies F(d_S) \end{array}\right)}$$

Figure 2.20: Rewrite rule for quantified array property formulas involving the $\textsf{app}_S$ function. Terms at the top of the rule are replaced by terms at the bottom of the rule in a formula Ψ.

The second step consists of simplifying array terms under quantifiers one variable at a time. In the forall-nth-app rule in Figure 2.20, the variable i is one of the variables in the vector of quantified variables $\bar{i}$. The rule turns a quantified array property formula into a simplified conjunction of quantified array property formulas.

The problem is then classified as tangle-free or not to determine its decidability. If the problem is tangle-free, the third step is to instantiate the quantifiers using the index set I. This is done by replacing every quantification $\forall \bar{i}.\ F(\bar{i})$ with $\bigwedge_{\bar{e} \in I + \delta(\bar{i})}$, where:

- The notation $I + k$ means $\{j + k \mid j \in I\}$.
- The notation $\bar{e} \in I + \delta(\bar{i})$ means $\{\bar{e} \mid \forall t.\ e_t \in I + \delta(i_t)\}$, where each i_t is a variable in $\bar{i}$ and $\bar{e}$ is the vector of all instantiations e_t.

This effectively replaces universal quantifiers with conjunctions over finite sets.

2.7 Colibri2

Colibri2 is a solver for SMT problems that behaves as an SMT solver, but that is actually a CP solver. Its reasoning relies on techniques from CP solving, such as powerful propagations, (abstract) domains and scheduling.

Colibri2 is a reimplementation of the COLIBRI CP solver [84], which was developed and used since the early 2000s and ranked first on multiple floating-point arithmetic benchmarks in the SMT-COMP. COLIBRI is implemented in Eclipse Prolog [109] where the notions of constraints, variables, and domains, as well as scheduling are already defined. It supports the quantifier-free fragments of the bit-vector, floating-point arithmetic, and array theories. It also supports the theories of integers and reals by encoding them using bit-vectors and floating-point numbers, respectively.

Since Colibri2 is implemented in OCaml, it has its own simplified implementations of a scheduler, domains, and domain propagation engine. Therefore, in Colibri2, integers and reals are not encoded as bit-vectors and floating-point numbers, but implemented natively using, respectively, the Zarith [88] library, which is based on GMP [64], and the Calcium library [71]. COLIBRI also relies on GMP, but it opts for finite domains, as

is traditional in CP. Colibri2 also supports quantifiers, algebraic data types, arrays, and sequences, and allows implementing Shostak theories [114] using domains.

2.7.1 Architecture

The architecture of Colibri2 is composed of various inter-dependent components, including nodes, the union-find environment, daemons, and the scheduler. Since Colibri2 is based on CP solving and not SMT, its architecture is not built around a SAT solver, and it does not use one. Which is why it lacks clause learning, contrary to most SMT solvers. This makes the cost of decisions significant, which is why they are avoided when possible by using scheduling heuristics and powerful propagations. On the other hand, the lack of learning has allowed for a simpler architecture and made the implementation easier.

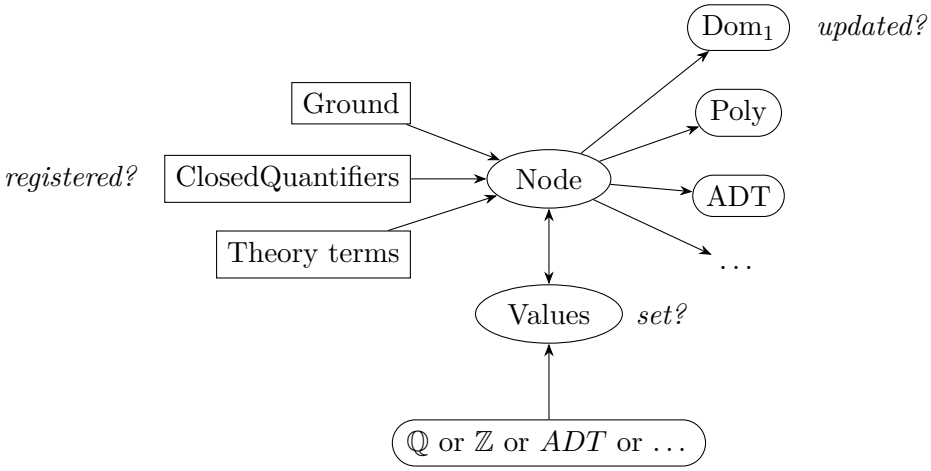

Figure 2.21: Relations between different concepts at the core of Colibri2.

Nodes and the union-find environment

In SMT solvers, congruence closure is often used to reason over equality and inequality between terms, and all the information on terms is regrouped in the congruence closure's e-graph [91]. Colibri2 takes a different approach that is similar to that of COLIBRI. In this approach, a union-find data structure is used to reason over equivalence between terms while congruence is implemented and used separately as a domain over terms that propagates new equivalences between them that are deduced by applying the congruence axiom (presented in Figure 2.10). This union-find is a separate generic entity, other information on terms, such as their domains, is stored separately and associated with the nodes.

Nodes represent semantic views of terms in equivalence classes. In each equivalence class, one node is the representative. A node does not correspond to a specific term, but to the semantic value which represents all terms that are equivalent to one another at a certain point. The nodes are maintained and updated as the union-find is updated. This approach allows for the independent and non-intrusive development of theories and domains outside the core engine and without interfering with existing theories, similar to CC(X) [40].

Figure 2.21 illustrates different concepts at the core of Colibri2. On the left side are shown different ways in which nodes can be created, on the right side examples of domains are provided, at the bottom are shown types of values that a node can have and in italic are shown events on which daemons can wait on.

As illustrated in Figure 2.21, nodes are created in different ways:

- By converting a ground term that comes from the original problem into a node.
- From a formula that is a closed quantifier, which is a quantified formula in which all variables are quantified.
- From theory terms, which are terms created by theories and do not necessarily occur in the original problem.

Figure 2.21 also shows that each node is associated with a set of domains. In the implementation, an environment env in the form of a record with two fields is used to handle the union-find used for equivalence and the association between nodes and their domains:

- $\mathsf{reprs} : \mathsf{Node.t} \to \mathsf{Node.t}$
  A mapping representing the equivalence classes of nodes, it associates each node n to the representative of its equivalence class r, which is by default n itself, i.e.:

$$\mathsf{reprs}[n] = \begin{cases} r & \text{if } n \in Dom(\mathsf{reprs}) \text{ and } \mathsf{reprs}[n] = r \\ n & \text{otherwise} \end{cases}$$

- $\mathsf{doms} : \mathsf{Node.t} \to \mathcal{D} \to \mathcal{R}$
  A mapping of each representative node r to a mapping of each domain identifier $\mathcal{D}$ to the domain's representation for the node r if it exists. The mapping returns an option for convenience and it is defined as follows:

$$\mathsf{doms}[n][\mathcal{D}] = \begin{cases} \mathsf{None} & \text{if } r \notin Dom(\mathsf{doms}) \\ \mathsf{None} & \text{if } \mathcal{D} \notin Dom(\mathsf{doms}[r]) \\ \mathsf{Some}(d) & \text{otherwise, with } d = \mathsf{doms}[r][\mathcal{D}] \end{cases}$$

  with r as the representative of the node n.

Values are also associated to the nodes, the values are elements of a universe A which represents all the possible values that any node can have. The value of a node also belongs to its sort domain.

Values can be set for nodes in three different ways:

- From ground terms: when a node is generated from a ground term that is a literal value itself.
- Through propagation: in some cases, propagations can restrict the domains of a node n until it is precise enough to determine a single value v to which the node can be equal. When that value is obtained, it is set for n.

- From model generation: when a value v is generated for a node n, another node n_v is created for the value v, which is then merged with the node n.

A value can also be set to a node when it does not have one and is merged with a node that has one. A contradiction is raised, when two nodes are merged with distinct values.

When a model is generated for a given problem, the model is in the form of a mapping of sort $\mathsf{Node.t} \to A$ where each node is associated to a value.

Domains

Domains in Colibri2 can be concrete or sort domains, which represent the possible values that a node can have, or abstract domains, which represent other useful properties.

Different kinds of nodes are associated with different domains. For example, arithmetic nodes are associated with a domain of interval unions, which states the possible values that the node can have, as well as a polynomial domain that holds their normalized polynomial representation. ADT nodes, on the other hand, are not associated with these two domains, but instead have a domain that represents their possible values, such as the set of possible constructors if the ADT is a sum type, for example.

Definition 2.24 (Domains in Colibri2)**.** *A domain $\mathcal{D}$ in Colibri2 can be formalized as a tuple $\langle \mathcal{R}, P_{\mathcal{A}}, \text{equal} \rangle$ where:*

- $\mathcal{R}$*: the set of domain representations.*
- $P_{\mathcal{A}} : \mathcal{R} \to 2^A$
  Given a model $\mathcal{A}$ and an element of the domain $d \in \mathcal{R}$, $P_{\mathcal{A}}(d)$ denotes the set of values in the universe A represented by d under $\mathcal{A}$.
- $\text{equal} : \mathcal{R} \to \mathcal{R} \to \mathit{Bool}$
  Checks whether two domain representations have the same semantic interpretation with a given model. It is defined for any $d_1, d_2 \in \mathcal{R}$ as:

$$\text{equal}(d_1, d_2) \implies \forall \mathcal{A}.\ P_{\mathcal{A}}(d_1) = P_{\mathcal{A}}(d_2)$$

  For convenience, it will be denoted as an equality "$=$" between domain representations.

In the formalization of domains in Colibri2 presented in Definition 2.24, $P_{\mathcal{A}}$ depends on the model $\mathcal{A}$ in order to support relational domains, which are domains whose elements can contain other nodes. An example of such a domain is the polynomial domain, where the polynomial representation r of a node can contain other nodes, therefore, to compute $P_{\mathcal{A}}(r)$, a model $\mathcal{A}$ is needed to retrieve the values of the nodes that may appear in r.

Colibri2 provides two ways to define domains in the implementation $\mathcal{D}_{\text{merge}}$ and $\mathcal{D}_{\text{inter}}$.

Definition 2.25 (The domain definition $\mathcal{D}_{\text{merge}}$)**.** *A domain $\mathcal{D}$ in Colibri2 can be defined as the tuple $\langle \mathcal{R}, P_{\mathcal{A}}, \text{equal}, \text{merge}_{\mathcal{D}} \rangle$, where $\mathcal{R}$, $P_{\mathcal{A}}$, and* equal *are the same as in Definition 2.24, and* merge *is:*

- $\text{merge}_{\mathcal{D}} : \textit{env} \to (n_1 : \textit{Node.t}) \to (n_2 : \textit{Node.t}) \to \textit{env}$
  Ensures that the nodes n_1 and n_2 have the same domain $\mathcal{D}$ in env.

When a domain is defined using $\mathcal{D}_{\text{merge}}$, it can be interacted with in the implementation through two functions:

- $\mathsf{get\_dom}_{\mathcal{D}} : \mathsf{env} \to (n : \mathsf{Node.t}) \to \mathcal{R}\ \mathsf{option}$
  Returns $\mathsf{env.doms}[n][\mathcal{D}]$, i.e. the domain $\mathcal{D}$ of the node n if it is set.

- $\mathsf{set\_dom}_{\mathcal{D}} : \mathsf{env} \to (n : \mathsf{Node.t}) \to (d : \mathcal{R}) \to \mathsf{env}$
  Sets the domain $\mathcal{D}$ of the node n with the domain representation d. Ensures that: $\mathsf{env.doms}[n][\mathcal{D}] = d$

Definition 2.26 (The domain definition $\mathcal{D}_{\text{inter}}$)**.** *A domain in Colibri2 can also be defined as the tuple $\langle \mathcal{R}, P_{\mathcal{A}}, \text{equal}, \text{inter}_{\mathcal{D}} \rangle$, where $\mathcal{R}$, $P_{\mathcal{A}}$, and* equal *are the same as in Definition 2.24, and* $\text{inter}_{\mathcal{D}}$ *is:*

- $\text{inter}_{\mathcal{D}} : \mathcal{R} \to \mathcal{R} \to \mathcal{R}$ *option*
  Computes the intersection between two domain representations and returns it if it is not empty, and returns None *otherwise.*

  For any $d_1, d_2 \in \mathcal{R}$, if $\text{inter}(d_1, d_2)$ *succeeds and the result of the intersection is $d \in \mathcal{R}$, then the function satisfies:*

  $$\text{inter}(d_1, d_2) = \textit{Some}(d) \implies \forall \mathcal{A}.\ \forall v \in A.\ v \in P_{\mathcal{A}}(d) \implies v \in P_{\mathcal{A}}(d_1) \wedge v \in P_{\mathcal{A}}(d_2)$$

  Otherwise, if $\text{inter}(d_1, d_2)$ *fails, then it satisfies:*

  $$\text{inter}(d_1, d_2) = \textit{None} \implies \forall \mathcal{A}.\ \forall v \in A.\ \neg\Big(v \in P_{\mathcal{A}}(d_1) \wedge v \in P_{\mathcal{A}}(d_2)\Big)$$

Definition 2.26 presents an alternative definition of domains in Colibri2. With this definition, in addition to get_dom and set_dom, it is possible to update the domain of a node with:

- $\mathsf{upd\_dom}_{\mathcal{D}} : \mathsf{env} \to (n : \mathsf{Node.t}) \to (d : \mathcal{R}) \to \mathsf{env}$: Updates the domain $\mathcal{D}$ of the node n. If n already had a domain d', it ensures that its new domain is $\text{inter}_{\mathcal{D}}(d, d')$, otherwise, sets n's domain to d.

In practice, when the inter function in $\mathcal{D}_{\text{inter}}$ returns None, a contradiction is raised as that implies that the two domain representations do not intersect, i.e.:

$$\forall \mathcal{A}.\ P_{\mathcal{A}}(\text{inter}(d_1, d_2)) = \emptyset$$

which means that no model can satisfy the resulting domain from the intersection.

In the implementation, when a domain is defined using Definition 2.26, a merge function is defined for it using the inter function as illustrated in Listing 2.6.

```
let merge env n1 n2 =
  match get_dom env n1, get_dom env n2 with
  | None, None → env
  | Some d1, None → set_dom env n2 d1
  | None, Some d2 → set_dom env n1 d2
  | Some d1, Some d2 →
    match inter d1 d2 with
    | None → raise Contradiction
    | Some d →
        let env = set_dom env n1 d in
        set_dom env n2 d
```

Listing 2.6: Definition of the merge function using the inter function.

The merge functions, the one that is part of $\mathcal{D}_{\text{merge}}$ and the one defined using inter for $\mathcal{D}_{\text{inter}}$, are used when merging two nodes. When the merger of two nodes is requested, before updating their representatives in env.reprs, their domains are first conciliated using the domain's respective merge functions, ensuring that they have the same domain representations for all domains before merging. A conflict is raised otherwise.

In fact, in Colibri2 values are represented as a domain $\mathcal{D}_v$ which associates to each node n a given unique value v to ensure that if two nodes n and n' are merged with different values v and v' then that leads to a contradiction during the merger. Values are also represented with ground terms, so each value v has an associated unique ground term g_v, ensuring that nodes that are merged with g_v have the value v. Colibri2 also ensures that when the value of a node n is set to v, the node is merged the ground term g_v, ensuring that all nodes that have the same values are merged. This last propagation is done through daemons.

Daemons and events

Daemons are propagations that are waiting for the triggering of some event before being applied. Figure 2.21 shows the most used kinds of events:

- A new theory term is registered: used to initialize the domains of the node produced from the term and to add useful propagations associated with it.
- The domain of a node changed: used to propagate the changes to other nodes that depend on it, or whose domains depend on it.
- A value of some kind appeared: mainly used to set the domains of its corresponding node to a singleton.

These events can be found in two flavors: they can either wait for such an event to occur on any node, or wait for it to occur on a specific node.

In order to help reason about and implement the propagations, when a propagation requests the merger of two nodes, the merger is not done immediately but scheduled to be done after the end of the execution of the propagation.

Daemons can either be repetitive, meaning that they indefinitely wait on their event and run whenever it occurs, as many times as it occurs, or they run only once when the

event occurs for the first time. They can also be parametrized to run directly after their creation if the event on which they are waiting occurred in the past, otherwise, they wait for it to occur again. They can also be made to run only after n occurrences of the event, instead of running whenever it occurs.

Scheduler

Since multiple daemons can wait on the same event, it is therefore necessary to order their execution. Naively using a stack of daemons to run can lead to starvation, since two daemons can trigger each other's events indefinitely, while a daemon that could deduce an inconsistency might be stuck inside the stack.

It is also necessary to differentiate daemons by their cost. It is often preferable to delay costly daemons until some cheaper ones have run, since that might help the costlier ones. An example of a costly daemon is the simplex algorithm, which can be aided by propagations on the interval domains of the nodes involved in the arithmetic problem it solves. Another costly daemon is the quantifier instantiation daemon, which can lead to starvation by creating too many new terms. This phenomenon is also called a matching loop [78].

To remedy these issues, Colibri2 uses a scheduler based on the efficient time wheel data structure (similar to §6.2 in [124]), which makes it possible to define cycles of time ticks on which to run daemons. Cheap daemons, for example, can be scheduled to run in one tick, while costlier ones can run every 64 ticks.

The scheduler also manages the execution phases of Colibri2, of which there are four:

1. Propagation phase:
   - Registering of theory terms
   - Initialization of domains
   - Constraint propagations
2. Decision phase: Making decisions, after every decision, go back to the propagation phase
3. Last-effort phase: Costly propagations, can go back to the decision and propagation phases
4. Fix-model phase: Model generation

Colibri2 starts with the propagation phase, in which it registers the terms of the problem and propagates the constraints that come from them. Once all propagations are done, the decision phase starts, during which the solver makes decisions that were created during the propagation phase. Decisions are created when some propagation needs additional information to know if and how to run, and it therefore needs to decide on that information before running the different possible scenarios.

In some cases, propagations need additional information to know how to be applied, so decisions are registered to split cases and do the right propagations in each case.

After each decision, new propagations can be applied, so the propagation phase is restarted. New decisions will be made only when the solver either finds a conflict and backtracks, or when all propagations are complete.

Once all initial propagations and decisions are made, the last-effort phase begins, during which costly propagations are made, with a notable one of them being quantifier instantiation. The next last-effort phase will only start when all registered decisions and scheduled propagations from the previous one are done.

When a last-effort phase leads to a conflict, the solver backtracks, and takes a new branch of the last decision. If all the branches of the last decision were explored, a new decision is made. The solver then restarts the propagation phase. The propagations of the last-effort phase can be reclassified to belong to the propagation phase if they lead to a conflict. The reason for this reclassification is that if a last-effort propagation leads to a conflict, then it would be better to perform it earlier, during propagation, in hopes of detecting the conflict sooner.

If a last-effort phase ends and no decisions or propagations were scheduled by it, then the fix-model phase starts. Contrary to how it's usually done in SMT solvers, Colibri2 does not use a sophisticated theory combination framework that guarantees the existence of a model when no conflict is found. It is therefore necessary, to guarantee soundness, for Colibri2, after generating a model, to check that the original problem is indeed satisfied by it before answering that the problem is satisfiable. During the fix-model phase, Colibri2 will choose values for all the nodes for which a value has not been found through constraint propagation. The values are chosen by taking into consideration the domains of the nodes. Colibri2 will iterate over the choices of values until a model is found.

After a certain number of iterations, by default 1000, if none of the generated models are satisfying, the scheduler closes the branch and acts as if a conflict was detected and backtracks to make a different decision. If, after exploring all branches, no satisfying model is found, then Colibri2 answers with "unknown-branch-cut", which means that no conflict was detected, but the solver did not manage to produce a model. This result usually indicates that some propagations are not powerful enough.

2.7.2 Theory implementations

As previously mentioned, theories in Colibri2 are not combined using a theory combination framework that constrains how they interact with the rest of the system and with each other, for example, by ensuring that they handle only their purified terms. In Colibri2, every theory can see every term and every other theory. Theories are defined generically by simply registering a converter function, which can receive all terms, and each theory can choose which terms it is interested in. Terms are therefore not purified in Colibri2, and interactions between theories are easier. In fact, theories have access to the domains of other theories and can register daemons on terms from other theories as well.

Another notable difference with most SMT solvers is in boolean reasoning. In SMT solvers, it is often a special kind of reasoning, as it is done through the SAT solver and not seen as a theory. In Colibri2, booleans are handled by a theory of booleans, which is like any other theory. That is also the case in the CDSAT theory combination [30]

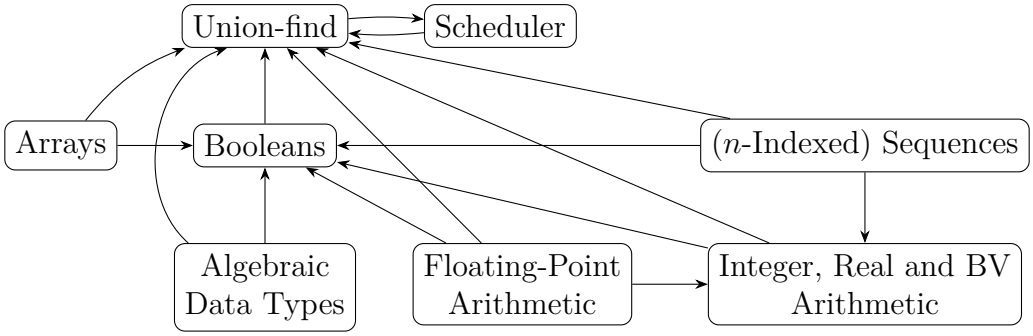

Figure 2.22: The interactions between the theories, the scheduler, the daemons, and the union-find in Colibri2.

framework, where the theory of booleans is like any other theory.

Figure 2.22 illustrates the interactions between the theories in Colibri2. It shows that the boolean theory does not use any other theory, it only depends on the union-find module which handles equivalence between terms. All the other theories depend on the theory of booleans and the union-find. They use the boolean theory to create decisions on boolean terms, notably equality and difference terms. The scheduler and union-find also interact with one another. When an event occurs, the daemons waiting on that event initially have read-only access to the union-find. That access allows them to inspect it and decide whether they still need to run or not. If the daemons need to run, they are scheduled to run and when they run they can bring changes, or write, on the union-find as they have read-write access to it at that point. The union-find module notifies the scheduler of the changes that happen to it. Figure 2.22 also shows that the integer, real, and bit-vector arithmetic theories are joined together, since they share most of the domains they use, notably the interval domain.

In addition to the access policy that Colibri2 has when it comes to interactions with the union-find, Colibri2 also relies on backtrackable data structures [7] to notably ensure that domains and other properties that theory modules might define for nodes are properly backtracked when the scheduler backtracks.

Chapter 3

Arithmetic I: Domains, Propagators and Relations

In programming languages, working with integers, rationals, and reals has historically been done using fixed-size bit-vectors and floating-point numbers. But arbitrary-sized integers, rationals, and reals are present in many programming languages, thanks to libraries such as GMP [64] and Calcium [71], which allow using numbers whose sizes grow as needed to represent their values. They are also often used in specification over programs manipulating fixed-size numbers, to reason over overflows for example as they are more convenient.

It is therefore necessary to be able to manipulate and reason over them in a software verification context. That is the purpose of the SMT theories of integer and real linear and non-linear arithmetic. While it is a well-studied topic [1, 49], reasoning approaches still tend to be costly and do not scale well, whether on the decidable theories (linear and non-linear real arithmetic, and linear integer arithmetic) or the undecidable theory of non-linear integer arithmetic.

In fact, some approaches even use arbitrary-sized integers and reals to reason over bit-vectors and floating-point numbers, respectively. This is done through the modulo operator $\mod$ for bit-vectors and through conversion to reals for floating-point numbers.

This chapter discusses how integer and real arithmetic terms are reasoned about in the Colibri2 CP solver. It begins by presenting the various domains that are used and the kinds of propagations that are done in Section 3.1. It then presents the labeled union-find data structure, introduced in Lesbre, Lemerre, Ait-El-Hara, and Bobot [80], as well as how it is used in Colibri2 for the constant difference relation over arithmetic terms in Section 3.2.

3.1 Arithmetic reasoning in Colibri2

In Colibri2, integers and reals share the same domains and mainly the same reasoning approaches, relying heavily on domains and powerful propagations. In fact they can even be part of the same equivalence class, when a real is converted from an integer for example. The domains used by the integer and real arithmetic decision procedures include the domain of interval unions, the domain of polynomials, the product domain, and the congruence domain.

3.1.1 Arithmetic domains

These domains are designed to represent various properties essential for efficient reasoning over real and integer arithmetic. Four of them are presented below.

Interval Union Domain

An interval is a typed set of values that is defined by its bounds. Given two reals a and b such that $a \leq b$, an interval of the values from a to b can be closed:

- $[a; b] = \{i \in \mathbb{R} \mid a \leq i \leq b\}$

The interval can also have one open and one closed bound to include only a or b within its values:

- $[a; b[= \{i \in \mathbb{R} \mid a \leq i < b\}$
- $]a; b] = \{i \in \mathbb{R} \mid a < i \leq b\}$

It can also exclude both bounds by having two open bounds:

- $]a; b[= \{i \in \mathbb{R} \mid a < i < b\}$

An interval can also be left-unbounded or right-unbounded:

- $]-\inf; b] = \{i \in \mathbb{R} \mid i \leq b\}$
- $]-\inf; b[= \{i \in \mathbb{R} \mid i < b\}$
- $[a; +\inf[= \{i \in \mathbb{R} \mid a \leq i\}$
- $]a; +\inf[= \{i \in \mathbb{R} \mid a < i\}$

If it is unbounded on both sides, then it represents all real values:

- $]-\inf; +\inf[= \mathbb{R}$

Alternatively, an empty interval is equal to the empty set $\emptyset$[1].

When working with integers, an integer interval is denoted as $\mathbb{Z}\,[a; b]$, with a and b its integer bounds. The operations described above also apply on integer intervals, and can be adapted by changing the set of values in the interval definitions from $\mathbb{R}$ to $\mathbb{Z}$.

Another difference is that, with integer intervals, an open bound on the left (resp. on the right) can be replaced with a closed bound of the successor (resp. predecessor) of the bound, i.e.:

- $\mathbb{Z}\,]a; b] = \mathbb{Z}\,[a + 1; b]$
- $\mathbb{Z}\,[a; b[= \mathbb{Z}\,[a; b - 1]$

[1] In Colibri2, since intervals are used in the interval domain, empty intervals are forbidden as they represent a contradiction, indicating that no value of the associated term can satisfy the problem's constraints.

- $\mathbb{Z}\,]a;b[= \mathbb{Z}\,[a+1;b-1]$
- $\mathbb{Z}\,]-\inf;b[= \mathbb{Z}\,]-\inf;b-1]$
- $\mathbb{Z}\,]a;+\inf[= \mathbb{Z}\,[a+1;+\inf[$

Usual arithmetic operations on these intervals are supported, given two intervals A and B:

- $A \diamond B = \{x \diamond y \mid x \in A \wedge y \in B\}$, with $\diamond \in \{+,-,\cdot,/\}$

Which comes down to:

- $[a;b] + [c;d] = [a+c;b+d]$
- $[a;b] - [c;d] = [a-d;b-c]$
- $[a;b] \cdot [c;d] = [\min(a\cdot c, a\cdot d, b\cdot c, b\cdot d); \max(a\cdot c, a\cdot d, b\cdot c, b\cdot d)]$
- $[a;b] \,/\, [c;d] = [a;b] \cdot (1/\,[c;d])$, where:
  - $1/\,[c;d] = [1/d;1/c]$, when $0 \notin [c;d]$
  - $1/\,[c;0] =]-\inf;1/c]$
  - $1/\,[0;d] = [1/d;\inf[$
  - $1/\,[c;d] =]-\inf;\inf[$, when $0 \in [c;d]$

In addition to Euclidean division presented above, Colibri2 also supports truncated and floored division, as well as Euclidean, truncated, and floored modulo, square root, power, and other operations ...

The definitions of these operations can easily be expanded to intervals that have open bounds. If a is an open bound and b is a closed one, the resulting $a \diamond b$ is an open interval.

These operations also apply on constants, which are simply singleton intervals. For example subtracting and adding a constant c to an interval $[a;b]$ is the same as:

- $[a;b] + c = [a;b] + [c;c]$
- $[a;b] - c = [a;b] - [c;c]$

The union and intersection operations on intervals are also supported, given two intervals A and B:

- $A \cup B = \{x \mid x \in A \vee x \in B\}$
- $A \cap B = \{x \mid x \in A \wedge x \in B\}$

Over-approximation is often used when trying to compute an interval union over disjoint intervals:

- $[a;b] \cup [c;d] = [a;d]$, when $a \leq b < c \leq d$

This leads to a loss of precision that can be significant, especially if the difference between the two intervals is large or unions are chained over multiple intervals. Therefore, to obtain better precision when doing arithmetic reasoning, it is necessary to use unions of intervals.

Example 3.1. *Given an unconstrained term of sort real x, its initial interval domain is unset, it is therefore* $]-\inf;+\inf[$ *(also denoted as* $\top$*) by default. If the propagation* $x \neq 0$ *is made, then the new interval domain of* x *will be* $]-\inf;0[\cup]0;+\inf[$.

The same operations are defined on unions of intervals. Given the interval unions A and B:

- $A = \bigcup_{i=1}^{n} I_i$
- $B = \bigcup_{j=1}^{m} J_j$

where all intervals I_i (resp. J_j) are disjoint, for any binary operation $\diamond \in \{+, -, \cdot, /\}$, the operations over the interval unions are defined as:

- $A \diamond B = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} (I_i \diamond J_j)$

where $I_i \diamond J_j$ is computed using standard interval arithmetic.
The interval union domain $\mathcal{D}_I$ in Colibri2 is defined with $\mathcal{D}_{\text{inter}}$, where:

- $\mathcal{R}_I$: An interval union $\bigcup_{i=1}^{n} I_i$
- $P_{\mathcal{A}} : (r : \mathcal{R}_I) \to 2^A$
  The subset of values included within the interval union: $\{i \mid i \in r\}$
- equal : $(r_1 : \mathcal{R}_I) \to (r_2 : \mathcal{R}_I) \to \mathsf{Bool}$
  Defined as equality between the interval unions r_1 and r_2:

$$(\forall x \in r_1.\ x \in r_2) \wedge (\forall x \in r_2.\ x \in r_1)$$

- inter : $\mathcal{R}_I \to \mathcal{R}_I \to \mathcal{R}_I$ option
  Defined as:

```
let inter (r1 : R_I) (r2 : R_I): R_I option =
  let r = r1 ∩ r2 in
  if r = ∅ then None else Some r
```

This domain is used to do propagations. For example, to determine that two terms x and y are distinct if their interval domains are disjoint. It is also used to select values during model generation for arithmetic terms.

Domain of Polynomials

Polynomials are terms of the form $\sum_i c_i v_i + c$, where c is a constant arithmetic term, c_i are non-zero constants called the coefficients, and v_i are variable terms that are either a single variable or a product of variables. The $\sum_i c_i v_i$ part of the polynomial is referred to as the variable part, and c is the constant part.

Addition and subtraction are supported on polynomials. Given two polynomials p_1 and p_2, such that $p_1 = \sum_i c_{1,i} v_{1,i} + c_1$ and $p_2 = \sum_j c_{2,j} v_{2,j} + c_2$, a constant c_3, and an operation $\diamond \in \{+, -\}$:

- $p_1 \diamond p_2 = \sum_k (c_{1,k} \diamond c_{2,k}) v_k + \sum_{i'} c_{1,i'} v_{1,i'} \diamond \sum_{j'} c_{2,j'} v_{2,j'} + (c_1 \diamond c_2)$
  where the variable terms v_k are those that appear in both p_1 and p_2, $v_{1,i'}$ are those that appear in p_1 only, and $v_{2,j'}$ are those that appear in p_2 only.

- $p_1 \diamond c_3 = \sum_i c_{1,i} v_{1,i} + (c_1 \diamond c_3)$

Multiplication and division are also supported between polynomials and constants, given an operation $\star \in \{\cdot, /\}$:

- $p_1 \star c_3 = \sum_i (c_{1,i} \star c_3) v_{1,i} + (c_1 \star c_3)$

The domain of polynomials in Colibri2 associates to each arithmetic node a unique normalized polynomial representation. It is an example of a relational domain. A relational domain of a node is one which can depend on other nodes, i.e. elements of that domain can contain other nodes. The polynomials are represented as a pair (c, m_v), where c is the constant, and m_v is a map associating to each variable node in the polynomial v_i its non-zero coefficient c_i. If a variable term's coefficient is zero, then that variable is not added to the map m_v.

If a variable term v_i has its polynomial domain set to p_i, then v_i is replaced by p_i in the polynomials in which it occurs. This ensures that the variable terms in the polynomials are always as simple as possible, and normalizes the polynomials for all arithmetic terms, which makes performing operations on them easier.

The domain of polynomials $\mathcal{D}_{Poly}$ in Colibri2 is defined with $\mathcal{D}_{\text{merge}}$, where:

- $\mathcal{R}_P$: A polynomial $p = (c, m_v)$

- $P_{\mathcal{A}} : (r : \mathcal{R}_P) \to 2^A$
  The singleton set of the value to which the polynomial evaluates to with the model $\mathcal{A}$.

- equal : $(p_1 : \mathcal{R}_P) \to (p_2 : \mathcal{R}_P) \to \mathsf{Bool}$
  Defined as the equality between the polynomials p_1 and p_2. Since these are normalized, this is a syntactic equality.

- merge : env → Node.t → Node.t → env
  Defined as:

```
let merge env (n1 : Node.t) (n2 : Node.t): env =
  match get_dom_poly env n1, get_dom_poly env n2 with
  | None, None → env
```

```
| Some p1, None →
  set_dom_poly env n2 p1
| None, Some p2 →
  set_dom_poly env n1 p2
| Some p1, Some p2→
  let r = p1 − p2 in
  let (c, m) = r in
  if m = ∅ then
    if c = 0 then
      let env = set_dom_poly env n1 r in
      set_dom_poly env n2 r
    else
      raise Contradiction
  else
    let (ci, vi) = Map.max_binding m in
    let m′ = Map.remove vi m in
    propagate env vi ((m′ + c)/ci)
```

where the propagate function propagates that the variable term v_i is equal to the polynomial $((m' + c)/c_i)$ and substitutes its occurrences with it. After the substitution in the $\mathcal{D}_{Poly}$ domains p_1 and p_2 of the nodes n_1 and n_2, p_1 and p_2 become equal.

This domain is mainly used as a convenient way to handle arithmetic terms in their polynomial form and perform operations on them without creating the corresponding node for each polynomial produced while performing these operations. Notably, when computing intermediary results, the number of produced polynomials can be high, and creating a node for each one would be detrimental to performance, as each new node must be registered and assigned its own domains, constraints, and propagations. Since this is normally unnecessary, it is preferable to avoid creating too many useless terms, as that would slow down performance.

Product Domain

The product domain, also called the domain of monomials, associates to each non-constant arithmetic term the set of variable products to which it is equal. A product is in the form of a pair (c, m_p), where c is a constant and m_p is a map associating to each variable v_i a power p_i to which it is raised, and that is different from zero.

This domain used as a basis for two other domains: the sign domain and the absolute value domain, since these two properties can be deduced from the product.

For example, if $x = a \cdot b$, such that $a = 2 \cdot v_1^2$ and $b = 3 \cdot v_2^3$, then $x = 6 \cdot v_1^2 \cdot v_2^3$, and the resulting product domain of the term x is $(6, \{v_1 \mapsto 2;\ v_2 \mapsto 3\})$. And if the sign of v_2 is known to be positive, then the sign of x will also be known to be positive.

Similarly to the domain of polynomials, this domain is also normalized by substituting all occurrences of a term t in other product domains with the product domain of t, when it is set.

The product domain $\mathcal{D}_{Prod}$ in Colibri2 is defined with $\mathcal{D}_{\text{inter}}$, where:

- $\mathcal{R}_{prod}$: A set of products (c_i, m_i).

- $P_{\mathcal{A}} : (r : \mathcal{R}_{prod}) \to 2^A$
  The singleton value that is computed from product by substituting the variables by their interpretations. If there are multiple products, then either they all result in the same value, or it is a contradiction.

- equal : $(P_1 : \mathcal{R}_{prod}) \to (P_2 : \mathcal{R}_{prod}) \to$ Bool
  Checks whether the sets of products P_1 and P_2 are equal.

- inter : $(P_1 : \mathcal{R}_{prod}) \to (P_2 : \mathcal{R}_{prod}) \to \mathcal{R}_{prod}$ option
  Since non-linear reasoning in Colibri2 is limited, this inter function is particular: for each $p_1 \in P_1$ and $p_2 \in P_2$, the function solve (presented in Listing 3.1) is called. It can either refine the two products or join them if no solution is found.

The solve function takes two pairs as arguments: $(u_1, (c_1, p_1))$ and $(u_2, (c_2, p_2))$, where (c_1, p_1) and (c_2, p_2) are two products that were propagated as equal to one another, and u_1 and u_2 are the variables in p_1 and p_2 that are "unknown", meaning that it is not known whether their values can be 0 or not, while all other variables are known to be non-zero. The function returns:

- AlreadyEqual: if the products are equal.

- Contradiction: if solving the product equality leads to a conflict.

- Unsolved: if solve is unable to solve the product equality.

- Subst m: if the product equality is solvable and new equalities are deduced. These equalities are provided as a map m, which maps variables to the products to which they are equal.

The source code of the solve function is illustrated in Listing 3.1. It essentially does some straightforward simplifications, such as when one of the products is equal to 0 (lines 30 and 31), or when there are no unknown variables (lines 32 to 38).

Otherwise, non_zero is called with $(u_1, (c_1, p_1))$ and $(u_2, (c_2, p_2))$. If $u_1 = \emptyset$, it tries to find a new equality between a variable of power one and the division between the two products (lines 10 to 20). If u_1 contains only one element n that also appears in u_2, an equality is propagated between n and the rest of the products (lines 22 to 25). In all other cases, the product equality is not solved and both products are kept. If the call to non_zero with $(u_1, (c_1, p_1))$ and $(u_2, (c_2, p_2))$ returns Unsolved, then non_zero is called again with the arguments swapped.

Modular arithmetic congruence Domain

When reasoning over arithmetic, modular arithmetic [62, 63] plays an important role, as it allows for quick deductions and helps avoid slow, potentially infinite, convergence during propagations when the unsatisfiability of a problem can be easily deduced through modularity.

In the modular arithmetic congruence domain, an arithmetic term t is associated with a pair (a, b), where a and b are rationals that respectively represent the divisor and the

```
let equal_to_zero u (c,p) =
  if u = ∅ then
    if c = 0 then AlreadyEqual else Contradiction
  else if Map.cardinal u = 1 then
    Subst (Map.map (fun _ → 0) u)
  else Unsolved

let non_zero (u1,(c1,p1)) (u2,(c2,p2)) =
  if u1 = ∅ then
    try
      Map.iter (fun n q1 →
          let q2 = Map.find_def 0 n p2 in
          if q1 − q2 = 1 then
            let (c,p) = ((c2,p2)/(c1,p1)) in
            let p = Map.remove n p in
            raise (Solved (n,(c,p)))
        ) p1;
      Unsolved
    with Solved (n,(c,p)) →
      Subst (Map.singleton n (c,p))
  else if Map.cardinal u1 = 1 then
    let (n,q) = Map.choose u1 in
    if not (Map.mem n u2) then
      let (c,p) = ((c2,p2)/(c1, Map.remove n p1))^(1/q) in
      Subst (Map.singleton n (c,p))
    else Unsolved
  else Unsolved

let solve (u1,(c1,p1)) (u2,(c2,p2)) =
  if c1 = 0 ∧ p1 = ∅ then equal_to_zero u2
  else if c2 = 0 ∧ p2 = ∅ then equal_to_zero u1 (c1,p1)
  else if u1 = ∅ ∧ u2 = ∅ then
    match ((c1,p1)/(c2,p2)) with
    | One → AlreadyEqual
    | Cst _ → Contradiction
    | Var (q,n,p) →
        let p = p^(−1/q) in
        Subst (Map.singleton n p)
  else
  match non_zero (u1,(c1,p1)) (u2,(c2,p2)) with
  | Unsolved →
    non_zero (u2,(c2,p2)) (u1,(c1,p1))
  | r → r
```

Listing 3.1: Implementation of the solve function used in the product domain.

remainder. This means that the term t is of the form $a\mathbb{Z} + b$, where $\mathbb{Z}$ represents any integer.

Traditionally, modular arithmetic works on integers, i.e. a and b are usually integers. However, when reasoning over both integer and real/rational arithmetic, it is useful to extend modular reasoning to rationals, due to conversions between integers and rationals. Example 3.2 shows a case in which this extension is useful, and how rationals are introduced into the modular arithmetic domain.

Example 3.2. *Given an integer term t_i that has its modular arithmetic domain set to (a, b), and a real term t_r such that $t_r = 0.5 \cdot \mathit{of\_int}(t_i)$, where of_int is a function that takes an integer and converts it to a real. Then the modular arithmetic domain of t_r will be set to $(a/2, b/2)$, with $a/2$ and $b/2$ as rationals.*

By default, all integer arithmetic terms are associated with the pair $(1, 0)$, which means that they can be any integer.

The modular arithmetic domain $\mathcal{D}_{Mod}$ in Colibri2 is defined with $\mathcal{D}_{\text{inter}}$, where:

- $\mathcal{R}_{mod} = \mathbb{Q} * \mathbb{Q}$: A pair of rationals (a, b) representing the divisor and the remainder.
- $P_{\mathcal{A}} : (r : \mathcal{R}_{mod}) \to 2^A$
  The set of terms that are of the form $a\mathbb{Z}+b$, for a modular arithmetic domain (a, b).
- equal : $((a_1, b_1) : \mathcal{R}_{mod}) \to ((a_2, b_2) : \mathcal{R}_{mod}) \to \mathsf{Bool}$
  Comes down to checking whether $a_1 = a_2$ and $b_1 = b_2$.
- inter : $((a_1, b_1) : \mathcal{R}_{mod}) \to ((a_2, b_2) : \mathcal{R}_{mod}) \to \mathcal{R}_{mod}$ option
  Defined as:

```
let inter ((a1, b1) : R_mod) ((a2, b2) : R_mod): R_mod option =
  let b = b1 − b2 in
  if a1 = a2 then
    if a1 | b then Some (a1, b1) else None
  else
    let (g, ux, _) = gcd a1 a2 in
    if g | b then
      let l = lcm a1 a2 in
      let r = b1 + a1 * (b/g) * ux in
      Some (l, r)
    else None
```

  Where | is the divisibility operator ($a \mid b$ is true if a divides b), gcd computes the greatest common divisor between two terms, and lcm computes the least common multiple.

The modular arithmetic domain is used to quickly detect contradictions when two terms with incompatible modular representations are merged. It is also used in combination with the interval domain during propagation to further refine the interval domain, and during model generation to restrict the kinds of terms that can be generated and to ensure that they respect their modular representation in addition to falling within their interval domain.

3.1.2 Propagators

To ensure consistency between the constraints applied to arithmetic terms, it is necessary to propagate them. To do so, a generic waiting system is implemented in Colibri2, which creates daemons that wait on the domains of terms. When these domains are updated, the daemons will ensure that the domains of terms that depend on them are also updated.

The domain of intervals is one of the domains for which the generic waiting system is used. Given a binary operation $\diamond \in \{+, -, *, /\}$, with $\diamond^{-1}$ as $-$ if $\diamond$ is $+$ and $/$ if $\diamond$ is $*$ (and inversely). Given a constraint of the form $x = y \diamond z$, such that I_x, I_y, and I_z are respectively the interval domains of x, y, and z, the daemons wait on changes to the domains of x, y, and z such that:

- If the domain of x changes to I'_x, the domains of y and z are respectively updated with $I'_x \diamond^{-1} I_z$ and $I'_x \diamond^{-1} I_y$
- If the domain of y changes to I'_y, the domains of x and z are respectively updated with $I'_y \diamond I_z$ and $I_x \diamond^{-1} I'_y$
- If the domain of z changes to I'_z, the domains of x and y are respectively updated with $I_y \diamond I'_z$ and $I_x \diamond^{-1} I'_z$

Such propagations are scheduled one after another until a fixpoint is reached. That is, when additional propagations no longer change the domains of the terms.

Example 3.3. *Given the terms $x \in [0; 200]$, $y \in [0; 10]$, and $z \in [0; 10]$, and the constraints $c_1 : x = y \cdot z$, $c_2 : y \leq 5$, and $c_3 : x > 50$:*

- *c_1 updates the domain of x to $[0; 100]$, and a daemon is created for it.*
- *c_2 updates the domain of y to $[0; 5]$, which, through the daemon, updates the domain of x to $[0; 50]$.*
- *c_3 results in a contradiction, since no value $x \in [0; 50]$ can satisfy the constraint.*

3.2 Labeled Union-Find and The Constant Difference Relation

This section begins with a reminder about the union-find data structure [120] in Section 3.2.1. It then introduces the labeled union-find data structure [80] in Section 3.2.2, which is an extension of the classical union-find data structure that is used to represent weaker relations than equivalence.

One such relation is the constant difference relation between arithmetic terms, presented in Section 3.2.3. It is used to represent constraints of the form $a = b + c$, where a and b are arithmetic terms and c is a constant.

Finally, Section 3.2.4 explains how a Shostak theory [17] can be used to detect constant difference relations, as well as equalities and disequalities between arithmetic terms.

3.2.1 The Union-Find data structure

The union-find data structure [120], also called the disjoint-set data structure, was designed to organize elements into disjoint sets. It is notably used to represent equivalence relations and as the basis for congruence closure in automated reasoning (see Section 2.3.3).

In the union-find data structure, sets are usually represented as trees, where the root element is called the representative of the set.

Let E be the sort of the elements in the union-find. The union-find data structure maintains a mapping $\pi : E \to E$, such that for any element t:

- $\pi[t]$:
  - If $t \in Dom(\pi)$: $\pi[t]$ is the parent of t in the tree that represents the set to which t belongs in the union-find.
  - If $t \notin Dom(\pi)$ either:
    - t is the root of a tree and does not have a parent which makes it the representative of the set it is part of.
    - t was not added to the union-find and is therefore not part of any tree. In this case it is considered as being in its own singleton set with itself as the representative.
- $\pi[t \leftarrow t']$: updates π by setting t' as the parent of t.

```
1  let find π (x : E): E =
2    if x ∉ Dom(π) then x
3    else
4      let p = π[x] in
5      if p = x then x
6      else find π p
7
8  let union π (x : E) (y : E): unit =
9    let x_r = find π x in
10   let y_r = find π y in
11   if x_r = y_r then ()
12   else
13     if rand()
14     then π[x_r ← y_r]
15     else π[y_r ← x_r]
```

Listing 3.2: Implementations of the find and union functions in a union-find.

The union-find data structure is used through two functions, find and union, implementations of which are shown in Listing 3.2. As illustrated, the find function takes an element and returns the representative of the set to which it belongs, while the union function takes two elements: if they already belong to the same set (i.e. have the same representative), nothing is done. Otherwise, their respective sets are merged by selecting the representative of one of them to be the new representative and setting the other as its child.

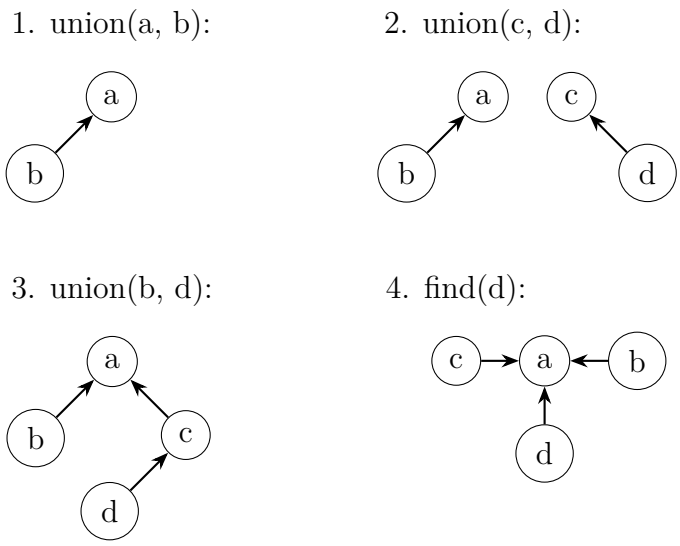

Figure 3.1: Example of the usage of a union-find data structure.

Various techniques are used to improve the performance of this data structure. A notable one is path compression, which is used when calling find(x) for any element x. It consists in setting $\pi[y \leftarrow \text{find}(x)]$ for every node y on the path that find traverses from x to its representative. This effectively makes all those elements point directly to the representative, allowing future calls to find on them to avoid traversing the full path again, unless the representative changes.

Another optimization is union by rank, which is a heuristic used when calling the union function on two elements x and y that belong to different sets. The optimization consists in choosing the root of the tree with the higher rank as the new representative of the resulting set. The rank of a tree is an upper bound on its height, computed when the tree is created, and not recomputed when its height is reduced due to path compression after calls to find. The purpose of this optimization is to minimize the increase in tree height.

These optimizations are commonly used when the union-find data structure is used for equivalence relations. As mentioned in Section 2.3.3, it is also used as a basis for congruence closure in which case variations of these optimizations are used [48, 105] which take into account the congruence axiom presented in Figure 2.10 and applications of functions to elements of the data structure.

Figure 3.1 illustrates an example of how a union-find data structure works. The first two calls to union simply join the elements $\{a, b\}$ and $\{c, d\}$. The third call uses the union by rank heuristic. Since the two trees have the same rank (1), a is arbitrarily chosen as the representative of the resulting tree. Finally, when find(d) is called, path compression is applied, so d's parent changes from c to a.

3.2.2 The Labeled Union-Find data structure

While the union-find data structure is used to represent equivalence relations, the labeled union-find [80] was developed as an extension of the union-find to parametrized equivalence relations where the edges in the union-find are labeled. The labels in the labeled

union-find must satisfy the group axioms presented in Definition 3.1. These properties are necessary for the construction and soundness of the labeled union-find, as well as for enabling path compression with labeled edges, and computing the relations between elements that are in the same class.

An example of such weaker relations is the constant difference relation which puts in relation two (real or integer) arithmetic terms x and y parametrized by a constant distance c such that $x = y + c$. With the labeled union-find, the value of the constant distance between arithmetic terms that are in relation with one another is expressed through the labels, which with the addition operation form a group.

Definition 3.1 (Group)**.** *A group is defined by a non-empty set G and a binary operation $\oplus : G \times G \to G$, called the composition operation, that combines two elements of G and produces an element of G, and that satisfies the following requirements:*

- *Associativity:* $\forall x, y, z \in G.\ (x \oplus y) \oplus z = x \oplus (y \oplus z)$
- *Existence of a neutral element $e_L \in G$:*

$$\forall x \in G.\ x \oplus e_L = x \wedge e_L \oplus x = x$$

- *Existence of an inverse element $x^{-1} \in G$ for each $x \in G$:*

$$\forall x \in G.\ x \oplus x^{-1} = e_L \wedge x^{-1} \oplus x = e_L \wedge (x^{-1})^{-1} = x$$

The labeled union-find is defined by the tuple: $\langle E, L, \oplus, \text{merge}, \text{conflict} \rangle$, where E is the sort of elements in the labeled union-find, L is the sort of labels, and $\oplus : L \times L \to L$ is a transitive label composition operation, such that the pair $(L, \oplus)$ forms a group.

The merge function determines what needs to be done when two elements are in relation with one another and the relation is parametrized with the neutral element e_L of the labels L, while the conflict function handles the case in which two elements are in relation with one another through two relations that are parametrized with different labels.

For example, when working on constant difference relations in the context of automated reasoning where the composition operation is addition $+$ and the neutral element is 0 then:

- Given $x = y + 0$, the merge function will have to propagate that x and y are equal.
- Given $x = y + c_1$ and $x = y + c_2$ such that $c_1 \neq c_2$, the conflict function will have to propagate a contradiction.

A mapping $\rho : E \to E \times L$ is maintained, which associates each element x with a pair (p, d_p), where p is the parent of x in the tree that represents x's class in the labeled union-find, and d_p is the label on the edge from x to p. If x has no parent, then $p = x$ and $d_p = e_L$.

Another mapping $\gamma : E \to (L \to E)$ associates each element x to a mapping in which a label d_i^{-1} is mapped to each child node n_i of x. Such that the label d_i^{-1} represents the inverse of the label d_i on the edge from n_i to x.

```
let find ρ (x : E) : E * L =
  if x ∉ ρ
  then (x, e_L)
  else
    let (p, d_p) = ρ[x] in
    if p = x then (p, d_p) (* d_p = e_L *)
    else
      let (r, d_r) = find ρ p in
      (r, d_p ⊕ d_r)

let add_nonrepr (ρ, γ) (x : E) (d : L) (r : E) : unit =
  if d = e_L then merge x r
  else
    let r_m = γ[r] in
    match Map.find_opt d r_m with
    | Some y →
      ρ[x ← ∅];
      merge x y
    | None →
      ρ[x ← (r, d)];
      let r_m = Map.add d^{-1} x r_m in
      γ[r ← r_m]

let add_relation (ρ, γ) (x : E) (d : L) (y : E) : unit =
  let (x_r, x_d) = find ρ x in
  let (y_r, y_d) = find ρ y in
  if x_r = y_r then
    if d ≠ x_d ⊕ y_d^{-1} then
      conflict (ρ, γ) x y d (x_d ⊕ y_d^{-1})
  else
    if rand()
    then
      let d_{x_r y_r} = x_d^{-1} ⊕ d ⊕ y_d in
      add_nonrepr (ρ, γ) x_r d_{x_r y_r} y_r
    else
      let d_{y_r x_r} = y_d^{-1} ⊕ d^{-1} ⊕ x_d in
      add_nonrepr ρ y_r d_{y_r x_r} x_r

let get_relation ρ (x : E) (y : E) : L option =
  let (x_r, x_d) = find ρ x in
  let (y_r, y_d) = find ρ y in
  if x_r = y_r
  then Some (x_d ⊕ y_d^{-1})
  else None
```

Listing 3.3: Implementations of the functions in a labeled union-find.

The labeled union-find differs from the classical union-find mainly in the operations it supports. Listing 3.3 shows implementations of its functions. The find function behaves

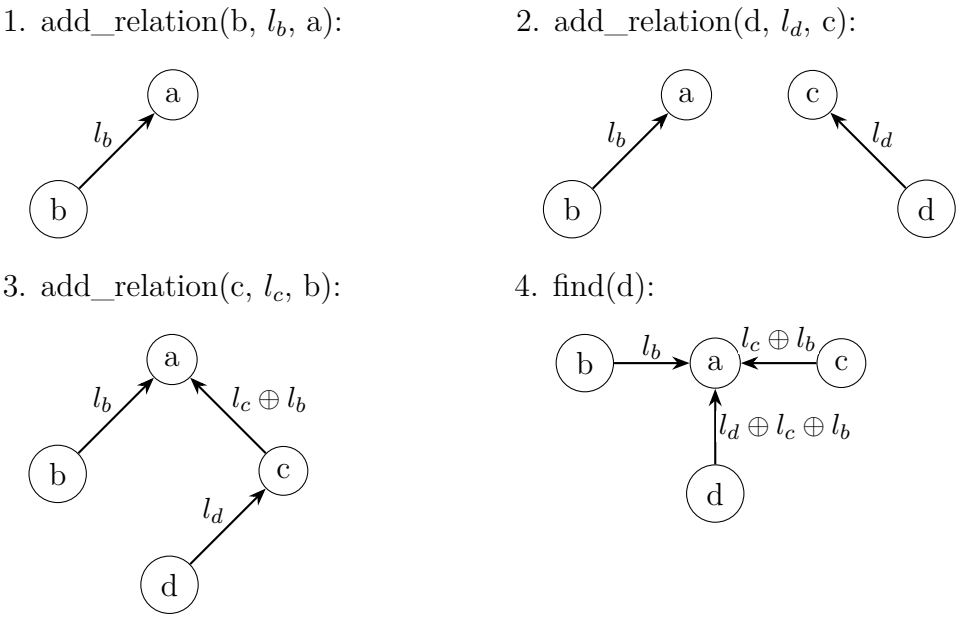

Figure 3.2: Example usage of the labeled union-find data structure.

similarly to that in Listing 3.2, and the *add_nonrepr* function is an internal auxiliary function that adds a non-representative element as a child with a label to a representative element.

Since the labeled union-find represents relations that are parameterized by a label, the union function is replaced with *add_relation*, which takes the label on the edge as an extra argument. Lines 25-26 show that if the two nodes already have the same representative but the provided label differs from the one that is computed from their positions in the tree, then a conflict occurs. The conflict function is a user-provided function that is called to handle such situations, in practice, it either raises a contradiction or tries to reconcile the labels when appropriate.

Another useful additional function is *get_relation*, which takes two nodes, computes and returns the label that represents the relation between them, if such relation exists.

Figure 3.2 illustrates how the labeled union-find works. Compared to Figure 3.1, the key differences are:

- The use of *add_relation*, which takes a label as an additional argument, instead of union.
- The usage of label composition:
  - When classes are merged in step 3 to compute the label between the old representative c and the representative of the resulting class a.
  - During path compression in step 4 to compute the new label from d to the representative a resulting from path compression.

In its implementation in Colibri2, the labeled union-find also supports a daemon that waits for a representative to become a non-representative, which occurs when sets are

merged. This daemon allows registering hook functions:

$$\mathsf{hook} : (o_r : E) \to (\delta : L) \to (n_r : E) \to \mathsf{unit}$$

These are called when a representative o_r is no longer a representative and starts pointing to node n_r with an edge labeled with δ.

Star topology

The most efficient topology for propagating relations between elements that are in the same tree is the star topology, in which all non-representative elements have a unique direct edge to the representative. This can be achieved with a labeled union-find, since the labels follow the group axioms.

```
1  let add_relation (ρ,γ) (x : E) (d : L) (y : E): unit =
2    let (x_r, x_d) = find ρ x in
3    let (y_r, y_d) = find ρ y in
4    if x_r = y_r then
5      if d ≠ x_d ⊕ y_d^-1 then conflict (ρ,γ) x y d (x_d ⊕ y_d^-1)
6    else
7      if rand()
8      then
9        let d_{x_r y_r} = x_d^-1 ⊕ d ⊕ y_d in
10       add_nonrepr (ρ,γ) x_r d_{x_r y_r} y_r;
11       Map.iter (fun d'^-1 n →
12         add_nonrepr (ρ,γ) n (d' ⊕ d_{x_r y_r}) y_r
13       ) (γ[x_r]);
14       γ[x_r ← ∅]
15     else
16       let d_{y_r x_r} = y_d^-1 ⊕ d^-1 ⊕ x_d in
17       add_nonrepr (ρ,γ) y_r d_{y_r x_r} x_r;
18       Map.iter (fun d'^-1 n →
19         add_nonrepr (ρ,γ) n (d' ⊕ d_{y_r x_r}) x_r
20       ) (γ[y_r]);
21       γ[y_r ← ∅]
```

Listing 3.4: Implementation of the *add_relation* function in the labeled union-find data structure in which the trees follow the star topology.

To do so, the implementation of the *add_relation* function from Listing 3.3 needs to be replaced by the one in Listing 3.4. The difference lies in lines 9-14 and 16-21, when adding a relation between two elements x and y that belong to different classes in which the representatives are respectively x_r and y_r, instead of simply making one representative a child of the other (e.g. making y_r the child of x_r), the new *add_relation* function will add a representative and all the non-representative elements in its class as children of the new representative. This ensures that the trees always follow the star topology.

There are multiple advantages to relying on this topology. It is notably the most efficient for computing relations between elements in a tree, since computing a relation requires at most one composition (when both elements are non-representatives).

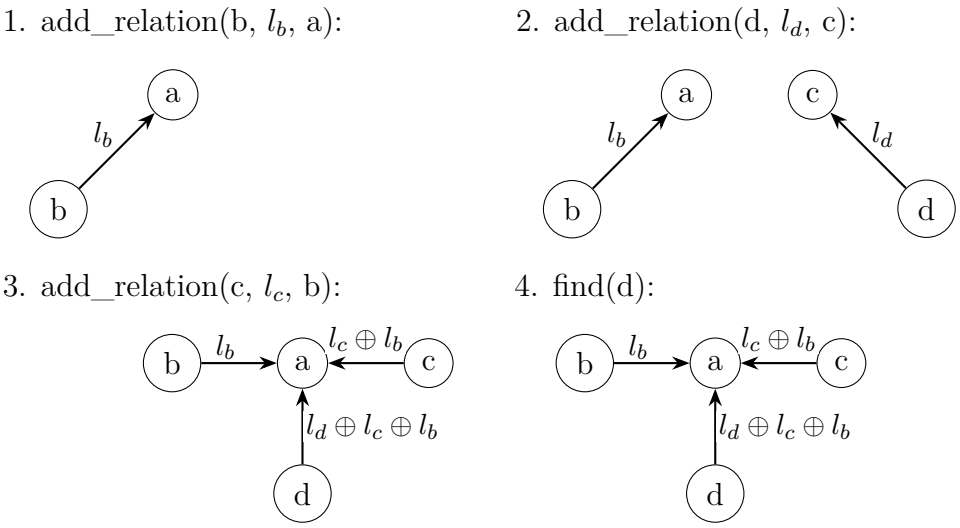

Figure 3.3: Example of the usage of the labeled union-find data structure following the star topology.

Another advantage is merge deduction, since all non-representatives n_i have an edge labeled with d_i to the representative r, if a new non-representative n_2 is added to r with an edge labeled with d_2, and there already exists a non-representative n_1 with an edge labeled with d_1 to r such that $d_1 \oplus^{-1} d_2 = e_L$ (i.e. $d_1 = d_2$), then the merge function is called on n_1 and n_2, in other words, a merge is deduced between n_1 and n_2. This would not be doable straightforwardly if n_2 were attached to a deeper element in the tree.

To go further, ensuring that $\{e_L \mapsto r\} \subset \gamma[r]$ holds for each representative r allows deducing mergers with the representative whenever a non-representative is added with label e_L.

Figure 3.3 is a modified version of Figure 3.2 in which the star topology is followed. In this case, the call to $add\_relation(c, l_c, b)$ in step 3 immediately applies the path compression between d and a as part of the class merger, and the call to find in step 4 does not change the structure of the tree.

Normalization

Since labels can be any set of values that respect the group axioms, they can, for example, be polynomials, since:

- Composition is addition on polynomials: $(\text{comp}(x, y) = x + y)$.
- The neutral element e_L is 0: $(\forall x.\ x + 0 = 0 + x = x)$.
- The inverse function is negation: $(\text{inv}(x) = -x)$.

As mentioned in Section 3.1.1, in Colibri2 polynomials are normalized. Whenever the polynomial domain of a variable term is set, the variable term is replaced in all the polynomials in which it occurs with its new polynomial domain. Therefore, to maintain this normalization in the labeled union-find as well, it is necessary to be able to substitute a label with a new one when its normal form changes. If normalization is not maintained,

some deductions might be missed, as labels can be syntactically distinct yet semantically equivalent.

```
let subst_label (ρ, γ) (x : E) (d1 : L) (d2 : L): unit =
  let (r, _) = find ρ x in
  if d2 = eL then merge x r
  else
    let rm = γ[r] in
    let rm = Map.remove d1^-1 rm in
    match Map.find_opt d2 rm with
    | Some y →
      ρ[x ← ∅];
      merge x y
    | None →
      ρ[x ← (r, d2)];
      let rm = Map.add d2^-1 x rm in
      γ[r ← rm]
```

Listing 3.5: Implementation of the *subst_label* function in the labeled union-find data structure in which the trees follow the star topology.

Listing 3.5 shows an implementation of the *subst_label* function, which takes an element x and two labels d_1 and d_2, such that d_1 is the current label on the edge from x to the representative of its class, and d_2 is the new label meant to replace d_1 for normalization purposes. For this function to be usable, it is necessary to maintain a mapping from labels to the elements that serve as sources for the edges in which these labels occur.

3.2.3 Constant Difference Relation

The constant difference relation is a binary relation between arithmetic terms that is parametrized by a constant. It associates to a term x a term y with a parameter constant c to represent that $x = y + c$. This relation has many uses, notably, it helps with equality and inequality detection. Given $x = y + c$, if $c = 0$, then $x = y$ can be deduced, and if $c \neq 0$, then $x \neq y$ can be deduced. The relation can also serve for constraint propagation between arithmetic terms that have a constant difference between them.

In practice, the relation specifically associates polynomials of the form $p_k = \sum_i c_i v_i + c_k$ that share the same variable part $\sum_i c_i v_i$ and differ only in the constant part c_k. In Colibri2, the constant difference relations are represented using a labeled union-find data structure, in which the sort of elements E is the semantic values that represent arithmetic terms, and the sort of L is the sort of real (resp. integer) constants, and $\oplus$ is addition on real (resp. integer) constants. The merge function does the propagation of equalities, while the conflict function simply raises a contradiction.

The implementation of the labeled union-find for this relation differs from the generic one. In the original, the representative of a given set is chosen arbitrarily as operations are performed on the data structure. In this implementation, for each set of arithmetic terms sharing the same variable part, the term corresponding to the variable part $p_0 = \sum_i c_i v_i$ is used as the representative of the set. The other terms are then of the form $p_j = p_0 + c_j$, such that $j \neq 0$ and each c_j is unique and different from zero. The edges in the labeled

union-find go from each non-representative term p_j to the representative p_0 and are labeled by the constant difference c_j.

This relation uses polynomials, but it is also used by the domain of polynomials as it helps in normalizing them. Whenever a polynomial $p = \sum_i c_i v_i + c$ is created, each variable term v_i is replaced by $p_k = \sum_i c_i(r_i+\delta_i)+c$, where the pair (r_i, δ_i) is obtained from calling the find function of the labeled union-find that represents the constant difference relations on v_i. That is, r_i is the representative of v_i with respect to the constant difference relation. To maintain this normalization, whenever there is a representative change from r_i to r'_i with a label δ', every occurrence of r_i in a polynomial term is replaced by $r'_i + \delta'$.

Whenever two elements x and y are propagated as equal, another normalization is done by adding a relation between x and y with an edge label e_L, ensuring that two different elements in the labeled union-find do not represent the same semantic values and that the labeled union-find knows they are equal.

3.2.4 Shostak Theories and Constant Difference Relations

Some constant difference relations can appear explicitly in the constraints that occur in a given problem. Given the arithmetic terms x, y, and z, and the constraints $C_1 : x = p_0$, $C_2 : y = x + c_1$, and $C_3 : z = x + c_2$, where p_0 is a zero constant polynomial and c_1 and c_2 are constants, the constraint C_2 represents a constant difference relation between x and y with the constant c_1, and C_3 represents a constant difference relation between x and z with the constant c_2.

However, other constant difference relations do not appear explicitly and need to be computed. One way to do that is by using a Shostak theory solver [17, 112, 114] over linear arithmetic.

A Shostak theory $\mathcal{T}$ is defined in Barrett, Dill, and Stump [17], with a signature Σ that does not contain predicate symbols, a canonizer function canon, and a solver function solve. The theory also has to be convex, i.e. any set of constraints from the theory that entails a disjunction of equalities also entails at least one of these equalities.

A Shostak theory $\mathcal{T}$ is able to decide if a given set of equations E in Σ implies an equality $t_1 = t_2$ in $\mathcal{T}$.

The solve function produces a substitution from an equation, and the canon function applies the substitution and normalizes a term ($\mathcal{T} \cup E \models t_1 = t_2$ if and only if $\mathsf{canon}(t_1) \equiv \mathsf{canon}(t_2)$, where $\equiv$ denotes syntactic equality).

The algorithm incrementally computes equisatisfiable sets S_i of equations where the left-hand side of each equation is a variable that appears only once in the set. These sets of equations are considered as substitutions of the variables with their right-hand sides.

The theory of linear real arithmetic is a good example of a Shostak theory. In it, the canon function is obtained from an ordering on the variables and a simplification of the arithmetic terms, and the solve function consists of Gaussian elimination (expressing one variable as a linear equation using the others).

Example 3.4. *Given the set of equations:*

$$E = \{\overbrace{-z + y - u = 0}^{e_1}, \overbrace{x + 2z = 2z - u}^{e_2}, \overbrace{-t - 2y = z + 2v}^{e_3}, \overbrace{z - 2 = -y - v}^{e_4}, \overbrace{w + 4y = u - 2v + 4}^{e_5}\}$$

Given $\sigma_i \triangleq \mathsf{solve}(e_i)$*, with* $S_0 \triangleq \emptyset$ *and* $S_i \triangleq \sigma_i[S_{i-1}] \cup \sigma_i$ *for* $i \geq 1$*, where* $\sigma_i[S_{i-1}]$ *is the application of the substitution* σ_i *to the set* S_{i-1}*. The Shostak theory solver processes the equations as follows:*

- *With* $i = 1$ *and* $\mathsf{solve}(\{\overbrace{-z + y - u = 0}^{e_1}\}) = \{u = y - z\}$ *(Rewriting):*
  - $\sigma_1 = \{u = y - z\}$
  - $S_1 = \{u = y - z\}$
- *With* $i = 2$ *and* $\mathsf{solve}(\{\overbrace{x + 2z = 2z - u}^{e_2}\}) = \{x = z - y\}$ *(after applying the substitution* $\{u = y - z\}$ *in* e_2*:* $x + 2z = 3z - y$*, and subtracting* $2z$ *from both sides:* $x = z - y$*):*
  - $\sigma_2 = \{x = z - y\}$
  - $S_2 = \{u = y - z, x = z - y\}$
- *With* $i = 3$ *and* $\mathsf{solve}(\{\overbrace{-t - 2y = z + 2v}^{e_3}\}) = \{t = -2y - 2v - z\}$ *(Rewriting):*
  - $\sigma_3 = \{t = -2y - 2v - z\}$
  - $S_3 = \{u = y - z, x = z - y, t = -2y - 2v - z\}$
- *With* $i = 4$ *and* $\mathsf{solve}(\{\overbrace{z - 2 = -y - v}^{e_4}\}) = \{z = -y - v + 2\}$ *(Rewriting):*
  - $\sigma_4 = \{z = -y - v + 2\}$
  - $\sigma_4[S_3] = \{u = 2y + v - 2, x = -2y - v + 2, t = -y - v - 2\}$
  - $S_4 = \{u = 2y + v - 2, x = -2y - v + 2, t = -y - v - 2, z = -y - v + 2\}$
- *With* $i = 5$ *and* $\mathsf{solve}(\{\overbrace{w + 4y = u - 2v + 4}^{e_5}\}) = \{w = -2y - v + 2\}$ *(With rewriting:* $w = u - 4y - 2v + 4$*, applying the substitution* σ_1*:* $w = -z - 3y - 2v + 4$*, and applying the substitution* σ_4*:* $w = -2y - v + 2$*):*
  - $\sigma_5 = \{w = -2y - v + 2\}$
  - $S_5 = \{u = 2y + v - 2, x = -2y - v + 2, t = -y - v - 2, z = -y - v + 2, w = -2y - v + 2\}$

With S_5 *and through* canon*:* $y = -w$ *and* $x = w$ *are deduced. It is also possible to deduce from* S_5 *that* $z = t + 4$*.*

The entailed equalities between variables (such as $x = w$ in S_5 in Example 3.4) can be found by keeping a reverse mapping M from the canonized right-hand side to a representative of the left-hand side for each equality. This is useful for propagating equalities among theories.

Moreover, a union-find-like data structure Δ can be used to remember that $x = w$ and store the right-hand side, $-2y - v + 2$ in S_5, only once at the representative of x and w (as described in Conchon, Contejean, Kanig, and Lescuyer [40]). Additionally, it avoids substituting in both definitions of x and w if future substitutions of their variables y or v need to be made.

Extension with a Labeled Union-Find

A labeled union-find data structure can be used to further factorize the equations and find disequalities between variables. Given a set of labels L that follow the group axioms, an extension to a Shostak theory requires, in addition to the canon function, a canon_rel function that takes a term t and returns a representative term t' and a label l: $\mathsf{canon\_rel}(t) = (t', l)$.

Proposition 3.1 (Equality). *For any two terms t_1 and t_2, with $\mathsf{canon\_rel}(t_1) = (t'_1, l_1)$ and $\mathsf{canon\_rel}(t_2) = (t'_2, l_2)$:*

$$t'_1 = t'_2 \wedge l_1 = l_2 \iff t_1 = t_2$$

Proposition 3.2 (Disequality). *For any two terms t_1 and t_2, with $\mathsf{canon\_rel}(t_1) = (t'_1, l_1)$ and $\mathsf{canon\_rel}(t_2) = (t'_2, l_2)$:*

$$t'_1 = t'_2 \wedge l_1 \neq l_2 \implies t_1 \neq t_2$$

Definition 3.2 (Group Action). *Given a group G and a set S, a group action $\mathcal{A}$ is a function:*

$$\mathcal{A} : G \times S \to S$$

such that the following properties hold:

- *For e_L, the neutral element in the group G:*

$$\forall x \in S.\ \mathcal{A}(e_L, x) = x$$

- *Associativity:*

$$\forall g, h \in G, x \in S.\ \mathcal{A}(g, \mathcal{A}(h, x)) = \mathcal{A}(g \oplus h, x)$$

  where $\oplus$ is the composition function of the group G.

A group action function $\mathcal{A}$ that rebuilds a term from a term and a label is also needed. Formally, if $\mathsf{canon\_rel}(t) = (t', l)$, then $\mathcal{A}(t', l) = \mathsf{canon}(t)$. Given these building blocks, the M and Δ mappings presented in Section 3.2.4 can be optimized. In M, the reverse mapping for $\mathsf{canon\_rel}(t_i) = (t', l_i)$ only needs to map t' to all t_i terms. And Δ can become a labeled union-find with label l, so that in addition to storing the right-hand side once for all equal elements, it also stores it once for all elements that are in relation with one another, decreasing the number of necessary substitutions.

Detection of Constant Difference Relations

Example 3.5. *Using the constant difference abstract relation (Section 3.2.3), the canon_rel function separates the constant part from the rest after normalization, e.g.:*

$$\mathit{canon\_rel}(-y - v - 2) = (-y - v, -2)$$

In S_4 (Example 3.4), it allows storing and substituting only in $t = -y - v - 2$, while keeping $z = t + 4$ in the labeled union-find Δ.

Example 3.5 illustrates how a constant difference relation was detected between the terms z and t through the constraint $z = t + 4$. Given Proposition 3.2, the disequality $t \neq z$ is also entailed from $\mathsf{canon\_rel}(t) = (-y - v, -2)$ and $\mathsf{canon\_rel}(z) = (-y - v, 2)$ (although the constant difference relation with a non-zero constant is sufficient to deduce disequality). Similarly, with Proposition 3.1 equalities can be entailed when the representatives and the labels obtained from canon_rel are the same.

Example 3.6. *With the equations from Example 3.4, if $t \in [0; 10]$, propagation of intervals on arithmetic operators and equalities cannot propagate any information through e_3, since, for example, propagation from $z + 2v$ to z is imprecise. However, with the labeled union-find Δ for constant differences from Example 3.5, $z \in [4; 14]$ is directly deduced.*

The collaboration between the Shostak theory of linear arithmetic and the constant difference abstract relation provides a qualitative gain on other domains. The newly found relational information allows new propagations that were not possible before, as shown in Example 3.6.

Extension to TVPE

The TVPE (Two-variables per equality) relation handles constraints of the form $y = ax+b$, where x and y are variables and a and b are constants. The TVPE relation can be represented using a labeled union-find in which the nodes are the variables and the edges are labeled with a pair (a, b) such that an edge from y to x, labeled with (a, b), indicates that $y = ax + b$. In this case:

- The composition function is defined as: $(a, b) \oplus (c, d) = (ac, ad + b)$
- The neutral element e_L is: $(1, 0)$
- The inverse function is: $(a, b)^{-1} = (1/a, -b)$.

Using the TVPE relation instead of constant difference would provide even more benefits, since $x = 3u+3v$ and $y = 8u+8v$ (related by $(8/3, 0)$) would be factorized, and their domains would be propagated directly. However, since it does not guarantee that every label is unique, some care is needed during conflict to propagate the learned constants.

Alternatively, a simpler extension could be used by ensuring that a key and its negation cannot both be bound in M. That way, when adding $\{(-2y - v + 2) \mapsto x\}$ to $\{(2y + v - 2) \mapsto u\}$, since $(2y + v - 2) \mapsto u$ was already added, the equality $u = -x$ is deduced, and M is not updated.

The map updating function is implemented as follows:

```
(* Adds the binding r ↦ l, from the equality r = l, to the mapping
   M *)
let update_map M r l =
  if r ∈ M then merge l M[r] else
  if (−r) ∈ M then merge (−l) M[r] else
    M[r ← l]
```

This extension combined with the constant difference abstract relation would allow for more propagations, without the difficulties that come with fully implementing TVPE. Although, it is clearly not as powerful as TVPE.

Chapter 4

n-Indexed Sequences I: Reasoning

The theory of n-indexed sequences [2, 4] is a variant of the theory of sequences. In it, n-indexed sequences, also called n-sequences, are defined as ordered collections of values of the same sort, indexed by integers from a first index n to a last index m. Such sequences are present in some programming languages, such as Ada. There is no dedicated theory for such sequences and no decision procedure to reason about them. Reasoning over them cannot be achieved straightforwardly using existing theories, such as the theories of arrays and sequences, because doing so would require extensions and axiomatizations, eventually with quantifiers, over these theories.

This chapter explores the topic of n-indexed sequences, proposes an SMT theory of n-indexed sequences, its signature and semantics, as well as different approaches to reasoning over it. These reasoning approaches include leveraging existing theories and adapting calculi designed for the theory of sequences to the theory of n-indexed sequences.

4.1 Syntax and Semantics

SMT-LIB symbol	Sort	Notation
nseq.first	NSeq(E) → Int	f_
nseq.last	NSeq(E) → Int	l_
nseq.get	NSeq(E) → Int → E	get(_, _)
nseq.set	NSeq(E) → Int → E → NSeq(E)	set(_, _, _)
nseq.const	Int → Int → E → NSeq(E)	const(_, _, _)
nseq.relocate	NSeq(E) → Int → NSeq(E)	relocate(_, _)
nseq.concat	NSeq(E) → NSeq(E) → NSeq(E)	concat(_, _)
nseq.slice	NSeq(E) → Int → Int → NSeq(E)	slice(_, _, _)
nseq.update	NSeq(E) → NSeq(E) → NSeq(E)	update(_, _)

Table 4.1: The signature of the theory of n-indexed sequences.

This section presents the theory of n-indexed sequences. Its signature is shown in Table 4.1, along with the notations for the symbols of the theory.

Definition 4.1 (Bounds)**.** *The bounds of an n-sequence s are its first and last indices, respectively denoted as f_s and l_s, and correspond to the values returned by the functions*

nseq.first(s) *and* *nseq.last*(s)*. An index* i *is said to be within the bounds of an* n*-sequence* s *if:*

$$\mathsf{f}_s \leq i \leq \mathsf{l}_s.$$

Definition 4.2 (Empty n-sequence)**.** *An* n*-sequence* s *is said to be empty if* $\mathsf{l}_s < \mathsf{f}_s$*. Two empty* n*-sequences* s_1 *and* s_2 *are equal if* $\mathsf{f}_{s_1} = \mathsf{f}_{s_2}$ *and* $\mathsf{l}_{s_1} = \mathsf{l}_{s_2}$*. Otherwise, they are distinct.*

The following list describes the semantics of each symbol in the theory:

- f_s: the first index of s.
- l_s: the last index of s.
- $\mathsf{get}(s, i)$: If $\mathsf{f}_s \leq i \leq \mathsf{l}_s$, returns the element associated with i in s, otherwise, returns an uninterpreted value.
- $\mathsf{set}(s_1, i, v)$: If $\mathsf{f}_{s_1} \leq i \leq \mathsf{l}_{s_1}$, creates a new n-sequence s_2 with the same bounds as s_1, where $\forall k.\ \mathsf{f}_{s_1} \leq k \leq \mathsf{l}_{s_1} \implies \mathsf{get}(s_2, k) = \mathsf{ite}(k = i, v, \mathsf{get}(s_1, k))$. Otherwise, returns s_1.
- $\mathsf{const}(f, l, v)$: Creates an n-sequence s with $\mathsf{f}_s = f$ and $\mathsf{l}_s = l$, where $\forall k.\ f \leq k \leq l \implies \mathsf{get}(s, k) = v$.
- $\mathsf{relocate}(s_1, f)$: Given an n-sequence s_1 and an index f, returns a new n-sequence s_2 with $\mathsf{f}_{s_2} = f$ and $\mathsf{l}_{s_2} = f + \mathsf{l}_{s_1} - \mathsf{f}_{s_1}$, where $\forall k.\ f \leq k \leq \mathsf{l}_{s_2} \implies \mathsf{get}(s_2, k) = \mathsf{get}(s_1, k - \mathsf{f}_{s_2} + \mathsf{f}_{s_1})$.
- $\mathsf{concat}(s_1, s_2)$: If s_1 is empty, returns s_2. If s_2 is empty, returns s_1. If $\mathsf{f}_{s_2} = \mathsf{l}_{s_1} + 1$, returns a new n-sequence s_3 with $\mathsf{f}_{s_3} = \mathsf{f}_{s_1}$ and $\mathsf{l}_{s_3} = \mathsf{l}_{s_2}$, where $\forall k.\ \mathsf{f}_{s_1} \leq k \leq \mathsf{l}_{s_2} \implies \mathsf{get}(s_3, k) = \mathsf{ite}(k \leq \mathsf{l}_{s_1}, \mathsf{get}(s_1, k), \mathsf{get}(s_2, k))$. Otherwise, returns s_1.
- $\mathsf{slice}(s_1, f, l)$: If $\mathsf{f}_{s_1} \leq f \leq l \leq \mathsf{l}_{s_1}$, returns a new n-sequence s_2 with $\mathsf{f}_{s_2} = f$ and $\mathsf{l}_{s_2} = l$, where $\forall k.\ f \leq k \leq l \implies \mathsf{get}(s_2, k) = \mathsf{get}(s_1, k)$. Otherwise, returns s_1.
- $\mathsf{update}(s_1, s_2)$: If s_1 is empty, s_2 is empty, or the property $\mathsf{f}_{s_1} \leq \mathsf{f}_{s_2} \leq \mathsf{l}_{s_2} \leq \mathsf{l}_{s_1}$ does not hold, returns s_1. Otherwise, returns a new n-sequence s_3 with the same bounds as s_1, where $\forall k.\ \mathsf{f}_{s_1} \leq k \leq \mathsf{l}_{s_1} \implies \mathsf{get}(s_3, k) = \mathsf{ite}(\mathsf{f}_{s_2} \leq k \leq \mathsf{l}_{s_2}, \mathsf{get}(s_2, k), \mathsf{get}(s_1, k))$.

Definition 4.3 (Extensionality)**.** *The theory of* n*-indexed sequences is extensional, which means that* n*-sequences that have the same bounds and contain the same elements are equal. Therefore, given two* n*-sequences* s_1 *and* s_2*:*

$$s_1 = s_2 \equiv$$
$$\mathsf{f}_{s_1} = \mathsf{f}_{s_2} \wedge \mathsf{l}_{s_1} = \mathsf{l}_{s_2} \wedge (\forall i.\ \mathsf{f}_{s_1} \leq i \leq \mathsf{l}_{s_1} \rightarrow \mathit{get}(s_1, i) = \mathit{get}(s_2, i))$$

Different semantics can be chosen for the functions of this theory, particularly for the slice and update functions. Ait-El-Hara, Bobot, and Bury [3] defined a set of theory design criteria. In particular, it shows that previously proposed semantics for the update

function in the theory of sequences are not symmetric: an overlapping update on the right behaves differently from one on the left, which does not align with the design criterion of avoiding surprising users. Instead, a symmetric semantics was proposed: in all cases of overlapping update, the shared indices are updated. It can similarly be adapted for the slice function, by ensuring that the resulting slice is always equal to the intersection between the bounds of the slice and the bounds of the original n-sequence.

However, a different, yet still symmetrical, semantics was chosen. This semantics consists of not updating the n-sequence whenever the update overlaps its bounds. This choice is justified by the main use case of n-indexed sequences, which is representing arrays from the Ada programming language, where arrays can be defined over an arbitrary range of integers. That is also the reason for the choice of the semantics of the concat and slice functions.

4.2 Reasoning with existing theories

When trying to reason over a theory that is not standard and that SMT solvers do not support, it is sometimes possible to encode formulas in this theory using other theories while maintaining the formulas' semantics. This is typically done by declaring all the symbols of the theory (sorts and functions) as uninterpreted symbols and then defining the theory's semantics through axioms that specify the interpretation of the symbols. For functions and predicates, it is sometimes possible to define them directly as interpreted functions. In fact, this is preferable to using axioms with quantifiers, since handling quantifiers is known to be challenging in SMT solvers [59].

4.2.1 Encoding n-Indexed Sequences using Sequences and Algebraic Data Types

An alternative way to reason over the theory of n-indexed sequences, that uses existing theories while relying less on axioms and quantifiers, is to encode the theory of n-indexed sequences using the theories of sequences and algebraic data types.

Listing 4.1 illustrates part of the encoding of the theory of n-indexed sequences using the theories of sequences and algebraic data types. The sort of n-sequences is defined as a product type composed of a pair of two values (lines 1 and 2). The first value can be accessed with nseq.first, it is an integer that represents the first index of the n-sequence. The second value can be accessed with nseq.seq, it is a sequence that represents the elements of the n-sequence. With this representation, an n-indexed sequence is essentially seen as a relocated or offsetted 0-indexed sequence.

With this defined sort of n-sequences, the last function, which returns the last index of an n-sequence, is defined as the sum of the first index and the length of the sequence, minus one (lines 4 and 5). This definition leads to a semantic difference between this version of the theory of n-indexed sequences and the one described in Section 4.1. The difference lies in the definition of an empty n-sequence (cf. Definition 4.2). In this version, empty n-sequences always have a last index equal to their first index minus one, meaning that all empty n-sequences with the same first index are equal. In contrast, in the original semantics, an empty n-sequence is defined as one whose last index is less than

```
(declare-datatype NSeq (par (T)
  ((nseq.mk (nseq.first Int) (nseq.seq (Seq T))))))

(define-fun nseq.last (par (T) ((s (NSeq T))) Int
  (+ (- (seq.len (nseq.seq s)) 1) (nseq.first s))))

(define-fun nseq.get (par (T) ((s (NSeq T)) (i Int)) T
  (seq.nth (nseq.seq s) (- i (nseq.first s)))))

(define-fun nseq.set (par (T)
  ((s (NSeq T)) (i Int) (v T)) (NSeq T)
    (nseq.mk (nseq.first s)
      (seq.update
        (nseq.seq s) (- i (nseq.first s)) (seq.unit v)))))
```

Listing 4.1: The sort of n-indexed sequences defined using the sort of sequences and a product type, and the definitions of the last, set, and get functions over this sort.

its first index, and the last index can be any value smaller than the first index, therefore, empty n-sequences can have the same first index but different last indices and therefore be distinct.

The encoding could be made faithful to the original theory by adding a specific constructor in the algebraic data type to represent the last index of the n-sequence, allowing it to take any value less than the first index in the case of empty n-sequences. However, this would ultimately hinder the performance of solvers that use this encoding, especially since the difference in semantics is not problematic. Empty n-sequences typically represent corner cases or failure scenarios, and the position of their last index is usually irrelevant for determining satisfiability.

The definitions of the get (lines 7 and 8) and set (lines 10 to 14) functions rely on the nth and update functions from the theory of sequences. They simply shift the index by subtracting the n-sequence's first index to get the right element in the underlying sequence that contains the n-sequence's elements.

```
(declare-fun nseq.const (par (T) (Int Int T) (NSeq T)))

; "nseq.const"
(assert (par (T) (forall ((f Int) (l Int) (v T))
  (!
    (let ((s (nseq.const f l v)))
      (and
        (= (nseq.first s) f)
        (= (nseq.last s) l)
        (forall ((i Int)) (=>
          (and (<= f i) (<= i l))
          (= (nseq.get s i) v)))))
  :pattern ((nseq.const f l v))))))

(define-fun nseq.relocate
```

```
16    (par (T) ((s (NSeq T)) (f Int)) (NSeq T)
17        (nseq.mk f (nseq.seq s))))
18
19  (define-fun nseq.concat
20    (par (T) ((s1 (NSeq T)) (s2 (NSeq T))) (NSeq T)
21      (ite (< (nseq.last s1) (nseq.first s1))
22        s2
23        (ite
24          (or
25            (< (nseq.last s2) (nseq.first s2))
26            (not (= (nseq.first s2) (+ (nseq.last s1) 1))))
27          s1
28          (nseq.mk
29            (nseq.first s1)
30            (seq.++ (nseq.seq s1) (nseq.seq s2)))))))
31
32  (define-fun nseq.slice
33    (par (T) ((s (NSeq T)) (f Int) (l Int)) (NSeq T)
34      (ite
35        (and
36          (<= f l)
37          (and (<= (nseq.first s) f) (<= l (nseq.last s))))
38        (nseq.mk f (seq.extract (nseq.seq s) (- f (nseq.first s))
              (+ (- l f) 1)))
39        s)))
40
41  (define-fun nseq.update
42    (par (T) ((s1 (NSeq T)) (s2 (NSeq T))) (NSeq T)
43      (ite
44        (and
45          (<= (nseq.first s2) (nseq.last s2))
46          (<= (nseq.first s1) (nseq.first s2))
47          (<= (nseq.last s2) (nseq.last s1)))
48        (nseq.mk (nseq.first s1)
49          (seq.update
50            (nseq.seq s1)
51            (- (nseq.first s2) (nseq.first s1))
52            (nseq.seq s2)))
53          s1)))
```

Listing 4.2: The declaration and axiomatization of the const function, and the definitions of the functions relocate, concat, slice, and update from the theory of n-indexed sequences, when the theory is encoded using the theories of sequences and algebraic data types.

The other symbols of the theory of n-indexed sequences, encoded with the symbols of the theories of sequences and algebraic data types, are presented in Listing 4.2. While most symbols are encoded as definitions, there is an exception with the const function, the semantics of which need to be axiomatized using a quantifier (lines 1 to 13), since there is no direct counterpart for this function in the theory of sequences.

The definition of the relocate function simply creates a new n-sequence tuple in which the first index is the index that was provided as an argument to the function.

Although this approach makes it possible to reason about n-indexed sequences, it is not ideal to depend on two different theories to do so, as it implies that the performance of reasoning about the theory becomes tied to the performance of reasoning over both underlying theories.

Furthermore, the definitions of the concat (lines 19 to 30), slice (lines 32 to 39), and update (lines 41 to 53) functions, which rely on their counterparts in the theory of sequences, are relatively complex due to differences in semantics between the two theories. This complexity could potentially lead to performance issues in practice, since such complex definitions are likely to be costly for solvers to handle. The complexity notably comes from the usage of ite to separate the different cases of the definitions.

4.3 Porting Calculi from the Theory of Sequences to the Theory of n-Indexed Sequences

Since the theory of n-indexed sequences shares similarities with the theory of sequences, and many symbols serve the same purpose in both, it is natural, when developing a calculus dedicated to the theory of n-indexed sequences, to base it on calculi developed for the theory of sequences.

The reasoning over the theory of n-indexed sequences was based on the calculi developed by Sheng, Nötzli, Reynolds, Zohar, Dill, Grieskamp, Park, Qadeer, Barrett, and Tinelli [113] (cf. Section 2.6.2) for the theory of sequences, in which two calculi were proposed.

The first calculus is called the BASE calculus, which is based on a string theory calculus [19, 81]. The BASE calculus reduces the functions of the theory of sequences into concatenations of subsequences, which are split when necessary to give sequences a normal form as concatenations of subsequences. This calculus effectively reduces the problem of reasoning over sequences into a problem of word equations [58].

The second calculus is called the EXT calculus. While most operations in EXT are handled similarly to BASE, it differs in its handling of the functions for selecting and storing an element at an index, by using array-like reasoning instead of reducing them to concatenations.

The differences in syntax and semantics between the theory of n-indexed sequences and the theory of sequences lie in the following symbols:

- const and relocate do not appear in the theory of sequences, while seq.empty, seq.unit, and seq.len do not appear in the theory of n-indexed sequences.
- The seq.nth function corresponds to the get function in the theory of n-indexed sequences.
- seq.update from the theory of sequences [113], with an element as the third argument, corresponds to set in the theory of n-indexed sequences, while seq.update

with a sequence as the third argument[1], corresponds to update in the theory of n-indexed sequences, which takes only two n-sequences as arguments.

- seq.extract in the theory of sequences takes a sequence, an offset, and a length, and corresponds to slice in the theory of n-indexed sequences, which takes an n-sequence, a first index, and a last index.
- The concatenation function (seq.++) in the theory of sequences is n-ary, while the concat function in the theory of n-indexed sequences is binary.

Therefore, it was necessary to make substantial changes to the calculi for the theory of sequences to adapt them to the theory of n-indexed sequences. The versions of the BASE and EXT calculi that were developed for n-indexed sequences are referred to as the NS-BASE and NS-EXT calculi, respectively. In this section, the resulting calculi are presented.

4.3.1 Reasoning over Relocation

Since n-indexed sequences can have as a first index any integer term, having a relocation function in the signature of the theory is natural as it allows having multiple n-indexed sequences that contain the same elements, but do not start at the same index.

Definition 4.4 (Equivalence Modulo Relocation)**.** *Given two n-sequences s_1 and s_2, they are said to be equivalent modulo relocation, denoted by the relation $s_1 =_{reloc} s_2$, such that:*

$$s_1 =_{reloc} s_2 \equiv$$
$$\mathsf{l}_{s_2} = \mathsf{l}_{s_1} - \mathsf{f}_{s_1} + \mathsf{f}_{s_2} \wedge \forall i : \mathit{Int}.\ \mathsf{f}_{s_1} \leq i \leq \mathsf{l}_{s_1} \implies \mathit{get}(s_1, i) = \mathit{get}(s_2, i - \mathsf{f}_{s_1} + \mathsf{f}_{s_2})$$

Two n-sequences are equivalent modulo relocation if they are equal, or if they start at different indices but contain the same sequence of elements.

$$\text{Reloc-Bounds}\ \frac{s_1 = \mathsf{relocate}(s_2, i)}{\begin{array}{l} i = \mathsf{f}_{s_2} \wedge s_1 = s_2 \quad || \\ i \neq \mathsf{f}_{s_2} \wedge \mathsf{f}_{s_1} = i \wedge \mathsf{l}_{s_1} = i + \mathsf{l}_{s_2} - \mathsf{f}_{s_2} \\ \wedge s_1 =_{reloc} s_2 \end{array}}$$

Figure 4.1: The inference rule used to introduce the $=_{reloc}$ relation.

The equivalence modulo relocation relation is used to reason over the relocate function. Figure 4.1 presents the Reloc-Bounds rule, which introduces the $=_{reloc}$ relation. The rule states that when an n-sequence s_2 is relocated to an index i equal to its first index f_{s_2}, the resulting n-sequence s_1 is equal to s_2. Otherwise, when $i \neq \mathsf{f}_{s_2}$ then $s_1 \neq s_2$, the relation $s_1 =_{reloc} s_2$ is created, and the bounds of s_1 are computed from those of s_2 and propagated.

[1]From cvc5's sequence theory:
https://cvc5.github.io/docs-ci/docs-main/theories/sequences.html

Proposition 4.1. *The equivalence modulo relocation relation is an equivalence relation over n-sequences.*

Proof. Proving that $=_{reloc}$ relation is an equivalence relation.

- (a) Reflexivity: Given an n-sequence s, $s =_{reloc} s$ holds since an n-sequence is equal to itself, therefore equivalent (modulo relocation) to itself by definition.
- (b) Symmetry:

  Given two n-sequences s_1 and s_2, $s_1 =_{reloc} s_2$ implies: $\mathsf{l}_{s_2} = \mathsf{l}_{s_1} - \mathsf{f}_{s_1} + \mathsf{f}_{s_2}$ (1) and $\forall i : Int, \mathsf{f}_{s_1} \leq i \leq \mathsf{l}_{s_1} \Rightarrow get(s_1, i) = get(s_2, i - \mathsf{f}_{s_1} + \mathsf{f}_{s_2})$ (2)

  From rearranging (1) we get: $\mathsf{l}_{s_1} = \mathsf{l}_{s_2} - \mathsf{f}_{s_2} + \mathsf{f}_{s_1}$ (3)

  By subtracting $\mathsf{f}_{s_1} - \mathsf{f}_{s_2}$ from the terms of the disequality in (2) we get:

  $\forall i : Int, \mathsf{f}_{s_2} \leq i - \mathsf{f}_{s_1} + \mathsf{f}_{s_2} \leq \mathsf{l}_{s_1} - \mathsf{f}_{s_1} + \mathsf{f}_{s_2} \Rightarrow get(s_1, i) = get(s_2, i - \mathsf{f}_{s_1} + \mathsf{f}_{s_2})$ (4)

  From (2), we can replace $\mathsf{l}_{s_1} - \mathsf{f}_{s_1} + \mathsf{f}_{s_2}$ with l_{s_2} in (4), to get:

  $\forall i : Int, \mathsf{f}_{s_2} \leq i - \mathsf{f}_{s_1} + \mathsf{f}_{s_2} \leq \mathsf{l}_{s_2} \Rightarrow get(s_1, i) = get(s_2, i - \mathsf{f}_{s_1} + \mathsf{f}_{s_2})$ (5)

  if we introduce a variable j such that $j = i - \mathsf{f}_{s_1} + \mathsf{f}_{s_2}$ in (5) we get:

  $\forall j : Int, \mathsf{f}_{s_2} \leq j \leq \mathsf{l}_{s_2} \Rightarrow get(s_1, j - \mathsf{f}_{s_2} + \mathsf{f}_{s_1}) = get(s_2, j)$ (6)

  From (3) and (6), we deduce that $s_2 =_{reloc} s_1$.
- (c) Transitivity:

  Given three n-sequences s_1, s_2 and s_3, $s_1 =_{reloc} s_2$ and $s_2 =_{reloc} s_3$ imply that:

  $\mathsf{l}_{s_2} = \mathsf{l}_{s_1} - \mathsf{f}_{s_1} + \mathsf{f}_{s_2}$ (1)

  $\forall i : Int, \mathsf{f}_{s_1} \leq i \leq \mathsf{l}_{s_1} \Rightarrow get(s_1, i) = get(s_2, i - \mathsf{f}_{s_1} + \mathsf{f}_{s_2})$ (2)

  $\mathsf{l}_{s_3} = \mathsf{l}_{s_2} - \mathsf{f}_{s_2} + \mathsf{f}_{s_3}$ (3)

  $\forall i : Int, \mathsf{f}_{s_2} \leq i \leq \mathsf{l}_{s_2} \Rightarrow get(s_2, i) = get(s_3, i - \mathsf{f}_{s_2} + \mathsf{f}_{s_3})$ (4)

  By replacing l_{s_2} with $\mathsf{l}_{s_1} - \mathsf{f}_{s_1} + \mathsf{f}_{s_2}$ in (3) we get:

  $\mathsf{l}_{s_3} = \mathsf{l}_{s_1} - \mathsf{f}_{s_1} + \mathsf{f}_{s_3}$ (5)

  From (1) we get:

  $\mathsf{l}_{s_1} = \mathsf{l}_{s_2} + \mathsf{f}_{s_1} - \mathsf{f}_{s_2}$ (6)

  By adding $\mathsf{f}_{s_1} - \mathsf{f}_{s_2}$ to the terms of the disequality in (4), we get:

  $\forall i : Int, \mathsf{f}_{s_1} \leq i + \mathsf{f}_{s_1} - \mathsf{f}_{s_2} \leq \mathsf{l}_{s_2} + \mathsf{f}_{s_1} - \mathsf{f}_{s_2} \Rightarrow get(s_2, i) = get(s_3, i - \mathsf{f}_{s_2} + \mathsf{f}_{s_3})$ (7)

  From (6), we can replace $\mathsf{l}_{s_2} + \mathsf{f}_{s_1} - \mathsf{f}_{s_2}$ with l_{s_1} in (7) and get:

  $\forall i : Int, \mathsf{f}_{s_1} \leq i + \mathsf{f}_{s_1} - \mathsf{f}_{s_2} \leq \mathsf{l}_{s_1} \Rightarrow get(s_2, i) = get(s_3, i - \mathsf{f}_{s_2} + \mathsf{f}_{s_3})$ (8)

  By introduction a variable j, such that $i = j - \mathsf{f}_{s_1} + \mathsf{f}_{s_2}$ and replacing i with $j - \mathsf{f}_{s_1} + \mathsf{f}_{s_2}$ in (8) we get:

  $\forall j : Int, \mathsf{f}_{s_1} \leq j \leq \mathsf{l}_{s_1} \Rightarrow get(s_2, j - \mathsf{f}_{s_1} + \mathsf{f}_{s_2}) = get(s_3, j - \mathsf{f}_{s_1} + \mathsf{f}_{s_3})$ (9)

  From (2) we get:

  $\forall j : Int, \mathsf{f}_{s_1} \leq j \leq \mathsf{l}_{s_1} \Rightarrow get(s_1, j) = get(s_3, j - \mathsf{f}_{s_1} + \mathsf{f}_{s_3})$ (10)

  From (5) and (10), we deduce that $s_1 =_{reloc} s_3$.

From (a), (b) and (c), we deduce that $=_{reloc}$ is a reflexive, symmetric and transitive relation, it is therefore an equivalence relation.

□

4.3.2 The common calculus

Definition 4.5 (Internal n-sequence concatenation operator)**.** *For simplicity and consistency with the Seq theory calculi, an internal concatenation operator* :: *is introduced, for which the following invariant holds:*

$$s = s_1 :: s_2 \quad \Longrightarrow \quad \mathsf{f}_s = \mathsf{f}_{s_1} \wedge \mathsf{l}_s = \mathsf{l}_{s_2} \wedge \mathsf{f}_{s_2} = \mathsf{l}_{s_1} + 1.$$

It differs from concat in that it does not require checking the condition $\mathsf{f}_{s_2} = \mathsf{l}_{s_1} + 1$ *before concatenation, as this condition is ensured by the invariant.*

Definition 4.6 (n-sequence normal form)**.** *The internal concatenation operation (Definition 4.5) is used to normalize n-sequences. An n-sequence* s *is associated to a normal form* $s_1[:: \ldots :: s_m]$ *for* $m > 1$*, such that:*

- *The normal form covers the bounds of the n-sequence* s*, i.e.* $\mathsf{f}_{s_1} = \mathsf{f}_s \wedge \mathsf{l}_s = \mathsf{l}_{s_n}$*.*
- *For each component (or subsequence)* s_i *in the normal form,* s *and* s_i *have the same elements at the same indices within the bounds of* s_i*.*

Definition 4.7 (Atomic n-indexed sequence)**.** *An atomic n-sequence is one that either does not have a normal form, or has itself as the only component in its normal form.*

Assumption 4.1. *It is assumed that the following simplification rewrites are applied whenever possible:*

$$\begin{array}{lcll} s_1 :: s_2 & \to & s_1 & \text{when } \mathsf{l}_{s_2} < \mathsf{f}_{s_2} \quad (1) \\ s_1 :: s_2 & \to & s_2 & \text{when } \mathsf{l}_{s_1} < \mathsf{f}_{s_1} \quad (2) \\ s_1 :: s_2 & \to & s_1 :: w_1 :: \ldots :: w_n & \text{when } s_2 = w_1 :: \ldots :: w_n \quad (3) \\ s_1 :: s_2 & \to & w_1 :: \ldots :: w_n :: s_2 & \text{when } s_1 = w_1 :: \ldots :: w_n \quad (4) \end{array}$$

Rules (1) and (2) remove empty n-sequences from the normal form. Rules (3) and (4) ensure that when an n-sequence appears in the normal form of another and has its own normal form, it is replaced by its normal form.

Proposition 4.2. *Assumption 4.1 ensures that when an n-sequence* s *has a normal form* $s_1 :: s_2 :: \cdots :: s_n$*, then the n-sequences* $s_1, \ldots, s_n$ *are atomic n-sequences.*

Proof. The rewrites in Assumption 4.1 ensure that when the normal form of an n-sequence s is no longer atomic (i.e. is of the form $\_ :: \_$), then all the occurrences of s in other n-sequence normal forms are replaced by its normal form. □

Proposition 4.3. *Given an n-sequence* s *that has for a normal form* $s_1 :: s_2$*. Assuming that* s_2 *is empty and the rule (1) of the rewrites in Assumption 4.1 is applied, then the equality* $s = s_1$ *holds.*

Proof. From Definition 4.6:

- An n-sequence s has the same elements at the same indices as s_i within the bounds of s_i, when s_i is a component of the normal form s.
- The normal form of an n-sequence covers the n-sequence's whole bounds.

Then if an n-sequence s, has only one component s_1 in its normal form, then s and s_1 have the same bounds and share the same elements at the same indices. Which is the definition of equality over n-sequences (cf. Definition 4.3), therefore $s = s_1$.

□

Figure 4.2 illustrates a set of common rules shared between the two calculi NS-BASE and NS-EXT. The rule Const-Bounds propagates the bounds of constant n-sequences, which are created using the const function.

The rules NS-Slice, NS-Concat, and NS-Update handle slice, concat, and update by normalizing the n-sequences involved in these functions. For example, with NS-Update, when the n-sequence s is within the bounds of the n-sequence s_2, three fresh n-sequence variables k_1, k_2, and k_3 are introduced, such that the normal form of s_2 is $k_1 :: k_2 :: k_3$, while the normal form of the resulting n-sequence s_1 is $k_1 :: s :: k_3$, ensuring that s_1 is equal to s_2 modified to have the elements of s within the bounds of s.

If an n-sequence s has two normal forms which contain two n-sequence terms y_1 and y_2 that start at the same index. The NS-Split rule ensures that if y_1 and y_2 end at the same index, then they are equal. If they end at different indices, the longer one is propagated as equal to a concatenation of the shorter one and a fresh n-sequence variable. For example, if y_1 is longer than y_2, then the equality $y_1 = y_2 :: k$, with k a fresh n-sequence variable, is propagated.

The NS-Comp-Reloc rule propagates normal forms over the $=_{reloc}$ relation. The NS-Exten rule is the extensionality rule, which states that any two n-sequences s_1 and s_2 are either equal or distinct. Two n-sequences are distinct if they have different bounds, contain distinct components with the same bounds in their normal forms, or differ in at least one element.

4.3.3 The base calculus

The base calculus comprises the rules in Figures 4.1 to 4.3. The rules R-Get and R-Set handle the get and set operations by introducing new normal forms for the n-sequences on which they operate.

In the R-Get rule, when i is within the bounds of s, a new normal form of s is introduced. This form includes a constant n-sequence of size one at index i, storing the value v, and two variables, k_1 and k_2, to represent the left and right segments of the n-sequence s, respectively.

The R-Set rule operates similarly: when i is within the bounds of s_2, new normal forms are introduced for both s_1 and s_2. These forms share the variables k_1 and k_3, which represent the segments to the left and right of index i. For s_1, the normal form contains a constant n-sequence of size one holding the value v at index i, while s_2's normal form contains an n-sequence variable k_2, also of size one at index i.

$$\text{Const-Bounds}\ \frac{s = \mathsf{const}(f, l, v)}{\mathsf{f}_s = f \wedge \mathsf{l}_s = l}$$

$$\text{NS-Slice}\ \frac{s_1 = \mathsf{slice}(s, f, l)}{\begin{array}{l}(f < \mathsf{f}_s \vee l < f \vee \mathsf{l}_s < l) \wedge s_1 = s \quad ||\\ \mathsf{f}_s \leq f \leq l \leq \mathsf{l}_s \wedge \\ \mathsf{f}_{s_1} = f \wedge \mathsf{l}_{s_1} = l \wedge s = k_1 :: s_1 :: k_2\end{array}}$$

$$\text{NS-Concat}\ \frac{s = \mathsf{concat}(s_1, s_2)}{\begin{array}{rl}\mathsf{l}_{s_1} < \mathsf{f}_{s_1} \wedge s = s_2 & ||\\ (\mathsf{l}_{s_2} < \mathsf{f}_{s_2} \vee \mathsf{l}_{s_1} + 1 \neq \mathsf{f}_{s_2}) \wedge s = s_1 & ||\\ \mathsf{f}_{s_1} \leq \mathsf{l}_{s_1} \wedge \mathsf{f}_{s_2} \leq \mathsf{l}_{s_2} \wedge \mathsf{f}_{s_2} = \mathsf{l}_{s_1} + 1 \wedge s = s_1 :: s_2 & \end{array}}$$

$$\text{NS-Update}\ \frac{s_1 = \mathsf{update}(s_2, s)}{\begin{array}{rl}(\mathsf{l}_s < \mathsf{f}_s \vee \mathsf{f}_s < \mathsf{f}_{s_2} \vee \mathsf{l}_{s_2} < \mathsf{l}_s) \wedge s_1 = s_2 & ||\\ \mathsf{f}_{s_2} \leq \mathsf{f}_s \leq \mathsf{l}_s \leq \mathsf{l}_{s_2} \wedge s_1 = k_1 :: s :: k_3 \wedge s_2 = k_1 :: k_2 :: k_3 & \end{array}}$$

$$\text{NS-Split}\ \frac{s = w :: y_1 :: z_1 \qquad s = w :: y_2 :: z_2}{\begin{array}{rl}\mathsf{l}_{y_1} = \mathsf{l}_{y_2} \wedge y_1 = y_2 & ||\\ \mathsf{l}_{y_1} > \mathsf{l}_{y_2} \wedge y_1 = y_2 :: k \wedge \mathsf{f}_k = \mathsf{l}_{y_2} + 1 \wedge \mathsf{l}_k = \mathsf{l}_{y_1} & ||\\ \mathsf{l}_{y_1} < \mathsf{l}_{y_2} \wedge y_2 = y_1 :: k \wedge \mathsf{f}_k = \mathsf{l}_{y_1} + 1 \wedge \mathsf{l}_k = \mathsf{l}_{y_2} & \end{array}}$$

$$\text{NS-Comp-Reloc}\ \frac{s_1 = k_1 :: k_2 :: \ldots :: k_n \qquad s_1 =_{reloc} s_2}{\begin{array}{rl}\mathsf{f}_{s_1} = \mathsf{f}_{s_2} \wedge s_1 = s_2 & ||\\ s_2 = \mathsf{relocate}(k_1, \mathsf{f}_{s_2}) :: \mathsf{relocate}(k_2, \mathsf{f}_{k_2} - \mathsf{f}_{s_1} + \mathsf{f}_{s_2}) :: \ldots & \\ :: \mathsf{relocate}(k_n, \mathsf{f}_{k_n} - \mathsf{f}_{s_1} + \mathsf{f}_{s_2}) & \end{array}}$$

$$\text{NS-Exten}\ \frac{s_1 \qquad s_2}{\begin{array}{rl}s_1 = s_2 & ||\\ \mathsf{f}_{s_1} \neq \mathsf{f}_{s_2} \vee \mathsf{l}_{s_1} \neq \mathsf{l}_{s_2} & ||\\ s_1 = \ldots :: k_1 :: \ldots \wedge s_2 = \ldots :: k_2 :: \ldots \wedge & \\ \mathsf{f}_{k_1} = \mathsf{f}_{k_2} \wedge \mathsf{l}_{k_1} = \mathsf{l}_{k_2} \wedge \mathsf{f}_{k_1} \leq \mathsf{l}_{k_1} \wedge k_1 \neq k_2 & ||\\ \mathsf{f}_{s_1} \leq i \leq \mathsf{l}_{s_2} \wedge \mathsf{get}(s_1, i) \neq \mathsf{get}(s_2, i) & \end{array}}$$

Figure 4.2: Common inference rules for the NS-BASE and NS-EXT calculi.

4.3.4 The extended calculus

The extended calculus (NS-EXT) consists of the rules in Figures 4.1, 4.2, 4.4 and 4.5. It differs from the base calculus by reasoning over the get and set functions in a manner inspired by the weakly equivalent arrays decision procedure [37] (cf. Section 2.5.3).

The rules in Figure 4.4 handle interactions between the get and set functions and n-sequence normal forms. The Get-Concat, Set-Concat, and Set-Concat-Inv rules illustrate how get and set operations are handled when applied to an n-sequence in normal form, where the operations affect the right component of the normal form. Get-Concat simply

$$\text{R-Get}\ \frac{v = \mathsf{get}(s, i)}{\begin{array}{l} i < \mathsf{f}_s \vee \mathsf{l}_s < i \qquad || \\ \mathsf{f}_s \leq i \leq \mathsf{l}_s \wedge s = k_1 :: \mathsf{const}(i, i, v) :: k_2 \end{array}}$$

$$\text{R-Set}\ \frac{s_1 = \mathsf{set}(s_2, i, v)}{\begin{array}{l} (i < \mathsf{f}_{s_2} \vee \mathsf{l}_{s_2} < i) \wedge s_1 = s_2 \qquad || \\ \mathsf{f}_{s_1} = \mathsf{f}_{s_2} \wedge \mathsf{l}_{s_1} = \mathsf{l}_{s_2} \wedge \mathsf{f}_{s_2} \leq i \leq \mathsf{l}_{s_2} \wedge \\ s_1 = k_1 :: \mathsf{const}(i, i, v) :: k_3 \wedge s_2 = k_1 :: k_2 :: k_3 \end{array}}$$

Figure 4.3: NS-BASE-specific inference rules.

propagates the $\mathsf{get}(s, i)$ application to the right n-sequence in the normal form of s by selecting the one that has i within its bounds. The rules Set-Concat and Set-Concat-Inv do that as well. Given $s_1 = \mathsf{set}(s_2, i, v)$, Set-Concat creates a modified version of the normal form of s_1 in which w_i is replaced by $\mathsf{set}(w_i, i, v)$, where w_i is an n-sequence that is part of the normal form of s_2 and that contains i within its bounds. Meanwhile, Set-Concat-Inv does the inverse, by creating a modified version of the normal form of s_2 in which a fresh n-sequence variable k replaces w_i and the equality $w_i = \mathsf{set}(k, i, v)$ is propagated.

Figure 4.5 illustrates reasoning rules of the NS-EXT calculus that are inspired by array reasoning. It focuses on the reasoning over the get and set functions and how they interact with one another, as well as with $=_{reloc}$ relations and const applications.

The Set-Bounds rule ensures that a set application such as $s_1 = \mathsf{set}(s_2, i, v)$ either results in $s_1 = s_2$ if i is not within the bounds of s_2, or else ensures $\mathsf{get}(s_2, i) \neq v$ when s_1 differs from s_2.

The Get-Intro rule introduces a get application derived from a set application when the index of the set application is within the bounds of the n-sequence to which it is applied. The Get-Const rule deals with the case in which get is applied to an n-sequence where all elements are the same value v, ensuring that when get is applied within the bounds of that n-sequence, its result is equal to v.

The Get-Over-Set rule, commonly referred to as the read-over-write or select-over-store rule in decision procedures for the theory of arrays, ensures that a get application over a set application returns the right value. Lastly, the Get-Reloc rule enables the propagation of constraints on elements of an n-sequence to other n-sequences that are equivalent modulo relocation to it.

4.3.5 Soundness Proofs

In this section, the soundness of the inference rules that constitute the NS-BASE and NS-EXT calculi is proven. A rule is said to be sound if, when applied under its premises, it produces an equisatisfiable environment to the one before its application. This is verified by proving that the consequences of the inference rules can in fact be deduced from their premises and the semantics of the operations involved in the rule.

$$\text{Get-Concat}\ \frac{v = \mathsf{get}(s, i) \qquad s = w_1 :: \ldots :: w_n}{\begin{array}{rl} i < \mathsf{f}_s \vee \mathsf{l}_s < i & || \\ \mathsf{f}_{w_1} \leq i \leq \mathsf{l}_{w_1} \wedge \mathsf{get}(w_1, i) = v & || \\ \ldots & || \\ \mathsf{f}_{w_n} \leq i \leq \mathsf{l}_{w_n} \wedge \mathsf{get}(w_n, i) = v & \end{array}}$$

$$\text{Set-Concat}\ \frac{s_1 = \mathsf{set}(s_2, i, v) \qquad s_2 = w_1 :: \ldots :: w_n}{\begin{array}{rl} i < \mathsf{f}_{s_2} \vee \mathsf{l}_{s_2} < i & || \\ s_1 = \mathsf{set}(w_1, i, v) :: \ldots :: w_n \wedge \mathsf{f}_{w_1} \leq i \leq \mathsf{l}_{w_1} & || \\ \ldots & || \\ s_1 = w_1 :: \ldots :: \mathsf{set}(w_n, i, v) \wedge \mathsf{f}_{w_n} \leq i \leq \mathsf{l}_{w_n} & \end{array}}$$

$$\text{Set-Concat-Inv}\ \frac{s_1 = \mathsf{set}(s_2, i, v) \qquad s_1 = w_1 :: \ldots :: w_n}{\begin{array}{rl} i < \mathsf{f}_{s_2} \vee \mathsf{l}_{s_2} < i & || \\ s_2 = k :: \ldots :: w_n \wedge \mathsf{f}_{w_1} \leq i \leq \mathsf{l}_{w_1} \wedge w_1 = \mathsf{set}(k, i, v) & || \\ \ldots & || \\ s_2 = w_1 :: \ldots :: k \wedge \mathsf{f}_{w_n} \leq i \leq \mathsf{l}_{w_n} \wedge w_n = \mathsf{set}(k, i, v) & \end{array}}$$

Figure 4.4: NS-EXT inference rules for reasoning over interactions between set and get applications and normal forms.

$$\text{Set-Bounds}\ \frac{s_1 = \mathsf{set}(s_2, i, v)}{\begin{array}{rl} s_1 = s_2 & || \\ \mathsf{f}_{s_1} = \mathsf{f}_{s_2} \wedge \mathsf{l}_{s_1} = \mathsf{l}_{s_2} \wedge \mathsf{f}_{s_1} \leq i \leq \mathsf{l}_{s_1} \wedge \mathsf{get}(s_2, i) \neq v & \end{array}}$$

$$\text{Get-Intro}\ \frac{s_1 = \mathsf{set}(s_2, i, v)}{i < \mathsf{f}_{s_1} \vee \mathsf{l}_{s_1} < i \quad || \quad \mathsf{f}_{s_1} \leq i \leq \mathsf{l}_{s_1} \wedge v = \mathsf{get}(s_1, i)}$$

$$\text{Get-Const}\ \frac{s = \mathsf{const}(f, l, v) \qquad u = \mathsf{get}(s, i)}{i < \mathsf{f}_s \vee \mathsf{l}_s < i \quad || \quad \mathsf{f}_s \leq i \leq \mathsf{l}_s \wedge u = v}$$

$$\text{Get-Over-Set}\ \frac{s_1 = \mathsf{set}(s_2, i, v) \qquad u = \mathsf{get}(s_1, j)}{\begin{array}{rl} i < \mathsf{f}_{s_1} \vee \mathsf{l}_{s_1} < i \vee j < \mathsf{f}_{s_1} \vee \mathsf{l}_{s_1} < j & || \\ \mathsf{f}_{s_1} \leq i \leq \mathsf{l}_{s_1} \wedge \mathsf{f}_{s_1} \leq j \leq \mathsf{l}_{s_1} \wedge i = j \wedge u = v & || \\ \mathsf{f}_{s_1} \leq i \leq \mathsf{l}_{s_1} \wedge \mathsf{f}_{s_1} \leq j \leq \mathsf{l}_{s_1} \wedge i \neq j \wedge u = \mathsf{get}(s_2, j) & \end{array}}$$

$$\text{Get-Reloc}\ \frac{v = \mathsf{get}(s_1, i) \qquad s_1 =_{reloc} s_2}{i < \mathsf{f}_{s_1} \vee \mathsf{l}_{s_1} < i \quad || \quad \mathsf{f}_{s_1} \leq i \leq \mathsf{l}_{s_1} \wedge v = \mathsf{get}(s_2, i - \mathsf{f}_{s_1} + \mathsf{f}_{s_2})}$$

Figure 4.5: NS-EXT inference rules for reasoning over interactions between set and get applications.

Soundness of the introduction rule of the equivalence modulo relocation relation

Proof. Reloc-Bounds is sound
Given $s_1 = \mathsf{relocate}(s_2, i)$, the rule states:

- if $i = \mathsf{f}_{s_2}$ then $s_1 = s_2$, which is sound by Definition 4.4 (for defining the bounds) and Definition 4.3 (for proving equality).
- otherwise it sets the bounds of s_1 and adds the relation $s_1 =_{reloc} s_2$, which is sound by Definition 4.4.

□

Common calculus soundness

This section proves the soundness of the rules in Figure 4.2.

Proof. Const-Bounds is sound
Given $s = \mathsf{const}(f, l, v)$, the rule simply sets the bounds for the n-sequence s to f and l, following the semantics of the const function. □

Proof. NS-Slice is sound
Given $s_1 = \mathsf{slice}(s, f, l)$, the rule states that if $f < \mathsf{f}_s$ or $l < f$ or $\mathsf{l}_s < l$, then $s_1 = s$. Otherwise, it introduces two fresh n-sequence variables k_1 and k_2 such that $s = k_1 :: s_1 :: k_2$, which amounts to stating that s_1 is equal to the section of the n-sequence s that is within the bounds f and l which are the bounds of s_1, which follows the semantics of the slice function. □

Proof. NS-Concat is sound
Given $s = \mathsf{concat}(s_1, s_2)$, the rule states that if s_1 is empty then $s = s_2$. If s_2 is empty or $\mathsf{f}_{s_2} \neq \mathsf{l}_{s_1} + 1$, then $s = s_1$. Otherwise, $s = s_1 :: s_2$, which corresponds to the semantics of the concat function. □

Proof. NS-Update is sound
Given $s_1 = \mathsf{update}(s_2, s)$, the rule states that if $\mathsf{l}_s < \mathsf{f}_s$ or $\mathsf{f}_s < \mathsf{f}_{s_2}$ or $\mathsf{l}_{s_2} < \mathsf{l}_s$, then $s_1 = s_2$. Otherwise, it introduces three fresh n-sequence variables k_1, k_2, and k_3 such that $s_1 = k_1 :: s :: k_3$ and $s_2 = k_1 :: k_2 :: k_3$, stating that s_1 shares the same elements with s_2 on all indices outside the bounds of s, where s_1 has the same elements as s inside those bounds, while s_2 has the n-sequence fresh variable k_2. This matches the semantics of the update function. □

Proof. NS-Comp-Reloc is sound
Given $s_1 = k_1 :: k_2 :: \ldots :: k_n$ and $s_1 =_{reloc} s_2$, the rule states that if $\mathsf{f}_{s_1} = \mathsf{f}_{s_2}$, then $s_1 = s_2$. Otherwise, it states:

$$s_2 = \mathsf{relocate}(k_1, \mathsf{f}_{s_2}) :: \mathsf{relocate}(k_2, \mathsf{f}_{k_2} - \mathsf{f}_{s_1} + \mathsf{f}_{s_2}) :: \ldots :: \mathsf{relocate}(k_n, \mathsf{f}_{k_n} - \mathsf{f}_{s_1} + \mathsf{f}_{s_2})$$

which corresponds to computing a normal form for s_2 by relocating that of s_1, and is sound by Definitions 4.4 and 4.6. □

Proof. NS-Exten is sound

Given two n-sequences s_1 and s_2, the rule states that they are either equal or unequal, with inequality presented in three cases:

- the two n-sequences have different bounds.
- their normal forms contain distinct components with the same bounds.
- there exists an index within the bounds of both n-sequences where they hold different elements.

This follows directly from the extensionality property in Definition 4.3. □

NS-BASE soundness

This section proves the soundness of the rules in Figure 4.3.

Proof. R-Get is sound

Given $v = \mathsf{get}(s, i)$, the rule does nothing if i is outside the bounds of s. Otherwise, it states that $s = k_1 :: \mathsf{const}(i, i, v) :: k_2$, with k_1 and k_2 as fresh n-sequence variables. This amounts to stating that the element at the ith index in s is equal to v, which corresponds to the semantics of the get function. □

Proof. R-Set is sound

Given $s_1 = \mathsf{set}(s_2, i, v)$, the rule states:

- If i is within the bounds of s_2, then fresh n-sequence variables k_1, k_2, and k_3 are introduced, and $s_1 = k_1 :: \mathsf{const}(i, i, v) :: k_3$ and $s_2 = k_1 :: k_2 :: k_3$ are propagated. This states that at index i, s_1 contains $\mathsf{const}(i, i, v)$, while s_2 contains k_2, and on the other indices, s_1 and s_2 share the same elements, represented by k_1 and k_3. This is sound by the semantics of the set function and Definition 4.6.
- If i is outside the bounds of s_2, then $s_1 = s_2$, which is sound by the semantics of the set function.

□

NS-EXT soundness

This section proves the soundness of the rules in Figures 4.4 and 4.5.

Proof. Get-Concat is sound

Given $v = \mathsf{get}(s, i)$ and $s = w_1 :: \ldots :: w_n$, the rule does nothing if i is outside the bounds of s. Otherwise, it states that $\mathsf{get}(w_m, i) = v$ for some $1 \leq m \leq n$, where w_m is the component of s's normal form encompassing the index i. This amounts to applying the get function to the component of s's normal form that encompasses index i, which is sound by the semantics of the get function and Definition 4.6. □

Proof. Set-Concat is sound

Given $s_1 = \mathsf{set}(s_2, i, v)$ and $s_2 = w_1 :: \ldots :: w_n$, the rule does nothing if i is outside the bounds of s_2. Otherwise, it sets the normal form of s_1 to be the same as s_2 except for component w_m, where $\mathsf{f}_{w_m} \leq i \leq \mathsf{l}_{w_m}$, replacing w_m with $\mathsf{set}(w_m, i, v)$. This amounts to applying the set function to the component of s_2's normal form that encompasses index i, which is sound by the semantics of set and Definition 4.6. □

Proof. Set-Concat-Inv is sound

Given $s_1 = \mathsf{set}(s_2, i, v)$ and $s_1 = w_1 :: \ldots :: w_n$, the rule does nothing if i is outside the bounds of s_2. Otherwise, it sets the normal form of s_2 to be the same as s_1 except for component w_m, where $\mathsf{f}_{w_m} \leq i \leq \mathsf{l}_{w_m}$, replacing w_m with a fresh n-sequence variable k such that $w_m = \mathsf{set}(k, i, v)$. This matches the semantics of set and Definition 4.6. □

Proof. Set-Bounds is sound

Given $s_1 = \mathsf{set}(s_2, i, v)$, the rule states:

- Either $s_1 = s_2$, because i is outside the bounds of s_2 or because $v = \mathsf{get}(s_2, i)$, which is sound by the semantics of set and get.
- Or i is within the bounds of s_2, s_1 and s_2 have equal bounds, and $v \neq \mathsf{get}(s_2, i)$, which is also sound by the semantics of set and get.

□

Proof. Get-Intro is sound

Given $s_1 = \mathsf{set}(s_2, i, v)$, the rule states that if i is within the bounds of s_2, then $v = \mathsf{get}(s_1, i)$. Otherwise, the rule does nothing, which is sound by the semantics of get and set. □

Proof. Get-Const is sound

Given $s = \mathsf{const}(f, l, v)$ and $u = \mathsf{get}(s, i)$, if i is within the bounds of s, then $u = v$ since s is a constant n-sequence, otherwise the rule does nothing. This is sound by the semantics of get and const. □

Proof. Get-Over-Set is sound

Given $s_1 = \mathsf{set}(s_2, i, v)$ and $u = \mathsf{get}(s_1, j)$, if i is outside the bounds of s_1, then the rule does nothing. Otherwise:

- If $i = j$, then $u = v$, which is sound by the semantics of get and set.
- If $i \neq j$, then $u = \mathsf{get}(s_2, j)$, which is also sound by the semantics of get and set.

□

Proof. Get-Reloc is sound

Given $v = \mathsf{get}(s_1, i)$ and $s_1 =_{reloc} s_2$, the rule does nothing if i is not within the bounds of s_1. Otherwise, it states that $v = \mathsf{get}(s_2, i - \mathsf{f}_{s_1} + \mathsf{f}_{s_2})$, which is sound by Definition 4.4. □

4.4 Reasoning with Shared Slices

The shared slices calculus, denoted as NS-ShS, was developed to reason over the theory of n-indexed sequences, but it can also be adapted for classical sequences. Contrary to the calculi presented in Section 4.3, this one is not inspired by string reasoning and therefore does not rely on word equations to reason over (n-indexed) sequences. Instead, it is based on the array decision procedure that relies on weak equivalence between arrays introduced by Christ and Hoenicke [37]. It uses the weak-equivalence graph and extends it with other relations that arise when reasoning over n-indexed sequences, notably equivalence modulo relocation and the shared-slice relation.

Definition 4.8 (Shared-Slice Relation)**.** *Given two n-sequences a and b, a shared slice between a and b is a relation denoted as $=_{[f;l]}$, where f and l are bounds such that $f \leq l$, and both are within the bounds of a and b. The relation $a =_{[f;l]} b$ states that a and b share a slice within the bounds f and l, and is defined as follows:*

$$\begin{gathered} a =_{[f;l]} b \equiv \\ f_a \leq f \leq l \leq l_a \wedge \\ f_b \leq f \leq l \leq l_b \wedge \\ \forall i.\ f \leq i \leq l \implies \mathit{get}(a, i) = \mathit{get}(b, i) \end{gathered}$$

Proposition 4.4. *The shared-slice relation is a family of equivalence relations.*

Proof. Given three n-sequences a, b, and c, and any two bounds f and l such that $f \leq l$ and the bounds $[f; l]$ are within the bounds of a, b, and c, the shared-slice relation $=_{[f;l]}$ satisfies:

- Reflexivity: $a =_{[f;l]} a$, since a has the same elements as itself on all indices, and thus also within the bounds $[f; l]$.
- Symmetry: The relation $a =_{[f;l]} b$ is undirected. It simply states that a and b share the same elements within the bounds $[f; l]$. Therefore, $b =_{[f;l]} a$ also holds.
- Transitivity: Given $a =_{[f;l]} b$ and $b =_{[f;l]} c$, the two relations respectively state that a and b share the same elements within the bounds $[f; l]$, and that b and c share the same elements within the same bounds. Therefore, a and c also share the same elements within the bounds $[f; l]$, which implies $a =_{[f;l]} c$.

Since the shared-slice relation is reflexive, symmetric, and transitive for given bounds, it is therefore a family of equivalence relations. □

This section presents the shared slices calculus in two parts. The first part describes how the relations graph is created. The relations graph manages the relations that are used by the calculus to reason over n-indexed sequences, notably the shared-slice and weak-equivalence relations. The second part presents the calculus in the form of the inference rules that it consists of. These rules do different propagations between n-indexed sequences and handle the detection of equalities and inequalities.

4.4.1 Relations Graph

The relations graph, used in the shared slices calculus, is an extended version of the weak equivalence graph [37] (cf. Section 2.5.3). In this graph, vertices are n-indexed sequences.

Definition 4.9 (Weak-Equivalence Edge on n-Indexed Sequences)**.** *Given two n-indexed sequences a and b, the existence of a weak equivalence edge labeled with an index k linking a and b, denoted $a \overset{k}{\leftrightarrow} b$, represents:*

$$\mathsf{f}_a = \mathsf{f}_b \wedge \mathsf{l}_a = \mathsf{l}_b \wedge \forall i.\ (\mathsf{f}_a \leq i \leq \mathsf{l}_a \wedge i \neq k) \implies \mathsf{get}(a, i) = \mathsf{get}(b, i).$$

This means that for all indices within the bounds of a (and b) that are different from k, a and b hold the same elements. If a and b are different, they can only differ on the element they hold at the index k.

Two n-indexed sequences a and b can be linked by a weak-equivalence or shared slice edge. They can also be in relation with one another through the equivalence modulo relocation relation described in Definition 4.4. This relation is represented using a labeled union-find structure following the star topology.

An undirected weak-equivalence edge labeled with the index term i is added between two n-sequences a and b whenever a term of the form $a = \mathsf{set}(b, i, \_)$ or $b = \mathsf{set}(a, i, \_)$ is created, such that $\mathsf{f}_a \leq i \leq \mathsf{l}_a$ (and $\mathsf{f}_b \leq i \leq \mathsf{l}_b$, knowing that $\mathsf{f}_a = \mathsf{f}_b$ and $\mathsf{l}_a = \mathsf{l}_b$). There can only be one weak-equivalence edge between two n-sequences a and b.

Proposition 4.5. *If two n-sequences a and b are already linked by a weak-equivalence edge labeled with the index term j, and a term of the form $a = \mathsf{set}(b, i, \_)$ is created with $\mathsf{f}_b \leq i \leq \mathsf{l}_b$, then, knowing that $\mathsf{f}_a = \mathsf{f}_b$ and $\mathsf{l}_a = \mathsf{l}_b$, it follows that either $i = j$ or $a = b$.*

Proof. Given two n-sequences a and b such that an edge $a \overset{j}{\leftrightarrow} b$ exists, if a second edge $a \overset{i}{\leftrightarrow} b$ is introduced:

- If $i = j$ then the edge $a \overset{j}{\leftrightarrow} b$ is kept and nothing changes.
- If $i \neq j$, then $a = b$:

  From the definition of weak-equivalence edges (Definition 4.9):

  (a) The existence of $a \overset{j}{\leftrightarrow} b$ means that a and b have the same elements in all indices except possibly in j.

  (b) Adding $a \overset{i}{\leftrightarrow} b$ means that a and b have the same elements in all indices, including j, except possibly in i.

From (a) the equality $\mathsf{get}(a, i) = \mathsf{get}(b, i)$ can be deduced, combined with the edge $a \overset{i}{\leftrightarrow} b$, it leads to deducing $a = b$ (and symmetrically from (b) with $a \overset{j}{\leftrightarrow} b$). □

The construction of the relations graph is done through the rules in Figure 4.6. The rule Set-Bounds-WEq introduces weak-equivalence relations from applications of the set function. Shared slices are introduced from applications of the functions concat, slice, and update, through the rules Concat-ShS-Intro, Slice-ShS-Intro, and Update-ShS-Intro, respectively. The equivalence modulo relocation relation is introduced via the same Reloc-Bounds rule shown in Figure 4.1.

$$\text{Set-Bounds-WEq}\ \frac{s_1 = \mathsf{set}(s_2, i, v)}{\begin{array}{l} s_1 = s_2 \quad || \\ \mathsf{f}_{s_1} \leq i \leq \mathsf{l}_{s_1} \wedge \mathsf{f}_{s_1} = \mathsf{f}_{s_2} \wedge \mathsf{l}_{s_1} = \mathsf{l}_{s_2} \wedge \mathsf{get}(s_1, i) = v \wedge s_1 \overset{i}{\leftrightarrow} s_2 \end{array}}$$

$$\text{Concat-ShS-Intro}\ \frac{a = \mathsf{concat}(b, c)}{\begin{array}{l} \mathsf{f}_b > \mathsf{l}_b \wedge a = c \quad || \\ \mathsf{f}_c > \mathsf{l}_c \wedge a = b \quad || \\ \mathsf{l}_b + 1 \neq \mathsf{f}_c \wedge a = b \quad || \\ \mathsf{f}_b \leq \mathsf{l}_b \wedge \mathsf{f}_c \leq \mathsf{l}_c \wedge \mathsf{l}_b + 1 = \mathsf{f}_c \wedge a =_{[\mathsf{f}_b;\mathsf{l}_b]} b \wedge a =_{[\mathsf{f}_c;\mathsf{l}_c]} c \end{array}}$$

$$\text{Slice-ShS-Intro}\ \frac{a = \mathsf{slice}(b, f, l)}{\begin{array}{l} (\mathsf{f}_b > \mathsf{l}_b \vee f > l \vee \mathsf{f}_b > f \vee l > \mathsf{l}_b) \wedge a = b \quad || \\ \mathsf{f}_b \leq f \leq l \leq \mathsf{l}_b \wedge \mathsf{f}_a = f \wedge \mathsf{l}_a = l \wedge a =_{[f;l]} b \end{array}}$$

$$\text{Update-ShS-Intro}\ \frac{a = \mathsf{update}(b, c)}{\begin{array}{l} (\mathsf{f}_c < \mathsf{f}_b \vee \mathsf{l}_b < \mathsf{l}_c \vee \mathsf{f}_c > \mathsf{l}_c) \wedge a = b \quad || \\ \mathsf{f}_b = \mathsf{f}_c \wedge \mathsf{l}_c = \mathsf{l}_b \wedge a = c \quad || \\ \mathsf{f}_a = \mathsf{f}_b \wedge \mathsf{l}_a = \mathsf{l}_b \wedge \mathsf{f}_b \leq \mathsf{f}_c \leq \mathsf{l}_c \leq \mathsf{l}_b \wedge \\ a =_{[\mathsf{f}_b;\mathsf{f}_c-1]} b \wedge a =_{[\mathsf{f}_c;\mathsf{l}_c]} c \wedge a =_{[\mathsf{l}_c+1;\mathsf{l}_b]} b \end{array}}$$

Figure 4.6: Rules that introduce the shared-slice and weak-equivalence relations.

4.4.2 Calculus

The shared slices calculus works by building the relations graph via the rules in Figures 4.1 and 4.6 for the introduction of the weak-equivalence, shared slice, and equivalence modulo relocation relations.

In addition to the graph construction rules, the shared slices reasoning relies on the rules in Figure 4.7, as well as the rule Const-Bounds in Figure 4.2 and the rules Get-Const and Get-Reloc in Figure 4.5.

Figure 4.7 presents rules that are specific to the shared slices calculus. In this calculus, element constraints are propagated over weak-equivalence relations with the rule Get-Over-WEq, which allows propagating an element constraint over a weak-equivalence edge when possible. The Get-Over-ShS rule describes how constraints on n-sequence elements are propagated over shared slices. The rule Match-ShS states that if two n-sequences b and c have a shared slice, eventually transitively through an intermediary n-sequence a, and this slice covers their whole bounds, then it follows that $b = c$.

Finally, the extensionality rule Shs-Ext states that for any two n-sequences a and b, these n-sequences are equal if they have the same bounds and the same elements. Otherwise, they are distinct. In the case of inequality, the index term k is a fresh index variable within the bounds of a and b, chosen such that $\mathsf{get}(a, k)$ and $\mathsf{get}(b, k)$ differ and no shared slice exists covering k.

$$\text{Get-Over-WEq}\ \frac{\mathsf{get}(a,i) = v \qquad a \overset{k}{\leftrightarrow} b}{\begin{array}{c} i < \mathsf{f}_a \vee i > \mathsf{l}_a \qquad || \\ i = k \qquad || \\ \mathsf{f}_a \leq i \leq \mathsf{l}_a \wedge i \neq k \wedge \mathsf{get}(b,i) = v \end{array}}$$

$$\text{Get-Over-ShS}\ \frac{v = \mathsf{get}(a,i) \qquad a =_{[f;l]} b}{\begin{array}{c} i < f \vee i > l \qquad || \\ f \leq i \leq l \wedge \mathsf{get}(b,i) = v \end{array}}$$

$$\text{Match-ShS}\ \frac{a =_{[\mathsf{f}_b;\mathsf{l}_b]} b \qquad a =_{[\mathsf{f}_c;\mathsf{l}_c]} c}{\begin{array}{c} \mathsf{f}_b \neq \mathsf{f}_c \vee \mathsf{l}_b \neq \mathsf{l}_c \qquad || \\ \mathsf{f}_b = \mathsf{f}_c \wedge \mathsf{l}_b = \mathsf{l}_c \wedge b = c \end{array}}$$

$$\text{Shs-Ext}\ \frac{a \qquad b}{\begin{array}{c} (\mathsf{f}_a \neq \mathsf{f}_b \vee \mathsf{l}_a \neq \mathsf{l}_b) \wedge a \neq b \qquad || \\ \mathsf{f}_a = \mathsf{f}_b \wedge \mathsf{l}_a = \mathsf{l}_b \wedge a = b \qquad || \\ \mathsf{f}_a = \mathsf{f}_b \wedge \mathsf{l}_a = \mathsf{l}_b \wedge \mathsf{f}_a \leq k \leq \mathsf{l}_a \wedge a \neq b \wedge \\ \mathsf{get}(a,k) \neq \mathsf{get}(b,k) \wedge \\ \forall f, l.\ a =_{[f;l]} b \implies k < f \vee k > l \end{array}}$$

Figure 4.7: Rules from the shared slices calculus that handle the propagation of element constraints over shared slice and weak-equivalence relations, as well as the interactions of shared slices with one another, in addition to an adapted version of the extensionality rule for shared slices.

Discussion

While not yet implemented, further simplifications based on interactions between relations can be envisioned. For example, given two n-sequences a and b that share two shared slices: $a =_{[f_1;l_1]} b$ and $a =_{[f_2;l_2]} b$. Instead of keeping these two as separate relations, they can be combined into a single union-of-slices relation:

$$a =_{[f_1;l_1]\cup[f_2;l_2]} b.$$

This allows simplifications such as merging adjacent or overlapping intervals. For instance, if $f_1 \leq f_2 \leq l_1 \leq l_2$, then the union can be simplified to $a =_{[f_1;l_2]} b$.

Another potential simplification involves the interaction between shared slices and weak-equivalence edges. If a weak-equivalence edge exists between two n-sequences a and b: $a \overset{k}{\leftrightarrow} b$, and a shared slice edge $a =_{[f;l]} b$ also exists such that $f \leq k \leq l$, then a and b can be merged. Indeed, k being within the shared slice $[f;l]$ implies that a and b hold the same value at index k. Given that weak-equivalence edges indicate that a and b can differ only at k, it follows that they are therefore equal.

4.4.3 Soundness Proofs

Similarly to Section 4.3.5, this section proves the soundness of the inference rules used by the shared slices calculus.

Soundness Proofs for the Shared Slice and Weak-Equivalence Relation Introduction Rules

Proof. Set-Bounds-WEq is sound

Given $s_1 = \mathsf{set}(s_2, i, v)$, the rule states that:

- If i is not within the bounds of s_2, then $s_1 = s_2$.
- Otherwise, the bounds of s_1 and s_2 are equal and a weak-equivalence edge is introduced, $s_1 \overset{i}{\leftrightarrow} s_2$, which states that for all indices except possibly i, the n-sequences s_1 and s_2 have the same elements.

This is sound, as it corresponds to the semantics of the set function and the definition of weak equivalence in Definition 4.9. □

Proof. Concat-ShS-Intro is sound

Given $a = \mathsf{concat}(b, c)$, the rule reproduces the semantics of concat:

- If $\mathsf{f}_b > \mathsf{l}_b$, then $a = c$.
- If $\mathsf{f}_c > \mathsf{l}_c$, then $a = b$.
- If $\mathsf{l}_b + 1 \neq \mathsf{f}_c$, then $a = b$.
- Otherwise, if b and c are both non-empty and $\mathsf{l}_b + 1 = \mathsf{f}_c$, then two shared slice relations are introduced:

  $$a =_{[\mathsf{f}_b;\mathsf{l}_b]} b \wedge a =_{[\mathsf{f}_c;\mathsf{l}_c]} c,$$

  stating that a has the same elements as b within the bounds of b, and the same elements as c within the bounds of c.

This is sound with respect to the semantics of concat. □

Proof. Slice-ShS-Intro is sound

Given $a = \mathsf{slice}(b, f, l)$, the rule states that if $\mathsf{f}_b \leq f \leq l \leq \mathsf{l}_b$, then a shared slice relation is introduced:

$$a =_{[f;l]} b$$

meaning that a has the same elements as b within the bounds $[f; l]$. Otherwise, $a = b$.

This is sound with respect to the semantics of slice. □

Proof. Update-ShS-Intro is sound

Given $a = \mathsf{update}(b, c)$, the rule states that if c is empty or its bounds are not contained within the bounds of b, then $a = b$. Otherwise:

- If c's bounds are equal to the bounds of b, then $a = c$.
- Otherwise, three shared slices are introduced:
  - $a =_{[\mathsf{f}_c;\mathsf{l}_c]} c$, stating that a has the same elements as c within the bounds of c.
  - $a =_{[\mathsf{f}_b;\mathsf{f}_c-1]} b$ and $a =_{[\mathsf{l}_c+1;\mathsf{l}_b]} b$, stating that a has the same elements as b within its bounds, except inside the bounds of c.

This is sound by the semantics of update. □

Soundness Proofs for the Shared Slice Reasoning Rules

Proof. Get-Over-WEq is sound

Given $v = \mathsf{get}(a, i)$ and $a \overset{k}{\leftrightarrow} b$, the rule states that if i is within the bounds of a and $i \neq k$, then $\mathsf{get}(b, i) = v$. This is sound by the semantics of weak equivalence, as defined in Definition 4.9, since $a \overset{k}{\leftrightarrow} b$ means a and b hold the same elements in all indices except possibly k. □

Proof. Get-Over-ShS is sound

Given $v = \mathsf{get}(a, i)$ and $a =_{[f;l]} b$, the rule states that if i is within the bounds of a and within the bounds $[f; l]$, then $\mathsf{get}(b, i) = v$. This is sound by the semantics of shared slices (cf. Definition 4.8). □

Proof. Match-ShS is sound

Given $a =_{[\mathsf{f}_b;\mathsf{l}_b]} b$ and $a =_{[\mathsf{f}_c;\mathsf{l}_c]} c$, the rule states that if $\mathsf{f}_b = \mathsf{f}_c$ and $\mathsf{l}_b = \mathsf{l}_c$, i.e. b and c have, possibly transitively, a shared slice covering their entire bounds, then $b = c$. This follows from the definition of equality over n-sequences in Definition 4.3 and is sound by the semantics of shared slices. □

Proof. Shs-Ext is sound

Given any two n-sequences a and b, the rule states that either:

- They have different bounds, therefore $a \neq b$.
- They have the same bounds and are equal.
- They have the same bounds, but there exists an index k such that:

– k is within the bounds of a (and b since the bounds of a and b are equal),
– k is not within the bounds of any shared slice between a and b,
– and $\mathsf{get}(a, k) \neq \mathsf{get}(b, k)$.

This rule essentially adapts the extensionality rule to take into account shared slices and is therefore sound with respect to the semantics of shared slices (in Definition 4.8) and the definition of equality over n-sequences (in Definition 4.3).

□

Chapter 5

n-Indexed Sequences II: Implementation and Evaluation

This chapter is composed of two sections. The first presents and discusses the implementations of the calculi that were developed for the theory of n-indexed sequences as described in Chapter 4. The second section focuses on the experimental evaluation of these calculi. It explains how benchmarks were produced and selected, and how the implementations were compared with those for the theory of sequences in other state-of-the-art SMT solvers such as cvc5 and Z3.

5.1 Implementation

To evaluate the performance of the calculi described in Chapter 4, they were implemented in Colibri2. This section discusses various implementation choices as well as how the CP features of Colibri2 were used to implement the inference rules that constitute the calculi presented in Chapter 4. This section also describes several heuristics that are used to improve the performance of the reasoning.

5.1.1 n-Indexed sequence Normal Forms

The n-sequence normal forms (defined in Definition 4.6) are implemented as a domain in Colibri2.

Definition 5.1 (The model of an n-indexed sequence)**.** *Given an n-sequence s, its model is defined as:*

$$\{i \in \mathbb{Z} \mid f_s \leq i \leq l_s\} \to A$$

which is a map from indices within its bounds to the values of the elements at those indices.

The domain of normal forms is denoted as $\mathcal{D}_{NF}$, and it is defined as follows:

- $\mathcal{R}_{NF}$: A map of indices to another map of indices to atomic n-sequences of sort $\mathsf{Node.t} \to (\mathsf{Node.t} \to \mathsf{Node.t})$, denoted as:

$$\{\, f_i \mapsto \{\, l_i \mapsto n_i \}\,\},$$

where each triplet (f_i, l_i, n_i) represents an atomic n-sequence n_i that is a component of the normal form and has for bounds f_i and l_i.

Since these triplets correspond to n-sequences that appear in normal forms (each with a first and last bound), therefore it is not possible to have a binding $\{f \to \emptyset\}$ in $\mathcal{R}_{NF}$.

- $P_{\mathcal{A}} : (r : \mathcal{R}_{NF}) \to 2^A$
  During model generation, after all decisions have been made, r is necessarily normalized thanks to the NS-Split rule such that no further splitting can be done on atomic n-sequences.

  Its semantics are defined as:
  $$P_{\mathcal{A}}(r) = \bigcup_i s_i^{\mathcal{A}},$$
  where $(\_, \_, s_i) \in r$, $s_i^{\mathcal{A}}$ denotes the value of the atomic n-sequence s_i under the model $\mathcal{A}$, and the union $\bigcup_i s_i^{\mathcal{A}}$ is the sequence obtained by concatenating the $s_i^{\mathcal{A}}$ n-sequences in ascending order of their first indices.

- equal : $(r_1 : \mathcal{R}_{NF}) \to (r_2 : \mathcal{R}_{NF}) \to \mathsf{Bool}$
  Checks whether:
  $$\forall (f_i, l_i, n_i) \in r_1,\ \exists (f_j, l_j, n_j) \in r_2.\ f_i = f_j \wedge l_i = l_j \wedge n_i = n_j,$$
  and conversely for all triplets in r_2 with respect to r_1.

- inter : $(r_1 : \mathcal{R}_{NF}) \to (r_2 : \mathcal{R}_{NF}) \to \mathcal{R}_{NF}$ option
  From the definition of normal forms (Definition 4.6) and given that inter is called during the merge of two nodes whose bounds have already been merged, the smallest and biggest bounds used as keys in r_1 and r_2 are the same.

  The reconciliation of normal forms represented by r_1 and r_2 is done by applying the NS-Split rule. Since splits are not applied right away, a decision is simply registered for the splitting of the normal forms, and the returned domain by inter will be in an intermediary state until the decision is actually applied and the real post-splitting normal form is set.

  Concretely, the contents of r_2 are simply added to r_1. For any tuple $(f, l, n) \in r_2$:

  - If $f \in r_1$ and $l \in r_1[f]$, then $n = r_1[f][l]$.
  - If $f \in r_1$ but $l \notin r_1[f]$, then there exists some $(f, l', n') \in r_1$ with $l \neq l'$. Decisions and splits from NS-Split are then registered:
    * If $l = l'$, then $n = n'$.
    * If $l < l'$, then the normal form of n' is set to $n :: k$, with k a fresh n-sequence variable.
    * If $l > l'$, then the normal form of n is set to $n' :: k$, with k a fresh n-sequence variable.

    Combined with the simplification rewrites in Section 5.1.2. When the decisions from the NS-Split rule are applied and an n-sequence component is eventually replaced by its normal form, new decisions can be introduced and that continues until a flat normal form is found.

Example 5.1. *Given the n-sequence terms s_1, s_2, a, b, c and d:*

- *s_1 has the normal form $a :: b$, represented by the domain r_1:*

$$r_1 = \{1 \mapsto \{5 \mapsto a\},\ 6 \mapsto \{10 \mapsto b\}\}.$$

- *s_2 has the normal form $c :: d$, represented by the domain r_2:*

$$r_2 = \{1 \mapsto \{i \mapsto c\},\ (i+1) \mapsto \{10 \mapsto d\}\}.$$

where $1 \leq i \leq 10$.

When merging s_1 and s_2, a new normal form domain is created:

$$r = \{1 \mapsto \{5 \mapsto a,\ i \mapsto c\},\ 6 \mapsto \{10 \mapsto b\},\ (i+1) \mapsto \{10 \mapsto d\}\}.$$

And a decision is registered on the relationship between 5 and i.

If the decision $i < 5$ is made, then $a = c :: e$ is propagated, where e is a fresh variable that has the bounds $[i+1, 5]$. The normal form domain r is updated to:

$$r = \{1 \mapsto \{i \mapsto c\},\ (i+1) \mapsto \{5 \mapsto e,\ 10 \mapsto d\},\ 6 \mapsto \{10 \mapsto b\}\}.$$

The rule NS-Split is then applied to the binding: $\{i+1 \mapsto \{5 \mapsto e,\ 10 \mapsto d\}\}$, resulting in $d = e :: f$, where the bounds of f are $[6, 10]$.

The final normal form domain r becomes:

$$r = \{1 \mapsto \{i \mapsto c\},\ (i+1) \mapsto \{5 \mapsto e\},\ 6 \mapsto \{10 \mapsto b\}\}.$$

A propagation $f = b$ is also made during substitution (cf. Section 5.1.2), since f has the same bounds as b.

5.1.2 Simplification rewrites

To ensure that Assumption 4.1 is maintained, Colibri2's daemon system is used. For Assumption 4.1, daemons are defined on the following events for a given n-sequence s:

- s is deduced to be empty (i.e. $\mathsf{l}_s < \mathsf{f}_s$ is determined to be true): all occurrences of s in the normal forms of other n-sequences are removed.
- s is determined no longer to be an atomic n-sequence (i.e. s its normal form domain is set to something like $r = \_ :: \_$): all occurrences of s in the normal forms of other n-sequences are replaced by the normal form r.

Whenever a simplification is applied to a given n-sequence s, it is also applied to all n-sequences that belong to the same $=_{reloc}$ class as s. For example, if an n-sequence s is determined to be empty, it is removed from all normal forms of other n-sequences in which it occurs. Similarly, all elements of the same $=_{reloc}$ class as s are removed from any normal forms where they appear. This ensures that Assumption 4.1 is consistently upheld.

5.1.3 Equivalence modulo relocation

As described in Section 4.3.1, the $=_{reloc}$ relation links pairs of n-sequences that contain the same sequence of elements but have different starting indices. The $=_{reloc}$ relation is an equivalence relation, as shown in Proposition 4.1. All n-sequence terms that are equivalent modulo relocation (possibly transitively) are said to belong to the same $=_{reloc}$ class.

Equivalence modulo relocation is necessary for propagating constraints between n-indexed sequences that are equivalent modulo relocation. These include constraints over the bounds of n-indexed sequences, equality propagations with the Reloc-Bounds rule, and maintaining the invariants of the equivalence modulo relocation relation on n-sequence normal forms with the NS-Comp-Reloc rule in Figure 4.2. It is also required for propagating constraints on n-sequence elements with the Get-Reloc rule in Figure 4.5.

To perform these various propagations and deductions efficiently, it is necessary to be able to easily retrieve the $=_{reloc}$ class of any given n-sequence.

This section presents three different ways to represent the $=_{reloc}$ relation to be able to reason over it and with it

Undirected Graph

A straightforward way to represent the $=_{reloc}$ relation is to use an undirected graph in which the vertices represent n-sequences. An edge between two vertices indicates that the n-sequences are equivalent modulo relocation. The graph is undirected because the $=_{reloc}$ relation is an equivalence relation (cf. Proposition 4.1).

In this approach, constraint propagation can be performed via graph exploration algorithms such as Breadth-First Search (BFS) to find all elements of the $=_{reloc}$ class of a given n-sequence. Alternatively, an auxiliary data structure that explicitly maps each n-sequence to all the other members of its $=_{reloc}$ class can be used.

However, both of these approaches can be costly:

- Graph exploration can be computationally expensive in time complexity, especially for large graphs, when used frequently.
- Maintaining auxiliary data structures mapping each n-sequence to its equivalence class can have a significant memory cost and time most when it is updated and maintained.

In addition, other features such as equality detection between two n-sequences at the same distance from another n-sequence are not straightforward in the graph approach. Such actions may require either additional auxiliary data structures or repeated graph traversals.

Union-Find

Another approach is to use the union-find data structure (cf. Section 3.2.1). The union-find structure efficiently joins objects related by an equivalence relation into sets, each set having a single representative element. When a relation is created between two elements from different sets, the sets are merged.

For equivalence modulo relocation, the elements of the union-find data structure are n-indexed sequences, and the equivalence relation is the $=_{reloc}$ relation. Union-find sets are represented as rooted trees in which the representative element is at the root, and non-representative elements are internal nodes or leaves.

The union-find data structure is better than an undirected graph as it does not require a full exploration of the tree to determine whether two n-sequences are at the same index. Instead, a traversal from each n-sequence to its representative, computing the cumulative distance difference along the way is sufficient. If the difference in distance between the two paths is equal, the n-sequences are equal. Furthermore, if the distance of a n-sequence relative to the representative is zero, then that n-sequence equals the representative.

However, this approach is still suboptimal: computing paths to the representative can still be costly. Ideally, this information should be readily available without traversal, since it can known at the creation of each $=_{reloc}$ relation.

Labeled Union-Find

A more suitable approach uses the labeled union-find data structure (cf. Section 3.2.2).

For the equivalence modulo relocation relation, the set of nodes $\mathbb{N}$ in the labeled union-find corresponds to n-sequence terms, while the set of labels $\mathbb{L}$ corresponds to integer polynomials that represent the differences in starting indices between n-sequence terms. Integer polynomials with addition satisfy the group axioms:

- Composition is integer addition: $\mathrm{comp}(x, y) = x + y$.
- The neutral element is 0: $\forall x.\ x + 0 = 0 + x = x$.
- The inverse function is negation: $\mathrm{inv}(x) = -x$.

The used variation of the labeled union-find is the one that follows the star topology, meaning that all paths are compressed so that non-representative terms in a $=_{reloc}$ class all have a direct edge to the representative. This topology is chosen because it simplifies equality detection.

Example 5.2. *Consider the following formulas:*

- F_1: $s_1 = \mathsf{relocate}(s, k_1)$
- F_2: $s_2 = \mathsf{relocate}(s, k_2)$
- F_3: $s_3 = \mathsf{relocate}(s_2, k_3)$
- F_4: $v_1 \neq v_2$
- F_5: $\mathsf{get}(s_1, k_1) = v_1$
- F_6: $\mathsf{f}_{s_2} = \mathsf{f}_{s_3}$
- F_7: $\mathsf{get}(s_2, k_2) = v_2$

Suppose that:

- *s, s_1, s_2, and s_3 do not initially belong to any $=_{reloc}$ class.*
- *$\mathsf{f}_s \neq k_1 \neq k_2$ and $\mathsf{f}_s \neq k_1 \neq k_3$.*
- *s is chosen as the representative of the newly created class.*

Applying the Reloc-Bounds rule to the formulas yields:

- *F_1: An edge labeled $\mathsf{f}_s - \mathsf{f}_{s_1}$ is added from s_1 to s.*
- *F_2: An edge labeled $\mathsf{f}_s - \mathsf{f}_{s_2}$ is added from s_2 to s.*
- *F_3: Since the distance difference between s_3 and s_2 is $\mathsf{f}_{s_2} - \mathsf{f}_{s_3}$, and the label on the edge from s_2 to s is $\mathsf{f}_s - \mathsf{f}_{s_2}$, an edge labeled $\mathsf{f}_s - \mathsf{f}_{s_3}$ is added from s_3 to s.*
- *F_4: The constraint $v_1 \neq v_2$ is propagated.*
- *F_5: The constraint $\mathsf{get}(s_1, k_1) = v_1$ is propagated over the $=_{reloc}$ class, creating $\mathsf{get}(s_2, k_2) = v_1$, $\mathsf{get}(s_3, k_3) = v_1$, and $\mathsf{get}(s, \mathsf{f}_s) = v_1$.*
- *F_6: The equality $\mathsf{f}_{s_2} = \mathsf{f}_{s_3}$ implies $s_2 = s_3$ after updating the constant difference relation.*
- *F_7: The constraint $\mathsf{get}(s_2, k_2) = v_2$ is propagated. Since F_5 has already created $\mathsf{get}(s_2, k_2) = v_1$, a contradiction is raised, making the problem unsatisfiable.*

Example 5.2 illustrates a general principle: using the labeled union-find in star topology for the equivalence modulo relocation relation ensures that every non-representative n-sequence s_i in a $=_{reloc}$ class with r as the representative has a direct edge to r labeled with $\mathsf{f}_r - \mathsf{f}_{s_i}$.

Concretely, the implementation works as follows: Each n-sequence is either a representative or a non-representative of a $=_{reloc}$ class. Each representative r is associated with a map:

$$\{k_1 \mapsto s_1, \ldots, k_n \mapsto s_n\}$$

where $s_1, \ldots, s_n$ are non-representative n-sequences in the $=_{reloc}$ class of r, and $k_1, \ldots, k_n$ are the labels on the edges from $s_1, \ldots, s_n$ to r, respectively. Each non-representative s_i is associated with a pair (r, k_i), where r is the representative of its $=_{reloc}$ class, and k_i is the label on the edge from s_i to r.

Initially, the representative of a $=_{reloc}$ class is chosen arbitrarily and remains unchanged unless a relation joins an element of one class with an element from another class. In that case, the representative of the larger class (the one containing more elements) is chosen to become the representative of the merged class.

In addition to path compression, edge labels are kept normalized. This is possible because Colibri2 associates a normalized polynomial to each arithmetic term (cf. Section 3.1.1). This normalization simplifies equality detection. For example, given a representative r associated with a map $\{k_1 \mapsto s_1, k_2 \mapsto s_2\}$. If a new non-representative s_3 is added with label k_3, and the equality $k_1 = k_3$ is propagated, then $s_1 = s_3$ can be deduced directly. Moreover, for each representative r, the binding $\{0 \mapsto r\}$ is added to its map of non-representatives, ensuring that if a new term s_0 is added with label 0, the equality $s_0 = r$ can be deduced immediately.

Labeled Union-Find with Map Factorization

In the equivalence modulo relocation relation represented as a labeled union-find, the labels $\mathbb{L}$ are arithmetic terms represented as linear polynomials which, with arithmetic addition, satisfy the group axioms. Therefore, $\mathbb{L}$ forms a group.

Two group actions can be defined over this group:

- $\mathcal{A}_e$: A group action of $\mathbb{L}$ on the element constraints (get operations) of an n-sequence, defined as maps from indices to n-sequence elements, i.e. $V : \mathsf{Int} \to \mathsf{E}$.
- $\mathcal{A}_{NF}$: A group action of $\mathbb{L}$ on the normal form of an n-sequence.

These group actions can be used to factorize constraints on n-sequence terms.

The first group action $\mathcal{A}_e : \mathbb{L} \times V \to V$ is defined over any set of element constraints $m = \{k_1 \mapsto v_1, \ldots, k_n \mapsto v_n\}$ and a difference polynomial δ as:

$$\mathcal{A}_e(\delta, m) = \{(k_1 + \delta) \mapsto v_1,\ \ldots,\ (k_n + \delta) \mapsto v_n\}$$

It has 0 as the neutral element and satisfies associativity for any difference polynomials δ_1 and δ_2:

$$\begin{array}{rcl} \mathcal{A}_e(\delta_1 + \delta_2, m) & = & \{(k_1 + \delta_1 + \delta_2) \mapsto v_1, \ldots, (k_n + \delta_1 + \delta_2) \mapsto v_n\} \\ \mathcal{A}_e(\delta_1, \mathcal{A}_e(\delta_2, m)) & = & \{(k_1 + \delta_1 + \delta_2) \mapsto v_1, \ldots, (k_n + \delta_1 + \delta_2) \mapsto v_n\} \end{array}$$

$\mathcal{A}_e$ allows propagating element constraints from any non-representative n-sequence in a $=_{reloc}$ class to its representative.

Similarly, the second group action $\mathcal{A}_{NF} : \mathbb{L} \times NF \to NF$ is defined over any normal form $n = n_1 :: \ldots :: n_m$ and difference polynomial δ as:

$$\mathcal{A}_{NF}(\delta, n) = \mathsf{relocate}(n_1, \mathsf{f}_{n_1} + \delta) :: \ldots :: \mathsf{relocate}(n_m, \mathsf{f}_{n_m} + \delta)$$

This action also has 0 as the neutral element and satisfies associativity:

$$\begin{array}{rcl} \mathcal{A}_{NF}(\delta_1 + \delta_2, n) & = & \mathsf{relocate}(n_1, \mathsf{f}_{n_1} + \delta_1 + \delta_2) :: \ldots \\ & & :: \mathsf{relocate}(n_m, \mathsf{f}_{n_m} + \delta_1 + \delta_2) \\ \mathcal{A}_{NF}(\delta_1, \mathcal{A}_{NF}(\delta_2, n)) & = & \mathcal{A}_e(\delta_1, \mathsf{relocate}(n_1, \mathsf{f}_{n_1} + \delta_2) :: \ldots \\ & & :: \mathsf{relocate}(n_n, \mathsf{f}_{n_n} + \delta_2)) \\ & = & \mathsf{relocate}(n_1, \mathsf{f}_{n_1} + \delta_1 + \delta_2) :: \ldots \\ & & :: \mathsf{relocate}(n_m, \mathsf{f}_{n_m} + \delta_1 + \delta_2) \end{array}$$

The group actions $\mathcal{A}_e$ and $\mathcal{A}_{NF}$ allow:

- restricting element constraint propagation to be done only from non-representative n-sequences to their representatives.
- storing normal forms only on representative n-sequence terms.

This factorization essentially replaces the rules NS-Comp-Reloc and Get-Reloc with their restricted variants NS-Comp-Reloc-Res and Get-Reloc-Res, as presented in Figure 5.1. The application of these rules is constrained by ensuring that r in NS-Comp-Reloc-Res and Get-Reloc-Res is always the representative of its $=_{reloc}$ class.

$$\text{Get-Reloc-Res}\ \frac{v = \mathsf{get}(s, i) \qquad s =_{reloc} r}{i < \mathsf{f}_s \vee \mathsf{l}_s < i \quad || \quad \mathsf{f}_s \leq i \leq \mathsf{l}_s \wedge v = \mathsf{get}(r, i - \mathsf{f}_s + \mathsf{f}_r)}$$

$$\text{NS-Comp-Reloc-Res}\ \frac{s = k_1 :: k_2 :: \ldots :: k_n \qquad s =_{reloc} r}{\begin{array}{l} \mathsf{f}_s = \mathsf{f}_r \wedge s = r \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad || \\ r = \mathsf{relocate}(k_1, \mathsf{f}_r) :: \mathsf{relocate}(k_2, \mathsf{f}_{k_2} - \mathsf{f}_s + \mathsf{f}_r) :: \ldots \\ \qquad :: \mathsf{relocate}(k_n, \mathsf{f}_{k_n} - \mathsf{f}_s + \mathsf{f}_r) \end{array}}$$

Figure 5.1: Inference rules used to factorize constraints and normal forms on n-sequence terms.

5.1.4 Reasoning

Most of the inference rules of the reasoning are applied as soon as possible. When applying a rule, it is necessary first to determine whether a decision needs to be made to know which of the inference rule's consequences should be applied. For instance, when the term $s_1 = \mathsf{relocate}(s_2, i)$ is encountered, if it is already known that $i = \mathsf{f}_{s_2}$, then $s_1 = s_2$ is propagated immediately. Otherwise, a decision is registered on whether $i = \mathsf{f}_{s_2}$ is true to determine which of the rule's consequences should be propagated.

This applies to the rules Reloc-Bounds, Const-Bounds, NS-Slice, NS-Concat, NS-Update, NS-Split, and NS-Comp-Reloc in Figures 4.1 and 4.2, as well as all rules in Figures 4.3 to 4.5. It also applies to the rules NS-Comp-Reloc-Res and Get-Reloc-Res in Figure 5.1 when they are used.

The rules Get-Concat, Set-Concat, and Set-Concat-Inv are notably re-executed whenever the normal form of an n-sequence is updated.

In contrast, the NS-Exten rule is applied only during the last-effort phase. This means that it is applied to all pairs of n-sequences that are not known to be equal, but only after all other rules, their propagations, and decisions have been executed. When the NS-Exten rule is applied, it may introduce new propagations and decisions, which in turn may introduce additional ones. In such cases, the NS-Exten rule is reapplied only after all new propagations and decisions are completed. If no new propagations or decisions are introduced, the NS-Exten rule is not reapplied.

The weak-equivalence graph (cf. Figure 2.16) is also used when reasoning over n-sequences. In this version, an edge labeled with the index i between two n-sequences a and b is added whenever a term of the form $a = \mathsf{set}(b, i, v)$ or $b = \mathsf{set}(a, i, v)$ is encountered. This graph is exploited during equality checking between two n-sequences a and b. Instead of simply applying NS-Exten to a and b, the weak-equivalence graph is explored if a and b are linked, to find the shortest simple paths between them. If there exists a path P between a and b such that for every index $i \in P$, it holds that $\mathsf{get}(a, i) = \mathsf{get}(b, i)$, then $a = b$ can be deduced.

Conversely, when searching for a difference witness index k, i.e. an index k satisfying $\mathsf{get}(a, k) \neq \mathsf{get}(b, k)$, the selected index k is constrained to lie within P. If $k \notin P$, then the propagation of the element at index k can still be made, therefore a and b cannot differ at index k, leading to a contradiction.

When the rule is applied and there are indices i for which it is not known whether

$\mathsf{get}(a, i) = \mathsf{get}(b, i)$ is true, a decision is registered.

In practice, the NS-Exten rule is applied in the last-effort phase as two last-effort propagations (a) and (b):

(a) Applies the NS-Exten rule only to pairs of n-sequences known to be distinct. After a run of this propagation, if it schedules propagations or registers decisions, it schedules itself again. Otherwise, it schedules propagation (b).

(b) Applies the NS-Exten rule to pairs of n-sequences not yet known to be distinct. If this propagation schedules any further propagations or registers decisions, it then schedules (a), as new disequalities or propagations may help propagation (a) detect contradictions.

This ordering of the propagations (a) and (b) prioritizes applying the NS-Exten rule on pairs of distinct n-sequences as a heuristic, since deducing equality between terms already believed to be distinct is likely to lead to contradictions.

5.1.5 Support for the Theory of Sequences

To ensure compatibility with the theory of sequences of cvc5 and Z3, support for a subset of their versions of the theory of sequences is added. This subset consists of the common sequence operations that are used to represent array-like data structures in programming languages. These operations include: $\mathsf{seq.unit}$, $\mathsf{seq.len}$, $\mathsf{seq.nth}$, $\mathsf{seq.update}$ (in cvc5 only), $\mathsf{seq.extract}$, and $\mathsf{seq.{+}{+}}$.

To support these operations in Colibri2, they are translated internally into operations from the theory of n-indexed sequences as follows:

- Sequence terms: n-sequence terms where the first index is 0 and the last index is ≥ -1.
- $\mathsf{seq.empty}$: Represented by a special constant term $\mathsf{nseq.empty}$, a constant empty n-sequence with first index 0 and last index -1.
- $\mathsf{seq.unit}(v)$: Translated to $\mathsf{const}(0, 0, v)$.
- $\mathsf{seq.len}(s)$: Translated to $\mathsf{l}_s - \mathsf{f}_s + 1$.
- $\mathsf{seq.nth}(s, i)$: Translated to $\mathsf{get}(s, i)$.
- $\mathsf{seq.update}(s, i, \mathsf{seq.unit}(v))$: Translated to $\mathsf{set}(s, i, v)$.
- $\mathsf{seq.update}(s_1, i, s_2)$:

$$\mathsf{let}\Big(r, \mathsf{relocate}(s_2, i), \mathsf{ite}(\mathsf{f}_{s_1} \leq i \leq \mathsf{l}_{s_1} \wedge \mathsf{l}_{s_1} < \mathsf{l}_r, \\ \mathsf{update}(s_1, \mathsf{slice}(r, i, \mathsf{l}_{s_1})), \mathsf{update}(s_1, r))\Big)$$

- $\mathsf{seq.extract}(s, i, j)$:

$$\mathsf{ite}(i < \mathsf{f}_s \vee \mathsf{l}_s < i \vee j \leq 0, \epsilon, \mathsf{relocate}(\mathsf{slice}(s, i, \min(\mathsf{l}_s, i + j - 1)), 0))$$

- $\mathsf{seq}.{+}{+}(s_1, s_2, s_3, \ldots, s_n)$:

$$\begin{array}{l}\mathsf{let}\Big(c_1, \mathsf{concat}(s_1, \mathsf{relocate}(s_2, \mathsf{l}_{s_1} + 1)),\\ \mathsf{let}(c_2, \mathsf{concat}(c_1, \mathsf{relocate}(s_3, \mathsf{l}_{c_1} + 1)),\\ \qquad \ldots\\ \mathsf{concat}(c_{n-2}, \mathsf{relocate}(s_n, \mathsf{l}_{c_{n-2}} + 1)))))\Big)\end{array}$$

5.1.6 Reasoning with Shared Slices

The shared-slices calculus relies heavily on daemons that wait on the creation of new relations to do the propagations from the Get-Over-WEq and Get-Over-ShS rules. Given two n-sequences a and b:

- If a weak-equivalence edge labeled with the index k is created $a \overset{k}{\leftrightarrow} b$: every element constraint on a of the form $\mathsf{get}(a, i) = v_1$, where $i \neq k$, is propagated into b by creating $\mathsf{get}(b, i) = v_1$. The same propagation occurs in the opposite direction, from b to a.
- If a shared-slice relation labeled with bounds f and l is created $a =_{[f;l]} b$: every element constraint on a of the form $\mathsf{get}(a, i) = v_1$, with $f \leq i \leq l$, is propagated into b by creating $\mathsf{get}(b, i) = v_1$. The same propagation is performed from b to a.
- When an element constraint of the form $\mathsf{get}(a, i) = v$ appears:
  - For every weak-equivalence edge $a \overset{k}{\leftrightarrow} c$ linking a to any n-sequence c with any label k, the propagation $\mathsf{get}(c, i) = v$ is made if $i \neq k$.
  - For every shared-slice relation $a =_{[f;l]} c$ linking a to any n-sequence c over any bounds $[f; l]$, the propagation $\mathsf{get}(c, i) = v$ is made if $f \leq i \leq l$.

For the Match-ShS rule, when a shared-slice relation links the two n-sequences b and c, the condition $f = \mathsf{f}_b \wedge f = \mathsf{f}_c \wedge l = \mathsf{l}_b \wedge l = \mathsf{l}_c$ is waited upon, if this condition becomes true, then the equality $b = c$ is propagated.

The Shs-Ext rule follows the same heuristics as NS-Exten described in Section 5.1.4, with the only addition being the check of belonging a shared slice.

5.2 Experimental Evaluation

This section presents experimental results of the implementations described in Section 5.1, of the calculi described in Section 4.3. These experiments were conducted on quantifier-free benchmarks that use only the theories of sequences and n-indexed sequences with the theory of uninterpreted functions.

The benchmarks are a subset of those used by Sheng, Nötzli, Reynolds, Zohar, Dill, Grieskamp, Park, Qadeer, Barrett, and Tinelli [113], which were originally translated into the theory of sequences from the QF_AX SMT-LIB benchmarks [16]. Additionally, the QF_AX benchmarks were translated into the theory of n-indexed sequences to test the native calculi from Section 4.3, and compare them with the encoding of the theory of

n-indexed sequences using the theories of sequences and algebraic data types as described in Section 4.2.1.

Implementations in Colibri2[1] of the NS-BASE, NS-EXT, and NS-ShS calculi can be used with the following commands:

- NS-ShS: `colibri2`
- NS-BASE: `colibri2 --nseq-base`
- NS-EXT: `colibri2 --nseq-ext`

For comparison, cvc5 (version 1.2.0) and Z3 (version 4.13.3) were used as reference solvers. Three configurations of cvc5, each using a different strategy for handling sequence operations, were used:

- cvc5: `cvc5`
- cvc5-eager: `cvc5 --seq-arrays=eager`
- cvc5-lazy: `cvc5 --seq-arrays=lazy`
- Z3: `z3`

Figures 5.2 and 5.3 show the number of goals solved over accumulated time for sequence and n-indexed sequence benchmarks, respectively. Detailed statistics, including the number of goals solved, timeouts, errors, and runtime metrics (average, median, and total solving time), are presented in Tables 5.1 to 5.4.

5.2.1 Translated n-Indexed Sequence Benchmarks

As previously mentioned, similarly to what was done in [113], QF_AX benchmarks were translated into the theory of n-indexed sequences. That was done by replacing the sort of indices with integers, the sort of arrays with the sort of n-sequences, and the array operations select and store with the n-sequence operations get and set, respectively.

To compare the native calculi approach with the encoding approach described in Section 4.2.1, an alternative version of the benchmarks was created, in which the tests were translated into the theory of n-indexed sequences, and the operations of the theory of n-indexed sequences were defined using the operations of the sequence and ADT theories, instead of being treated as built-in theory symbols.

Figure 5.2 shows that NS-ShS performs best overall on unsatisfiable benchmarks, solving a larger number of goals than other solvers in the same time frame. It is followed closely by NS-EXT. NS-BASE performs better than Z3 and slightly worse than cvc5, though it trails behind the others.

Table 5.1 confirms that NS-ShS achieves the best overall results (85 goals solved) with an average runtime of 1.266 seconds. It is followed by NS-EXT which solves 79 goals with a low average runtime of 0.893 seconds. While NS-ShS and NS-EXT have similar

[1] Available at: https://git.frama-c.com/pub/colibrics/-/tree/acta_informatica_2024 (commit SHA: `8d654690eb5c08643a87f0e41334f66311186e40`)

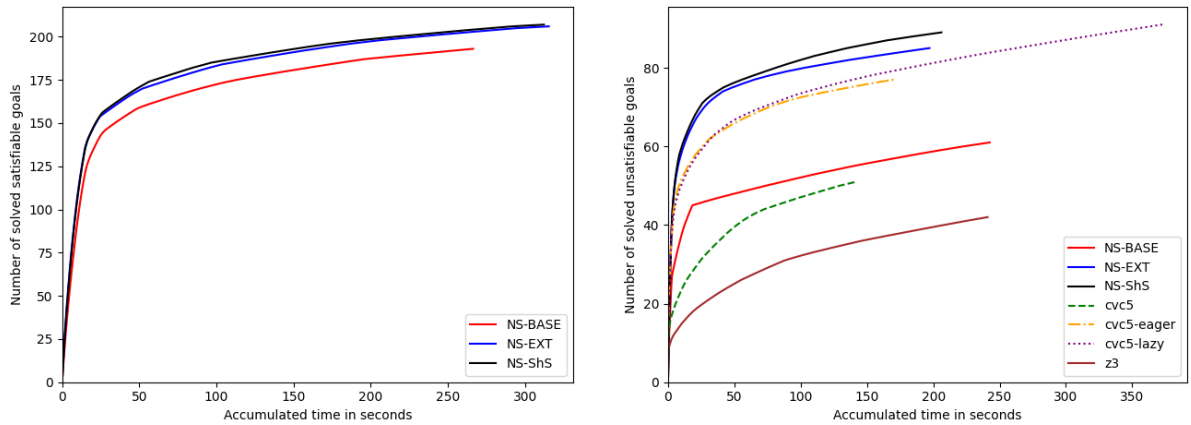

Figure 5.2: Number of solved goals by accumulated time in seconds on quantifier-free NSeq benchmarks translated from the QF_AX SMT-LIB benchmarks.

Solver	Solved	Timeout	Err.	Unk.	Avg. T.	Med. T.	Tot. T.
NS-BASE	48	116	0	0	0.893	0.145	42.844
NS-EXT	79	85	0	0	0.932	0.109	73.589
NS-ShS	85	79	0	0	1.266	0.143	107.599
cvc5	50	114	0	0	2.434	1.573	121.695
cvc5-eager	72	92	0	0	1.301	0.167	93.669
cvc5-lazy	75	89	0	0	1.491	0.211	111.813
z3	25	23	0	0	2.546	2.105	63.656

Table 5.1: Statistics on the performance of the solvers on quantifier-free unsatisfiable NSeq benchmarks.

median times (0.143 and 0.109 seconds, respectively), the average solving time of NS-ShS is higher, and its total runtime exceeds 100 seconds. In contrast, NS-EXT's total runtime is only 73.589 seconds. This suggests that the difference in average and total time is likely due to the six additional tests solved only by NS-ShS that complete near the time limit. It's also worth noting that NS-BASE solves 48 goals, slightly fewer than cvc5 (50 goals), but with significantly lower average, median, and total runtimes (0.893, 0.145, and 42.844 seconds, respectively) compared to cvc5 (2.434, 1.573, and 121.695 seconds).

These benchmarks originate from array benchmarks and contain many get and set operations, which, as shown in the rules of Figure 4.3, require multiple decisions and introduce n-sequence normal forms containing small n-sequence components. As a result, the solver demands substantial computation and does not scale well for such cases.

Solver	Solved	Timeout	Err.	Unk.	Avg. T.	Med. T.	Tot. T.
NS-BASE	158	229	0	0	0.292	0.163	46.189
NS-EXT	212	175	0	0	0.728	0.128	154.234
NS-ShS	197	190	0	0	1.100	0.133	216.742

Table 5.2: Statistics on the performance of the solvers on quantifier-free satisfiable NSeq benchmarks.

Reasoning over such problems would likely benefit from clause learning, which could help manage decisions more efficiently.

Regarding satisfiable goals, Figure 5.2 and table 5.2 shows that only Colibri2 managed to solve goals within the time limit, with NS-EXT clearly surpassing NS-ShS and NS-BASE in both speed and number of goals solved. That is notably due to the presence of quantifiers in the encoding of n-sequences with sequences and algebraic data types that is used for the other solvers, which makes them unable to solve satisfiable problems.

Table 5.2 indicates that while NS-ShS solves fewer goals (197) than NS-EXT (212), its average, median, and total runtimes (1.100, 0.133, and 216.742 seconds) are higher than those of NS-EXT (0.728, 0.128, and 154.234 seconds), suggesting that model generation works better with the word-equation-based approach than with the relations-based approach.

5.2.2 Translated Sequence Benchmarks

Support for the theory of sequences was implemented by encoding it atop the theory of n-indexed sequences. To evaluate the performance of this support for the theory of sequences, it was compared with the calculi implemented in cvc5 and Z3 on sequence benchmarks, which were translated from the QF_AX benchmarks.

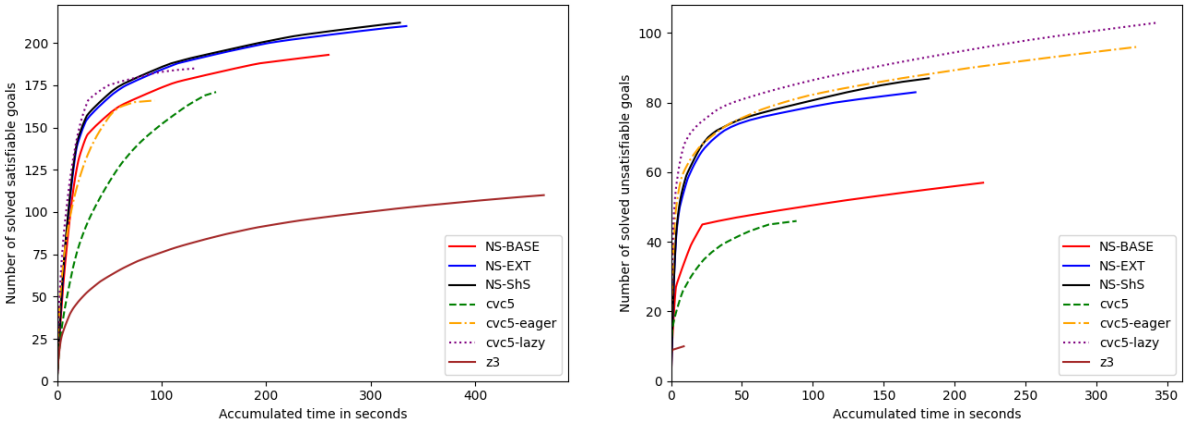

Figure 5.3: Number of solved goals by accumulated time in seconds on quantifier-free Seq benchmarks translated from the QF_AX SMT-LIB benchmarks.

The plot on the right in Figure 5.3 and the data in Table 5.3 show that on unsatisfiable goals, cvc5-lazy performs best overall among the solvers. It is followed by NS-ShS, cvc5-eager, and NS-EXT, whose performances are similar. NS-BASE solves more goals than cvc5 and Z3, and has lower average, median, and total runtimes (0.821, 0.163, and 38.594 seconds, respectively) than cvc5 (1.521, 0.748, and 68.455 seconds).

In the satisfiable case, Table 5.4 shows that NS-EXT performs better in the number of goals solved as well as in average, median, and total runtime (169 goals and 0.245, 0.118, and 41.336 seconds, respectively) compared to NS-BASE (156 goals and 0.359, 0.186, and 55.965 seconds) and NS-ShS (153 goals and 0.642, 0.119, and 98.165 seconds). In terms of goals solved, cvc5-lazy (184) and cvc5 (171) outperform the others. These trends are reflected in Figure 5.3, where the curve for NS-EXT rises steeply early on but levels off

Solver	Solved	Timeout	Err.	Unk.	Avg. T.	Med. T.	Tot. T.
NS-BASE	47	117	0	0	0.821	0.163	38.594
NS-EXT	79	85	0	0	0.938	0.119	74.137
NS-ShS	85	79	0	0	1.397	0.169	118.705
cvc5	45	119	0	0	1.521	0.748	68.455
cvc5-eager	82	82	0	0	1.138	0.123	93.332
cvc5-lazy	90	74	0	0	1.461	0.097	131.477
z3	10	38	0	0	0.818	0.019	8.182

Table 5.3: Statistics on the performance of the solvers on quantifier-free unsatisfiable Seq benchmarks.

Solver	Solved	Timeout	Err.	Unk.	Avg. T.	Med. T.	Tot. T.
NS-BASE	156	231	0	0	0.359	0.186	55.965
NS-EXT	169	218	0	0	0.245	0.118	41.336
NS-ShS	153	234	0	0	0.642	0.119	98.165
cvc5	171	216	0	0	0.893	0.599	152.696
cvc5-eager	165	222	0	0	0.447	0.221	73.791
cvc5-lazy	184	203	0	0	0.599	0.158	110.254
z3	95	408	0	0	2.393	1.375	227.382

Table 5.4: Statistics on the performance of the solvers on quantifier-free satisfiable Seq benchmarks.

before reaching the maximum. A similar trend is observed with NS-ShS and NS-BASE compared to cvc5-eager, the curves are close to one another and cross each other, but cvc5-eager ends up taking over.

5.2.3 Discussion

In the context of program verification, performance on unsatisfiable goals is of greater importance, although the satisfiable cases remain valuable. Since Colibri2 constructs concrete models before concluding satisfiability, it is crucial to improve the current model-generation techniques for n-sequences.

For unsatisfiable goals, Colibri2 performs competitively with state-of-the-art SMT solvers such as cvc5 and Z3. However, it has been observed that certain goals that remain unsolved within a short timeout (e.g. 5 seconds) also remain unsolved even with significantly longer timeouts. This suggests potential performance bottlenecks in the propagators for the NSeq theory or that the problems simply require many decisions.

A notable pattern visible in Figures 5.2 and 5.3 is the presence of inflection points in the performance curves of NS-BASE, NS-EXT, and NS-ShS. These may indicate that the solver struggles with specific classes of problems, warranting further investigation.

It is also worth noting that the translation from the theory of sequences to the theory of n-indexed sequences in Colibri2 often introduces more complex terms. Additionally, Colibri2 currently does not implement clause learning, which can make the search space exploration more costly compared to other SMT solvers.

Chapter 6

Arithmetic II: Extending Arithmetic Reasoning for n-Indexed Sequences

When reasoning over n-indexed sequences as described in Chapter 4, terms of the form $x \leq y (\leq z)$ appear in most inference rules. Therefore, being able to efficiently manage and reason over them is important, first to help with arithmetic reasoning in general, and also to help with reasoning over n-indexed sequences.

This chapter presents some of the experiments that were conducted in this direction. Section 6.1 presents a difference logic reasoning engine that was implemented in Colibri2 to help reason over linear inequalities. Section 6.2 presents different ways in which the labeled union-find data structure, through the constant difference relation, is exploited to improve propagations between arithmetic terms in Colibri2.

6.1 Difference Logic

Difference logic [36] is a fragment of linear arithmetic that is restricted to handling only constraints of the form $x - y \leq c$, where x and y are integer, rational, or real arithmetic variables and c is a numerical constant. Such constraints are common when working on planning, scheduling, and program verification problems.

While the simplex algorithm can solve problems composed of such constraints, in practice, the simplex algorithm is costly and should not be used frequently. Difference logic, on the other hand, is usually reasoned about using a graph data structure and graph exploration algorithms. These can also be costly but might allow making early deductions that would decrease reliance on the simplex, notably in unsatisfiable cases.

6.1.1 The Constraints Graph

Given a difference logic problem defined by a set of arithmetic variables V and a set of difference logic constraints C of the form $x - y \leq c$ such that c is a constant and $\{x, y\} \subset V$. The difference logic constraints graph is a directed labeled graph $G = (V, \Sigma, E)$, where the set of vertices V is the same set of variables V, Σ is the set of labels, which are constant arithmetic terms, and $E \subseteq V \times \Sigma \times V$ is the set of labeled edges. Each edge in the graph $x \xleftarrow{c} y$ represents a constraint of the form $x - y \leq c$. Therefore, the set of edges E is defined as $E = \{(x, c, y) \mid (x - y \leq c) \in C\}$.

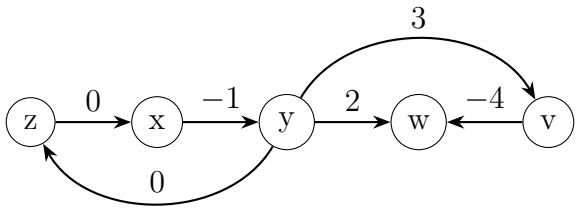

Figure 6.2: Representation of a difference logic problem in the form of a graph.

Two variables x and y are connected in the difference logic graph if there exists a path from one to the other. There is a path from a vertex y to x if there is a chain of edges that goes from y to x: $x \overset{c_n}{\leftarrow} \_ \overset{c_1}{\leftarrow} y$. From the chain of edges, the constraint $y - x \leq c_1 + \ldots + c_n$ can be extracted.

Contradictions can be detected from the difference logic graph. They occur when there exists a negative cycle in the graph. Given n vertices $x_1, \ldots, x_n$, such that each x_i has an outgoing edge with weight d_i towards x_{i-1}, except x_1, for which the outgoing edge goes to x_n, as illustrated in Figure 6.1.

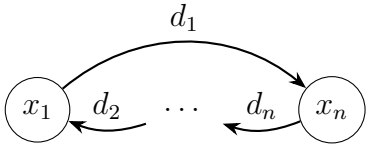

Figure 6.1: Illustration of a cycle in a difference logic graph.

Figure 6.1 represents the following constraints: $x_1 - x_2 \leq d_2 \wedge \ldots \wedge x_n - x_1 \leq d_1$. If the cycle formed by the path from x_n to itself has a negative weight d_{cycle}, then that implies that $x_n - x_n \leq d_{cycle} < 0$, which implies $0 < 0$, which is false.

Equalities can also be detected between variables in the difference logic graph. Given two connected variables x and y, the strongest constraint on $x-y$ corresponds to $x-y \leq d$, where d is the weight of the path with the smallest weight from y to x. If $x - y \leq d$ is the strongest constraint from y to x, and $y - x \leq d'$ is the strongest constraint from x to y, and $d = d' = 0$, then $x - y \leq 0 \wedge y - x \leq 0$, from which $x - y = 0$ can be deduced, resulting in the detection of the equality $x = y$. Therefore, in a difference logic graph, if the smallest weight of the path from x to y and from y to x is 0, then $x = y$.

Example 6.1. *Given the difference logic problem composed of the following constraints:* $y-x \leq -1$, $w-y \leq 2$, $x-z \leq 0$, $z-y \leq 0$, $v-y \leq 3$ *and* $w-v \leq -4$. *The difference logic problem is unsatisfiable because of the negative cycle formed by the constraints:* $x-z \leq 0$, $z - y \leq 0$, *and* $y - x \leq -1$.

Example 6.1 presents an unsatisfiable difference logic problem. Figure 6.2 illustrates the corresponding difference logic graph for the problem in Example 6.1.

6.1.2 Solving Difference Logic Problems

Solving a difference logic problem consists in determining its satisfiability. As mentioned in Section 6.1.1, a difference logic problem is unsatisfiable if its graph of constraints contains a negative cycle. If the graph of constraints does not contain a negative cycle, then the problem is satisfiable.

Bellman–Ford Algorithm

In a directed weighted graph, detecting negative cycles is typically done using the Bellman–Ford algorithm, which computes the paths in a graph from a single source vertex to all other vertices.

```
let iter_edges check Δ s =
  for (s, w, d) in E do
    if Δ[d] > Δ[s] + w then (
      Δ[d ← Δ[s] + w]
      if check then raise Contradiction
    )
  done

let bellman_ford s: E → Σ =
  let Δ = Map.empty in
  Δ[s ← 0];
  for i: 1 to |V| do        (* |V| iterations *)
    iter_edges (i = |V|) Δ s
  done;
  Δ
```

Listing 6.1: Implementation of the Bellman–Ford algorithm used for negative cycle detection in the difference logic graph.

The Bellman–Ford algorithm uses a mapping $\Delta : V \to \Sigma$, in which each $\Delta[x]$ is the distance of x from the source vertex. If $x \notin \Delta$, then $\Delta[x] = +\infty$.

Listing 6.1 shows the code of the algorithm. The function iter_edges iterates over all the edges (s, w, d) of the graph and checks if the distance $\Delta[d]$ is greater than $\Delta[s] + w$. If it is, then $\Delta[d]$ is replaced by $\Delta[s] + w$. This process is called edge relaxation. The function also takes a boolean argument check which determines whether to check for negative cycles, the value of the argument is only true on the nth iteration, where n is the number of vertices in the graph. The algorithm does $n - 1$ iterations in which the paths are simply relaxed. On the nth iteration, if any additional edge relaxation is possible, then the graph contains at least one negative cycle.

The reason for the $n - 1$ iterations is that in a graph with n vertices, the shortest path between any two nodes can contain at most $n - 1$ edges. Therefore, iterating over the edges $n - 1$ times ensures that the shortest paths are computed. During the last iteration, if an edge can still be relaxed after the shortest paths have been computed, then a negative cycle exists.

Incremental Cycle-Detection Algorithm

The incremental negative cycle detection algorithm [36] does not rely on the Bellman–Ford algorithm but checks whenever an edge is added to the difference logic graph if the new edge introduces a negative cycle.

The algorithm uses an artificial vertex $v*$, which has an outgoing edge of weight 0 towards all other vertices. By construction, it is assumed that $v*$ is less than or equal to every other vertex The algorithm also uses a mapping $\Delta_{v_\star} : V \to \Sigma$ that associates to each vertex in the graph its distance from the artificial node $v*$.

```
let neg_cycle_check s h =
  if not (Heap.is_empty h) then (
    let (_, d_heap, i_heap) = Heap.min_elt h in
    Δ_{v⋆}[d_heap ← Δ_{v⋆}[d_heap] + i_heap];
    for (w_out, d_out) ∈ E[d_heap]
      let i_new = Δ_{v⋆}[d_heap] + w_out − Δ_{v⋆}[d_out] in
      let (h, b) =
        match Heap.find_first_opt
          (fun (s', d', _) → s' = d_heap ∧ d' = d_out)
        with
        | None →
          Heap.add (d_heap, d_out, i_new) h, true
        | Some i' when i' > i_new →
          Heap.add (d_heap, d_out, i_new) h, true
        | Some _ → h, false
      in
      if b ∧ s = d_out ∧ i_new < 0 then raise Contradiction;
      neg_cycle_check s h
  )

let on_new_edge s w d =
  let i = Δ_{v⋆}[s] + w − Δ_{v⋆}[d] in
  if i < 0 then
    let h = Heap.singleton (s, d, i) in
    neg_cycle_check s h
```

Listing 6.2: Implementation of the incremental negative cycle detection algorithm in the difference logic graph.

Listing 6.2 illustrates the code of the incremental negative cycle detection algorithm. When a new edge (s, w, d) is added, its improvement to $\Delta_{v_\star}[d]$ is computed as $i = \Delta_{v_\star}[s] + w - \Delta_{v_\star}[d]$. If the new edge indeed improves the distance, i.e. $i < 0$, then the improvement is propagated to all reachable vertices in the graph. As this improvement is propagated, if the distance of the source node $\Delta_{v_\star}[s]$ ends up being improved, then it can be improved indefinitely, meaning that a negative cycle exists.

6.1.3 Implementation

Difference logic reasoning is implemented in Colibri2 with the two negative cycle detection approaches described in Section 6.1.2.

For each term of the form $x \diamond y$, where x and y are arithmetic terms and $\diamond \in \{\leq, \geq, <, >, =, \neq\}$, a decision is made on their truth value. When the value is set, if it is false, the negation is applied (e.g. $\neg(x > y) \rightarrow x \leq y$, $\neg(x \neq y) \rightarrow x = y$, ...).

The normalize_add_constraint function illustrated in Listing 6.3 is used to normalize terms into constraints of the form $x - y \diamond c$, where x or y can be v_0, which is a special variable the value of which is zero.

The domain of polynomials is used to extract the polynomial representation of arithmetic terms through the get_dom_{poly} function. The extracted polynomials are in the form

(c, p) where c is a constant and p is a list of pairs (c_i, v_i) where the non-zero constants c_i are the coefficients and the terms v_i are variables.

```
let normalize_add_constraint env ◇ x y =
  match get_dom_poly env x, get_dom_poly env y with
  | None, None →
    add_constraint (x − y ◇ 0)
  | Some (a_x, []), None →
    add_constraint (v_0 − y ◇ −a_x)
  | None, Some (a_y, []) →
    add_constraint (x − v_0 ◇ a_y)
  | Some (a_x, [(c_x, v_x)]), None when c_x = 1 →
    add_constraint (v_x − y ◇ −a_x)
  | None, Some (a_y, [(c_y, v_y)]) when c_y = 1 →
    add_constraint (x − v_y ◇ a_y)
  | Some (a_x, [(c_x, v_x)]), Some (a_y, [(c_y, v_y)]) when c_x = 1 ∧ c_y = 1 →
    add_constraint (v_x − v_y ◇ a_y − a_x)
  | Some (a_x, [(c_x, v_x)]), Some (a_y, [(c_y, v_y)]) when c_x = −1 ∧ c_y = −1 →
    add_constraint (v_y − v_x ◇ a_y − a_x)
  | Some (a_x, [(c_x1, v_x1); (c_x2, v_x2)]), Some (a_y, []) when c_x1 = 1 ∧ c_x2 = −1 →
    add_constraint (v_x1 − v_x2 ◇ a_y − a_x)
  | Some (a_x, [(c_x1, v_x1); (c_x2, v_x2)]), Some (a_y, []) when c_x1 = −1 ∧ c_x2 = 1 →
    add_constraint (v_x2 − v_x1 ◇ a_y − a_x)
  | Some (a_x, []), Some (a_y, [(c_y1, v_y1); (c_y2, v_y2)]) when c_y1 = 1 ∧ c_y2 = −1 →
    add_constraint (v_y2 − v_y1 ◇ a_y − a_x)
  | Some (a_x, []), Some (a_y, [(c_y1, v_y1); (c_y2, v_y2)]) when c_y1 = −1 ∧ c_y2 = 1 →
    add_constraint (v_y1 − v_y2 ◇ a_y − a_x)
  | _ →
    add_constraint (x − y ◇ 0)
```

Listing 6.3: Normalization of binary arithmetic operations for difference logic.

The constraints are added to the difference logic constraints graph through the add_constraint function. This function takes constraints of the form $x - y \diamond c$, and if $\diamond \in \{\geq, <, >, =, \neq\}$, it converts them into $x - y \leq c$ as follows:

- $x - y \geq c$ becomes $y - x \leq -c$.
- $x - y < c$ becomes $x - y \leq c - \epsilon$.
- $x - y > c$ becomes $y - x \leq -c - \epsilon$.
- $x - y = c$ becomes the two constraints $x - y \leq c$ and $y - x \leq c$.
- $x - y \neq c$: a decision is made to choose which one of $x - y < c$ or $x - y > c$ to set to true, such that:
  - $x - y < c$ becomes $x - y \leq c - \epsilon$
  - $x - y > c$ becomes $y - x \leq -c - \epsilon$

When working on integers, ϵ is simply 1, and when working on rationals (resp. reals), then ϵ is an infinitesimal rational (resp. real) considered smaller than any other rational (resp.real).

Changes in the constant difference relation are taken into account to update the difference logic graph. That is done by defining a hook that is triggered when a representative variable r_o in the constant difference relation becomes a non-representative, with r_n as the new representative and a difference of δ between them. This hook adds, for each outgoing edge from r_o to a vertex n_o with weight c_o, an edge from r_n to n_o with weight $c_o + \delta$, and for each incoming edge from a vertex n_i with weight c_i, an edge from n_i to r_n with weight $c_i - \delta$. The hook then removes the vertex r_o and its incoming and outgoing edges since its constraints are represented by the edges that were added to r_n.

6.1.4 Experimental evaluation

The experimental evaluation was conducted on the SMT-LIB benchmarks of difference logic problems QF_IDL, QF_RDL, QF_UFIDL, and UFIDL, using a timeout of 60 seconds per goal.

The experimentation compares five configurations of Colibri2:

- BASE: the default Colibri2 configuration, using the simplex.
- DL: the Bellman–Ford-based difference logic engine.
- DL-Incr: the incremental difference logic engine.
- DL-Incr-NS: DL-Incr with the simplex disabled.
- DL-NS: DL with the simplex disabled.

The goal of this evaluation is to determine the contribution of the difference logic engines and to assess whether they bring significant improvements over the simplex-based default reasoning.

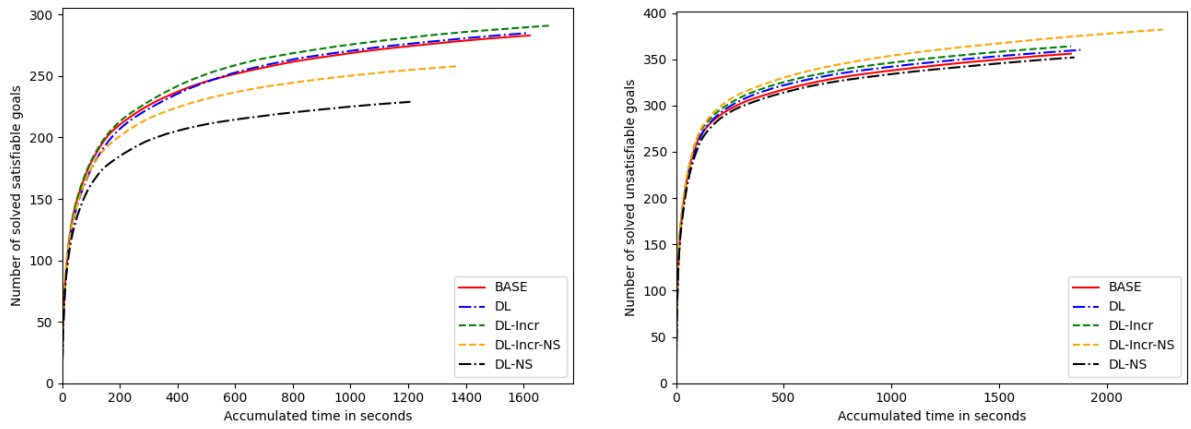

Figure 6.3: Number of solved goals by accumulated time in seconds on difference logic benchmarks.

Solver	Solved	Timeout	Err.	Unk.	Avg. T.	Med. T.	Tot. T.
BASE	283	1500	4	2	5.728	0.959	1621.102
DL	285	1498	4	2	5.665	1.176	1614.497
DL-NS	229	1554	4	2	5.273	0.742	1207.553
DL-Incr	291	1492	4	2	5.800	1.087	1687.679
DL-Incr-NS	258	1525	4	2	5.305	0.934	1368.624

Figure 6.4: Statistics for satisfiable benchmarks using Colibri2 with and without the difference logic engines.

Solver	Solved	Timeout	Err.	Unk.	Avg. T.	Med. T.	Tot. T.
BASE	356	1131	12	10	5.152	0.447	1833.963
DL	360	1136	12	1	5.218	0.504	1878.645
DL-NS	352	1139	12	6	5.257	0.514	1850.597
DL-Incr	364	1132	12	1	5.039	0.492	1834.206
DL-Incr-NS	382	1112	12	3	5.919	0.510	2261.098

Figure 6.5: Statistics for unsatisfiable benchmarks using Colibri2 with and without the difference logic engines.

For satisfiable benchmarks, Figure 6.4 and the left plot of Figure 6.3 show that the combination of the simplex with the incremental difference logic engine DL-Incr produces the best results. The DL configuration follows closely, and its performance is close to that of BASE. The simplex-free variants DL-NS and DL-Incr-NS perform significantly worse, with DL-NS being the weakest.

This degradation of the performance of the approaches that do not use the simplex on satisfiable benchmarks is expected, as the current implementation does not exploit the difference logic engine during model generation, incorporating it would likely improve performance on satisfiable goals.

For unsatisfiable benchmarks, however, Figure 6.5 and the right plot of Figure 6.3 show that three configurations that use the difference logic engines DL-Incr-NS, DL-Incr and DL solve more goals than BASE. With notably the simplex-free incremental solver DL-Incr-NS achieving the best overall performance. This is unsurprising as these benchmarks are specifically designed for difference logic, and the simplex algorithm, known to be costly in worst-case behaviour [116], adds a significant cost, especially in goals for which using it is not necessary. This explains why DL-Incr-NS surpasses even DL-Incr in this setting.

Overall, the difference logic engines provide slight improvements in Colibri2, particularly on unsatisfiable benchmarks, but further work is needed to evaluate their impact on benchmarks not tailored to difference logic. Using the difference logic engines during model generation and adding support for equality detection within the engines is expected to improve results notably on satisfiable problems. A tighter integration with the simplex, allowing mutual sharing of inequalities and enabling each system to handle the constraints best suited to it, may also improve the performance.

6.2 Labeled Union-Find for Constraint Propagation

In Colibri2, the labeled union-find data structure, used through the constant difference relation described in Section 3.2.3, is also used for constraint propagation, notably in the interval domain described in Section 3.1.1, to help with problems for which standard propagations are not sufficient to prove validity.

While working on the theory of sequences (dynamically sized arrays [3]), a lack of propagation of the interval domain between some arithmetic terms was noticed. For example, in Figure 6.6, t is represented as a unique sequence of size `100`, with access modeled as $\mathsf{nth}(\mathrm{t}, 10i+(j+1))$. This requires proving that $10i + (j + 1) \in [0; 99]$, which cannot be deduced from $10i + j \in [0; 89]$ using the basic interval propagation described in Section 3.1.2, which is the method used in Colibri2 by default. This version of Colibri2 will be referred to as BASE.

```
int t[10][10];
if(0 <= 10*i + j < 90){
  a = t[i][j + 1]; ...
}
```

Figure 6.6: Fragment of C program.

Example 6.2. *Given two real variables a and b, a function $f(x) = 2a + x + 3b$ and the assertion that $10 < f(4)$ holds. Then $f(9)^2 \leq 225$ is unsatisfiable for any values of a and b.*

The problem in Example 6.2 only uses one multiplication, yet the basic propagations in BASE are not sufficient to solve it. It would be solvable with a full-fledged decision procedure for non-linear arithmetic, but that is difficult to implement and costly. The simplex algorithm can, in theory, also be used for this propagation, but adding the negation of every unknown comparison or calling a maximization and minimization for every term is costly. Alternatively, a propagation between $f(4)$ and $f(9)$, which are at a constant difference of 5, would suffice.

In this section, two ways to do such propagations are presented. The first one, called LABELED-UF, uses Colibri2's interval domain in conjunction with the constant difference relation to compute a reduced product of the two and produce more refined interval domains for arithmetic terms that are at a constant difference from one another. The second, called GROUP-ACTION, consists in reimplementing the interval domain so that in a class of the constant difference relation labeled union-find, only the interval domain of the representative is maintained, and the domains of the non-representatives in that class can be computed from the domain of the representative when necessary.

6.2.1 Labeled Union-Find for Reduced Product Computation

The constant difference relation makes it possible to propagate constraints on interval domains between arithmetic terms that are in the same constant difference class. The goal with LABELED-UF is to have, for every constant difference class, the following invariants hold:

Invariant 1 When the interval domain of a node is updated and the triggered hooks by the update have finished running, then no additional propagation be-

tween the interval domains of the elements of the class can lead to further refinements.

Invariant 2 Either all elements of the class have a set interval domain, or none of them do.

To ensure that these invariants are always maintained, two additional propagation hooks are needed, one for when the interval domain of a node is updated and another for when two constant difference classes are merged.

Interval Domain Update Hook

The first hook waits on the interval domain of any arithmetic node. When the interval domain is updated, whether it is set for the first time or changed after being set, the hook ensures that the interval domains of all the other nodes in its constant difference class are also updated by recomputing the reduced product of the interval domain and the constant difference relation for all the nodes in the constant difference class.

```
1  let updintrdom_class env (m : L → E) (I_r : R_I): unit =
2    Map.iter (fun n_d^-1 n →
3      upd_dom env n (I_r + n_d)
4    ) m
5
6  let on_updintrdom_hook env (ρ, γ) (x : E) (I_x : R_I): unit =
7    let (x_r, x_d) = ρ[x] in
8    let I_r = I_x − x_d in
9    let r_m = γ[x_r] in
10   let r'_m = Map.remove x_d^-1 r_m in
11   updintrdom_class env r'_m I_r
```

Listing 6.4: Interval domain update hook function.

Listing 6.4 shows the code of the hook function. It takes as arguments the node x, which is the node whose interval domain was changed, as well as the interval domain element I_x, which is the resulting interval domain after the update. The interval domain of x was either initialized to I_x, or I_x resulted from an intersection between x's old domain and a new domain with which it was updated. If the intersection did not refine x's interval domain, then the hook is not triggered.

The hook function on_updintrdom_hook computes I_r (line 8) by subtracting x_d from I_x, which is the interval domain with which to update the interval domain of the representative x_r of the constant difference class of x (line 7). It gets the map r_m of the elements of the constant difference class of x_r (line 9), removes x from it (line 10) because its domain is already updated, then iterates over the elements of the class to compute the new domains with which to update their domains (line 11) with the function updintrdom_class (lines 1 to 4).

This hook ensures that Invariant 2 holds, as it ensures that the domains of all the nodes in a constant difference class are set when the domain of one of them is set. It also partially ensures Invariant 1, since it recomputes the reduced product when the interval domain of a node in a constant difference class changes. However, it is not sufficient,

since propagations can also be done when the representative of a class changes after a merge with another class, for example.

Constant Difference Class Representative Change Hook

The second is a constant difference representative change hook which is triggered whenever two constant difference classes are merged. Its purpose is to propagate the interval domains in both directions between the two classes, which allows computing the reduced product of the resulting constant difference class. This hook is also triggered when a node n is added to a constant difference class s, since it also acts as a merger between a singleton class containing only n and the class s.

```
let repr_change_hook env (ρ, γ) (o_r : E) (δ : L) (r : E): unit =
  match get_dom_I env o_r, get_dom_I env n_r with
  | None, None → ()
  | Some I_o, None →
    let I_n = I_o − δ in
    updintrdom_class env γ[n_r] I_n
  | None, Some I_n →
    let I_o = I_n + δ in
    updintrdom_class env γ[o_r] I_o
  | Some I_o, Some I_n →
    let I_{n+δ} = I_n + δ in
    let I'_o =
      match inter_I I_o I_{n+δ} with
      | None → raise Contradiction
      | Some I'_o → I'_o
    in
    let I'_n = I'_o − δ in
    let o_m = γ[o_r] in
    updintrdom_class env I'_o γ[o_r];
    updintrdom_class env I'_n γ[n_r]
```

Listing 6.5: Constant difference class represetative change hook function.

Listing 6.5 illustrates the code of the representative change hook that is used. When two constant difference classes s_1 and s_2, with representatives o_r and n_r respectively, are merged, and n_r is chosen as the representative of the new class. The hook is called in o_r, δ and n_r, such that δ is the difference between o_r and n_r, and if $\delta \neq 0$, it will be the label on the new edge from o_r to n_r.

This hook function starts by checking the interval domains of both nodes o_r and n_r. And splits into 4 cases:

- When neither o_r nor n_r have a set interval domain, nothing is done.
- When o_r has a set interval domain I_o, while n_r does not, a new interval domain I_n is computed for n_r by subtracting δ from I_o, and is propagated to all the nodes in the constant difference class of n_r.
- Conversely, when n_r has a set interval domain and o_r does not, I_o is computed for o_r by adding δ to I_n.

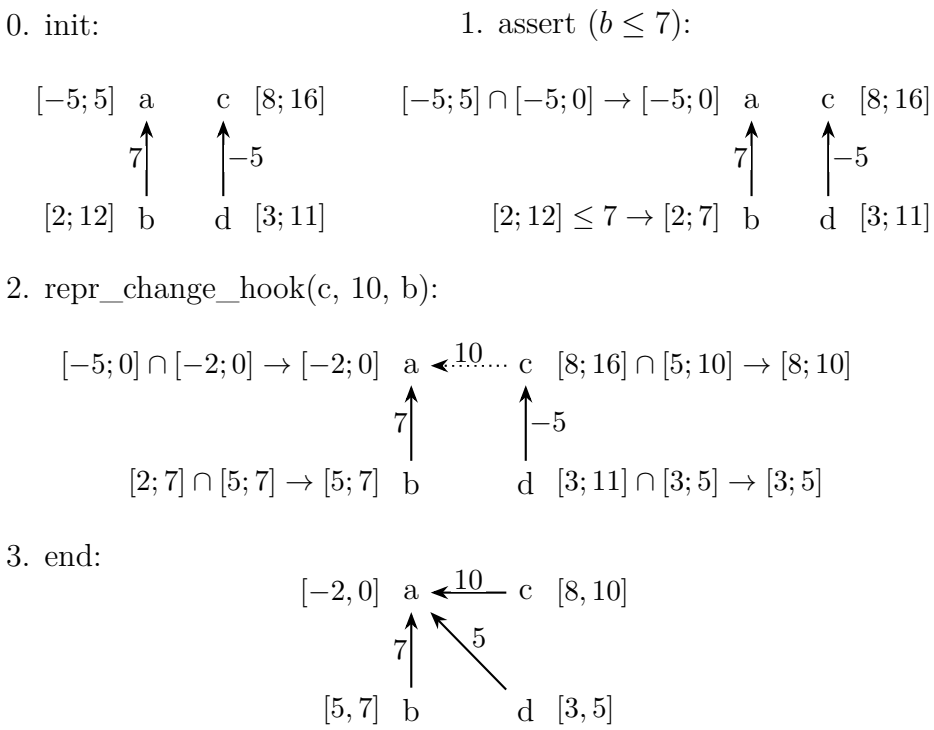

Figure 6.7: Example of the usage of the constant difference relation for constraint propagation over the domain of intervals.

- When both o_r and n_r have set interval domains, I_o and I_n respectively, $I_{n+\delta}$ is created by adding δ to I_n, and an intersection between I_o and $I_{n+\delta}$ is calculated to conciliate the domains of the two representatives (after shifting). If the resulting domain is empty, then a contradiction is raised because it means that the problem is unsatisfiable. If it is not empty, then the result I'_o is used to compute I'_n by subtracting δ. I'_o and I'_n, represent the domains with which to update the interval domains or o_r and n_r respectively, as well as those of their respective constant difference classes.

With this propagation and the one described previously in Section 6.2.1, the two cases in which the interval domain of a node can be refined through propagation over the constant difference class it belongs to are covered, and the reduced product is recomputed in both these cases, ensuring that Invariant 2 holds.

Example

Figure 6.7 shows how the constant difference relation is used for constraint propagation over the domain of intervals using the hooks defined previously. The example starts in step 0 with two trees representing the constant difference relation classes with a and c as their representatives, and with set interval doamins for all the terms.

In step 1 the constraint $b \leq 7$ is asserted, which leads to the pruning of the interval domain of b from $[2; 12]$ to $[2; 7]$, this in turns triggers the domain update hook presented in Listing 6.4, which leads to pruning the interval domain of a as well from $[-5; 5]$ to

$[-5;0]$ though intersection between $[-5;5]$ and $[2;7]-7$ (the new interval domain of b minus the label 7 on the edge from b to a).

The representative change hook is called in step 2 to set the representative of c to a with the label 10. The call falls into the 4th case of the pattern matching in the code of the function, presented in Listing 6.5, with $I_o = [8;16]$, $I_n = [-5;0]$ and $\delta = 10$, therefore $I'_o = [8;16] \cap [5;10] = [8;10]$ and $I'_n = [8;10] - 10 = [-2;0]$. I'_o is then used to prune the domains of c and d, while I'_n is used to prune the domains of a and b.

Finally, in step 3 the merger of the classes is completed after their domains are updated.

6.2.2 Domain Factorization with a Group Action

The operations that are done with LABELED-UF in Section 6.2.1 to compute the reduced product, notably addition between interval domain elements and constants, can be defined as a group action over the group represented by the labels L (constants that represent differences) and the set of interval domain elements $\mathcal{R}_I$.

The group action of sort $\mathcal{A}_I : L \times \mathcal{R}_I \to \mathcal{R}_I$ is an addition operation between an interval domain element and a constant. It simply comes down to addition between an interval union and a singleton interval that represents the constant, i.e. shifting an interval by a constant. The neutral element e_L is zero for constant differences, and since addition is associative, so is the group action.

When subtraction is used, e.g. $I - \delta$, where I is an interval domain element and δ is an arithmetic constant, it is also done through the group action since the subtraction can be rewritten as $I + \delta^{-1}$.

GROUP-ACTION uses this group action definition to factorize the interval domains of all arithmetic terms that are in the constant difference class. That can be done since the group action allows computing the interval domain of any node from any other node. E.g. if $x = y + c$, such that x and y are nodes and c is a constant, if I_y is the interval domain of y, then the interval domain of x, I_x, can be computed: $I_x = I_y + c$. An additional mapping $\theta : E \to \mathcal{R}_I$ option is used to associate each representative r of a constant difference class to its interval domain, which will be used to compute the interval domains of the other non-representative elements of the constant difference class of r.

Listing 6.6 illustrates implementations of the get_dom and upd_dom functions that are used with the implementation of the interval domain as a group action, as well as the repr_change_hook hook function that is necessary to ensure that only one interval domain is stored for each constant difference class.

The get_dom function, called on a node x, simply retrieves the interval domain of the representative of the constant difference class to which x belongs (if it exists), and then applies the group action to compute x's interval domain and return it. Similarly, the upd_dom function, when called on a node x with a new interval domain I_x, subtracts from I_x the distance to the representative r of the constant difference class of x, becoming I_{xr}, and then uses I_{xr} to update r's interval domain. Finally, the repr_change_hook ensures that when two constant difference classes are merged, the interval domains of their representatives are merged and used as the interval domain of the representative of the resulting class.

These functions ensure that this version of the interval domain behaves the same as

```
let get_dom_{A_I} (ρ, θ) x : R_I option =
  let (r, d_x) = ρ[x] in
  match θ[r] with
  | None → None
  | Some I_r → Some (I_r + d_x)

let upd_dom_{A_I} (ρ, θ) x I_x : unit =
  let (r, d_x) = ρ[x] in
  let I_xr = I_x − d_x in
  match θ[r] with
  | None → θ[r ← I_xr]
  | Some I_r →
    match inter_I I_r I_xr with
    | None → raise Contradiction
    | Some I'_r → θ[r ← I'_r]

let repr_change_hook_{A_I} (ρ, θ) (o_r : E) (δ : L) (n_r : E) : unit =
  match get_dom_{A_I} (ρ, θ) o_r, get_dom_{A_I} (ρ, θ) n_r with
  | None, None
  | None, Some _ → ()
  | Some I_o, None →
    let I_n = I_o − δ in
    θ[o_r ← ∅];
    θ[n_r ← I_n]
  | Some I_o, Some I_n →
    let I_{o−δ} = I_o − δ in
    match inter_I I_{o−δ} I_n with
    | None → raise Contradiction
    | Some I'_n →
      θ[o_r ← ∅];
      θ[n_r ← I'_n]
```

Listing 6.6: Definitions of the get_dom, upd_dom and repr_change_hook functions, used by the interval domain implemented as a group action $\mathcal{A}_I$.

the one described in Section 6.2.2, while factorizing the number of interval domains that are used. This approach is called map factorization, and it is described in §5 in Lesbre, Lemerre, Ait-El-Hara, and Bobot [80].

Example

Figure 6.8 is a modified version of Figure 6.7 in which map factorization is used. As shown in step 0, only the representatives of the classes a and c have set interval domains.

When the constraint $b \leq 7$ is asserted in step 1, what is propagated to the representative through the $\mathsf{upd\_dom}_{\mathcal{A}_I}$ function is the interval $]-\infty; 7] - 7$, i.e. the interval from the constraint from which the label on the edge from b to a was subtracted, resulting in an interval domain of $[-5; 0]$ for a.

In step 2, when the $\mathsf{repr\_change\_hook}_{\mathcal{A}_I}$ function is called, its 4th case in its top-level

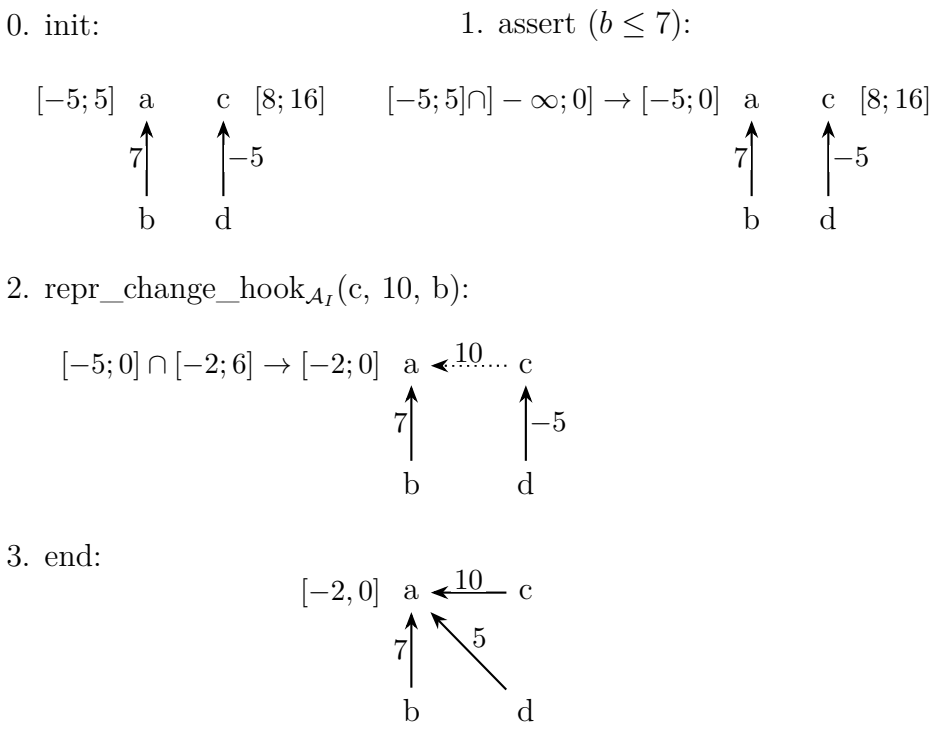

Figure 6.8: Example of the usage of the constant difference relation for constraint factorization over the domain of intervals.

pattern matching is applied with $I_o = [8; 16]$ and $I_n = [-2; 0]$. The new interval is then $I'_n = ([8; 16] - 10) \cap [-2; 0] = [-2; 0]$, which is set as the interval domain of a, while the interval domain of c is removed.

In the last step, the merger of the classes is completed with a as the new representative with the interval domain $[-2; 0]$.

Although the trees seem different, the information is the same as that stored in Figure 6.7. Calling $\mathsf{get\_dom}_{\mathcal{A}_I}$ on b results in $[-2; 0] + 7 = [5; 7]$, while calling it on c results in $[-2; 0] + 10 = [8; 10]$, which are the same domains associated with b and c in Figure 6.7. The same applies for d.

6.2.3 Implementation

The implementations of LABELED-UF and GROUP-ACTION, which were described in Sections 6.2.1 and 6.2.2 respectively, were made in the Colibri2 CP solver.

The original interval domain in Colibri2 was not designed as a group action for the constant difference relation, since it was used to support both reals and integers. It notably included a flag to indicate whether the intervals contained only integers or reals. However, this flag was not a group action: for example, adding and then subtracting 0.5 to an interval of integers would result in a loss of precision. The flag was therefore removed, and the modular congruence domain, described in Section 3.1.1, was added in its stead.

This modular congruence domain is a group action for the constant difference relation. It associates to an arithmetic term t a pair (a, b) which means that t can be written in the

form $a\mathbb{Z} + b$, where $\mathbb{Z}$ denotes any integer constant. This domain serves as an additional constraint on the possible values that a term can have.

Given the group representing constant differences in the constant difference relation L, and the set of rational pairs that represent the elements of the modular congruence domain $\mathcal{R}_{mod}$, the group action of L on the set $\mathcal{R}_{mod}$, of sort $L \times \mathcal{R}_{mod} \to \mathcal{R}_{mod}$, is defined as follows for any $c \in L$ and $(a, b) \in \mathcal{R}_{mod}$:

$$c \oplus (a, b) = (a, (b + c) \mod a)\,.$$

Which has e_L as the neutral element and is associative:

$$\begin{aligned}
(c_1 + c_2) \oplus (a, b) &= (a, (b + c_1 + c_2) \mod a) \\
\\
c_1 \oplus \Big(c_2 \oplus (a, b)\Big) &= c_1 \oplus (a, (b + c_2) \mod a) \\
&= \Big(a, \big((b + c_2) \mod a + c_1\big) \mod a\Big) \\
&= (a, (b + c_1 + c_2) \mod a)\,.
\end{aligned}$$

The initial implementations of LABELED-UF and GROUP-ACTION showed some regressions in the number of solved goals compared to BASE. These were mainly due to slow convergences [31], i.e. very long or infinite sequences of propagations. With the additional propagations that are done in LABELED-UF and GROUP-ACTION, new slow convergences were naturally introduced.

In Colibri2, these slow convergences are normally limited by stopping the propagations on a node's domain when it has been updated too many times (by default, 50 times between decisions). But in this case, the presence of non-linear constraints, rationals, and unbounded variables led to an unforeseen kind of slow convergence: the rational numbers used in the bounds of the intervals grew too fast to fit into memory, e.g. by converging to $\pm\infty$ or to a finite bound, leading to "out of memory" errors. To remedy this, propagations on the interval domain are limited when its bounds take more than 20 memory words. This change allowed LABELED-UF to be on par with BASE in the number of goals solved.

For GROUP-ACTION, a second improvement was made: when propagations of constraints involving multiplications create a domain that uses numbers that are too large in memory, the domains are over-approximated with bounds that use smaller denominators. This change acts as a kind of on-demand floating-point approximation.

These improvements have been applied in all variants of the Colibri2 solver. Although GROUP-ACTION benefited the most from these fixes, LABELED-UF and BASE also improved.

In the resulting implementations, both variants LABELED-UF and GROUP-ACTION are able to propagate between $10i + j$ and $10i + (j + 1)$, for Figure 6.6, and solve Example 6.2. This is something BASE cannot do.

6.2.4 Experimental Evaluation

To evaluate the implementations on a large scale, they were tested on the SMT-LIB [13] 2024 benchmarks [103] for integer and real arithmetic theories, corresponding to the logics QF_LIA, QF_LRA, QF_LIRA, QF_IDL, QF_RDL, QF_NIA, QF_NRA, and

	BASE	LABELED-UF
LABELED-UF	-49 +61 (+12)	
GROUP-ACTION	-65 +70 (+5)	-39 +22 (-17)

Figure 6.9: Comparison of Colibri2 variants using different interval domains on SMT-LIB arithmetic benchmarks.

QF_NIRA, totaling 55,449 problems. These problems include generated problems as well as problems from other use cases of SMT solvers beyond program verification.

The experiments compare LABELED-UF and GROUP-ACTION against BASE, the original implementation in Colibri2. Experiments were conducted using six cluster nodes, each equipped with 72 cores at 3GHz and 187GB of RAM. 30 cores per node were used, for 180 parallel executions, with a time limit of one minute and a memory limit of 4GB per problem.

Figure 6.9 presents the results of the experimental evaluation, the number of newly solved (+), no longer solved (-) problems, and difference for each row compared to column. In total, 17,465 problems are solved by BASE. Comparing only the number of solved problems before the 60s timeout biases the result toward the problems that are solved near the time limit. Therefore, it is considered that a solver variant improves on a problem compared to another solver variant if it solves it in less than 55s (cutoff), while the other is not able to solve it in 60s.

Both LABELED-UF and GROUP-ACTION marginally improve upon BASE, with respectively 12 and 5 more improvements. Interestingly, if a cutoff of 5s is used, 14 problems are solved by both variants in less than 5s, while BASE cannot solve them in 60s (conversely, there are 1 and 3 problems solved, respectively). However, for some problems that remain to be investigated, GROUP-ACTION solves them much slower than LABELED-UF or BASE.

In conclusion, the labeled union-find allowed for an easy implementation of new propagations compared to BASE. LABELED-UF and GROUP-ACTION achieve results comparable to BASE, while being able to solve problems it cannot (e.g. the propagation in Figure 6.6 and Example 6.2). Yet the additional propagations have also introduced regressions, which shows a possible trade-off between performing more propagations and suffering slowdowns in the general case. It is also worth noting that GROUP-ACTION currently lags behind LABELED-UF, its implementation is more complex and requires further refinement. Finally, the next step will be to implement TVPE, which will enable even more new propagations.

Chapter 7

Conclusion and Perspectives

This manuscript presented the results of this thesis. They consist of the theory of n-indexed sequences as well as various ways to reason over it, their implementations, and their experimental evaluations.

In addition, it described the formalization of parts of the arithmetic reasoning and constraint propagation system in the Colibri2 CP solver, as well as the extensions to arithmetic reasoning in Colibri2 that were made to improve its efficiency for reasoning over n-indexed sequences.

This chapter concludes this manuscript with a discussion about some perspectives and possible future work.

7.1 Implementation and Experimental Evaluation

While the implementations have shown comparable results with state-of-the-art SMT solvers, the lack of clause learning in Colibri2 is a clear limitation that prevents scaling the reasoning, notably in problems that intrinsically require many decisions. Therefore, it would be interesting to see how adding some form of clause learning to Colibri2, or implementing the calculi in an SMT solver with proper SAT solving and clause learning, would perform.

Another limitation was noticed with model generation. As previously mentioned, Colibri2 explicitly computes models, therefore, failing to generate a model quickly usually means that constraint propagation is not complete and needs to be improved.

7.2 Proofs of Completeness and Decidability

Proofs of soundness of the inference rules were provided, but the question of completeness was not truly explored. This is something that could eventually help with constraint propagation in general and improve the results.

Another question that was not explored is the decidability of (fragments of) the theory. Since other theories, such as that of strings and 0-indexed sequences, are known to be undecidable, the results would likely be similar.

7.3 Applications

In this thesis, the theory of n-indexed sequences was shown to be useful for reasoning over n-indexed sequences themselves, as well as for encoding 0-indexed sequences and getting comparable results with state-of-the-art SMT solvers.

It would be interesting to explore other use cases for such n-indexed sequences, notably for reasoning about other data structures or even memory model representations.

Bibliography

[1] Erika Ábrahám, James H. Davenport, Matthew England, and Gereon Kremer. "Deciding the consistency of non-linear real arithmetic constraints with a conflict driven search using cylindrical algebraic coverings". In: *Journal of Logical and Algebraic Methods in Programming* 119 (2021), p. 100633. ISSN: 2352-2208. DOI: 10.1016/j.jlamp.2020.100633.

[2] Hichem Rami Ait-El-Hara, François Bobot, and Guillaume Bury. "An SMT Theory for n-Indexed Sequences". In: *Proceedings of the 22nd International Workshop on Satisfiability Modulo Theories*. Ed. by Giles Reger and Yoni Zohar. Vol. 3725. CEUR Workshop Proceedings. Montreal, Canada: CEUR, July 2024, pp. 64–74. URL: https://ceur-ws.org/Vol-3725/#short13.

[3] Hichem Rami Ait-El-Hara, François Bobot, and Guillaume Bury. "On SMT Theory Design: The Case of Sequences". In: *LPAR 2024 Complementary Volume*. Ed. by Nikolaj Bjørner, Marijn Heule, and Andrei Voronkov. Vol. 18. Kalpa Publications in Computing. EasyChair, May 2024, pp. 14–29. DOI: 10.29007/75tl.

[4] Hichem Rami Ait-El-Hara, François Bobot, and Guillaume Bury. "Reasoning over n-indexed sequences in SMT". In: *Acta Informatica* 62.3 (Aug. 2025), p. 33. ISSN: 1432-0525. DOI: 10.1007/s00236-025-00496-w.

[5] Hichem Rami Ait-El-Hara, Guillaume Bury, Basile Clément, and Pierre Villemot. "Constraint Propagation for Bit-Vectors in Alt-Ergo". en. In: *Joint Proceedings of the 23rd International Workshop on Satisfiability Modulo Theories and the 16th Pragmatics of SAT International Workshop*. Ed. by Jochen Hoenicke, Mikoláš Janota, Aina Niemetz, and Sophie Tourret. Vol. 4008. CEUR Workshop Proceedings. ISSN: 1613-0073. Glasgow, UK: CEUR, Aug. 2025, pp. 65–76. URL: https://ceur-ws.org/Vol-4008/#SMT_paper20.

[6] Akers. "Binary Decision Diagrams". In: *IEEE Transactions on Computers* C-27.6 (1978), pp. 509–516. DOI: 10.1109/TC.1978.1675141.

[7] Clément Allain, Basile Clément, Alexandre Moine, and Gabriel Scherer. "Snapshottable Stores". In: *Proc. ACM Program. Lang.* 8.ICFP (Aug. 2024). DOI: 10.1145/3674637.

[8] Léo Andrès, Filipe Marques, Arthur Carcano, Pierre Chambart, José Fragoso Santos, and Jean-Christophe Filliâtre. "Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly". In: *The Art, Science, and Engineering of Programming* 9.1 (Oct. 2024). ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2025/9/3.

[9] "Arrays". In: *The Calculus of Computation: Decision Procedures with Applications to Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 291–310. ISBN: 978-3-540-74113-8. DOI: 10.1007/978-3-540-74113-8_11.

[10] Gilles Audemard and Laurent Simon. "On the Glucose SAT Solver". In: *International Journal on Artificial Intelligence Tools* 27.01 (2018), p. 1840001. DOI: 10.1142/S0218213018400018.

[11] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. "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. Munich, Germany: Springer, 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24.

[12] Clark Barrett, Morgan Deters, Leonardo de Moura, Albert Oliveras, and Aaron Stump. "6 Years of SMT-COMP". In: *Journal of Automated Reasoning* 50.3 (Mar. 2013), pp. 243–277. ISSN: 1573-0670. DOI: 10.1007/s10817-012-9246-5.

[13] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org. 2016.

[14] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.7*. Tech. rep. Available at www.SMT-LIB.org. Department of Computer Science, The University of Iowa, 2025.

[15] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. "Chapter 33. Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. IOS Press, 2021, pp. 1267–1329. DOI: 10.3233/FAIA201017.

[16] Clark Barrett, Aaron Stump, and Cesare Tinelli. "The SMT-LIB Standard: Version 2.0". In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*. Ed. by A. Gupta and D. Kroening. 2010.

[17] Clark W. Barrett, David L. Dill, and Aaron Stump. "A Generalization of Shostak's Method for Combining Decision Procedures". In: *Proceedings of the 4th International Workshop on Frontiers of Combining Systems*. FroCoS '02. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 132–146. ISBN: 3540433813. DOI: 10.5555/646821.706603.

[18] Peter van Beek and Xinguang Chen. "CPlan: a constraint programming approach to planning". In: *Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence*. AAAI '99/IAAI '99. Orlando, Florida, USA: American Association for Artificial Intelligence, 1999, pp. 585–590. ISBN: 0262511061. DOI: 10.5555/315149.315406.

[19] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. "Z3str3: a string solver with theory-aware heuristics". In: *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*. FMCAD '17. Vienna, Austria: FMCAD Inc, Oct. 2017, pp. 55–59. ISBN: 978-0-9835678-7-5. DOI: 10.5555/3168451.3168468.

[20] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. "CaDiCaL 2.0". In: *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*. Ed. by Arie Gurfinkel and Vijay Ganesh. Vol. 14681. Lecture Notes in Computer Science. Springer, 2024, pp. 133–152. DOI: 10.1007/978-3-031-65627-9_7.

[21] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. "CaDiCaL, Gimsatul, IsaSAT and Kissat Entering the SAT Competition 2024". In: *Proc. of SAT Competition 2024 – Solver, Benchmark and Proof Checker Descriptions*. Ed. by Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Vol. B-2024-1. Department of Computer Science Report Series B. University of Helsinki, 2024, pp. 8–10.

[22] Armin Biere, Mathias Fleury, Nils Froleyks, and J.H. Marijn Heule. "The SAT Museum". In: *Proceedings of the 14th Internantional Workshop on Pragmatics of SAT Co-located with the 26th International Conference on Theory and Applicationas of Satisfiability Testing (SAT 2003), Alghero, Italy, July, 4, 2023*. Ed. by Matti Järvisalo and Daniel Le Berre. Vol. 3545. CEUR Workshop Proceedings. CEUR-WS.org, 2023, pp. 72–87. URL: http://ceur-ws.org/Vol-3545/paper6.pdf.

[23] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*. Second edition. Amsterdam: IOS Press, 2021. ISBN: 9781643681610.

[24] Armin Biere and Daniel Kröning. "SAT-Based Model Checking". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Cham: Springer International Publishing, 2018, pp. 277–303. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_10.

[25] N Bjørner, Vijay Ganesh, R Michel, and Margus Veanes. "An SMT-LIB Format for Sequences and Regular Expressions". In: *Strings* (Jan. 2012).

[26] Jasmin Christian Blanchette and Andrei Paskevich. "TFF1: The TPTP Typed First-Order Form with Rank-1 Polymorphism". In: *Automated Deduction – CADE-24*. Ed. by Maria Paola Bonacina. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 414–420. ISBN: 978-3-642-38574-2. DOI: 10.1007/978-3-642-38574-2_29.

[27] Bernard Boigelot, Pascal Fontaine, and Baptiste Vergain. "Decidability of Difference Logic over the Reals with Uninterpreted Unary Predicates". en. In: *Automated Deduction – CADE 29*. Ed. by Brigitte Pientka and Cesare Tinelli. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2023, pp. 542–559. ISBN: 978-3-031-38499-8. DOI: 10.1007/978-3-031-38499-8_31.

[28] Maria Paola Bonacina. "Set of Support, Demodulation, Paramodulation: A Historical Perspective". In: *Journal of Automated Reasoning* 66.4 (Nov. 2022), pp. 463–497. ISSN: 1573-0670. DOI: 10.1007/s10817-022-09628-0.

[29] Maria Paola Bonacina, Stephane Graham-Lengrand, and Natarajan Shankar. "CDSAT for Nondisjoint Theories with Shared Predicates: Arrays With Abstract Length". In: *Proceedings of the 20th International Workshop on Satisfiability Modulo Theories (SMT)*. Ed. by David Deharbe and Antti E. Hyvarinen. Vol. 3185. CEUR Proceedings. CEUR WS-org, Aug. 2022, pp. 18–37. URL: https://ceur-ws.org/Vol-3185/paper9712.pdf.

[30] Maria Paola Bonacina, Stéphane Graham-Lengrand, and Natarajan Shankar. "Conflict-Driven Satisfiability for Theory Combination: Transition System and Completeness". In: *Journal of Automated Reasoning* 64.3 (Mar. 2020), pp. 579–609. ISSN: 1573-0670. DOI: 10.1007/s10817-018-09510-y.

[31] Lucas Bordeaux, Youssef Hamadi, and Moshe Y. Vardi. "An Analysis of Slow Convergence in Interval Propagation". In: *Principles and Practice of Constraint Programming – CP 2007*. Ed. by Christian Bessière. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 790–797. ISBN: 978-3-540-74970-7. DOI: 10.1007/978-3-540-74970-7_56.

[32] Stéphane Bourdais, Philippe Galinier, and Gilles Pesant. "hibiscus: A Constraint Programming Application to Staff Scheduling in Health Care". In: *Principles and Practice of Constraint Programming – CP 2003*. Ed. by Francesca Rossi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 153–167. ISBN: 978-3-540-45193-8. DOI: 10.1007/978-3-540-45193-8_11.

[33] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. "Efficient implementation of a BDD package". In: *Proceedings of the 27th ACM/IEEE Design Automation Conference*. DAC '90. Orlando, Florida, USA: Association for Computing Machinery, 1991, pp. 40–45. ISBN: 0897913639. DOI: 10.1145/123186.123222.

[34] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. "What's Decidable About Arrays?" In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by E. Allen Emerson and Kedar S. Namjoshi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 427–442. ISBN: 978-3-540-31622-0. DOI: 10.1007/11609773_28.

[35] Cristian Cadar and Martin Nowack. "KLEE symbolic execution engine in 2019". In: *International Journal on Software Tools for Technology Transfer* 23.6 (Dec. 2021), pp. 867–870. ISSN: 1433-2787. DOI: 10.1007/s10009-020-00570-3.

[36] Diego Caminha Barbosa de Oliveira. "Fragments of arithmetic in a combination of decision procedures". PhD Thesis. Université Nancy II, Mar. 2011. URL: https://theses.hal.science/tel-00578254.

[37] Jürgen Christ and Jochen Hoenicke. "Weakly Equivalent Arrays". In: *Frontiers of Combining Systems*. Ed. by Carsten Lutz and Silvio Ranise. Cham: Springer International Publishing, 2015, pp. 119–134. ISBN: 978-3-319-24246-0. DOI: 10.1007/978-3-319-24246-0_8.

[38] Edmund Clarke, Muralidhar Talupur, Helmut Veith, and Dong Wang. "SAT Based Predicate Abstraction for Hardware Verification". In: *Theory and Applications of Satisfiability Testing*. Ed. by Enrico Giunchiglia and Armando Tacchella. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 78–92. ISBN: 978-3-540-24605-3. DOI: 10.1007/978-3-540-24605-3_7.

[39] Alain Colmerauer and Philippe Roussel. "The birth of Prolog". In: *History of Programming Languages—II*. New York, NY, USA: Association for Computing Machinery, 1996, pp. 331–367. ISBN: 0201895021. DOI: 10.1145/234286.1057820.

[40] Sylvain Conchon, Evelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. "CC(X): Semantic Combination of Congruence Closure with Solvable Theories". In: *Electronic Notes in Theoretical Computer Science* 198.2 (2008). Proceedings of the 5th International Workshop on Satisfiability Modulo Theories (SMT 2007), pp. 51–69. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2008.04.080.

[41] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. "Alt-Ergo 2.2". In: *SMT Workshop: International Workshop on Satisfiability Modulo Theories*. Oxford, United Kingdom, July 2018. URL: https://inria.hal.science/hal-01960203.

[42] P. Cousot and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, New York, NY, 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[43] Pascal Cuoq and Raphaël Rieu-Helft. "Result graphs for an abstract interpretation-based static analyzer". In: *28èmes Journées Francophones des Langages Applicatifs*. Ed. by Julien Signoles and Sylvie Boldo. Gourette, France, Jan. 2017. URL: https://hal.science/hal-01503064.

[44] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. "BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. 2016, pp. 653–656. DOI: 10.1109/SANER.2016.43.

[45] Martin Davis, George Logemann, and Donald Loveland. "A machine program for theorem-proving". In: *Commun. ACM* 5.7 (July 1962), pp. 394–397. ISSN: 0001-0782. DOI: 10.1145/368273.368557.

[46] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory". In: *J. ACM* 7.3 (July 1960), pp. 201–215. ISSN: 0004-5411. DOI: 10.1145/321033.321034.

[47] Luc De Raedt, Tias Guns, and Siegfried Nijssen. "Constraint Programming for Data Mining and Machine Learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 24.1 (July 2010), pp. 1671–1675. DOI: 10.1609/aaai.v24i1.7707.

[48] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. "Variations on the Common Subexpression Problem". In: *J. ACM* 27.4 (Oct. 1980), pp. 758–771. ISSN: 0004-5411. DOI: 10.1145/322217.322228.

[49] Bruno Dutertre and Leonardo de Moura. "A Fast Linear-Arithmetic Solver for DPLL(T)". In: *Computer Aided Verification*. Ed. by Thomas Ball and Robert B. Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 81–94. ISBN: 978-3-540-37411-4. DOI: 10.1007/11817963_11.

[50] 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. ISBN: 978-3-540-24605-3. DOI: 10.1007/978-3-540-24605-3_37.

[51] Michael D. Ernst, Todd D. Millstein, and Daniel S. Weld. "Automatic SAT-compilation of planning problems". In: *Proceedings of the Fifteenth International Joint Conference on Artifical Intelligence - Volume 2*. IJCAI'97. Nagoya, Japan: Morgan Kaufmann Publishers Inc., 1997, pp. 1169–1176. ISBN: 15558604804. DOI: 10.5555/1622270.1622325.

[52] Stephan Falke, Florian Merz, and Carsten Sinz. "Extending the Theory of Arrays: memset, memcpy, and Beyond". In: *Verified Software: Theories, Tools, Experiments*. Ed. by Ernie Cohen and Andrey Rybalchenko. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 108–128. ISBN: 978-3-642-54108-7. DOI: 10.1007/978-3-642-54108-7_6.

[53] Thibaut Feydy, Andreas Schutt, and Peter J. Stuckey. "Global difference constraint propagation for finite domain solvers". In: *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*. PPDP '08. Valencia, Spain: Association for Computing Machinery, 2008, pp. 226–235. ISBN: 9781605581170. DOI: 10.1145/1389449.1389478.

[54] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3 — Where Programs Meet Provers". In: *Programming Languages and Systems*. Ed. by Matthias Felleisen and Philippa Gardner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 125–128. DOI: 10.1007/978-3-642-37036-6_8.

[55] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. New York, NY: Springer, 1996. ISBN: 978-1-4612-7515-2 978-1-4612-2360-3. DOI: 10.1007/978-1-4612-2360-3.

[56] Eugene C. Freuder and Alan K. Mackworth. "Chapter 2 - Constraint Satisfaction: An Emerging Paradigm". In: *Handbook of Constraint Programming*. Ed. by Francesca Rossi, Peter van Beek, and Toby Walsh. Vol. 2. Foundations of Artificial Intelligence. Elsevier, 2006, pp. 13–27. DOI: 10.1016/S1574-6526(06)80006-4.

[57] Carlo A. Furia. "What's Decidable about Sequences?" In: *Automated Technology for Verification and Analysis*. Ed. by Ahmed Bouajjani and Wei-Ngan Chin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 128–142. ISBN: 978-3-642-15643-4. DOI: 10.1007/978-3-642-15643-4_11.

[58] Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin Rinard. "Word Equations with Length Constraints: What's Decidable?" In: *Hardware and Software: Verification and Testing*. Ed. by Armin Biere, Amir Nahir, and Tanja Vos. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 209–226. ISBN: 978-3-642-39611-3. DOI: 10.1007/978-3-642-39611-3_21.

[59] Yeting Ge and Leonardo de Moura. "Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories". In: *Computer Aided Verification*. Ed. by Ahmed Bouajjani and Oded Maler. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 306–320. ISBN: 978-3-642-02658-4. DOI: 10.1007/978-3-642-02658-4_25.

[60] Evgueni Goldberg and Yakov Novikov. "BerkMin: A Fast and Robust Sat-Solver". In: *Design, Automation, and Test in Europe: The Most Influential Papers of 10 Years Date*. Ed. by Rudy Lauwereins and Jan Madsen. Dordrecht: Springer Netherlands, 2008, pp. 465–478. ISBN: 978-1-4020-6488-3. DOI: 10.1007/978-1-4020-6488-3_34.

[61] Arnaud Gotlieb. "TCAS software verification using constraint programming". In: *The Knowledge Engineering Review* 27.3 (2012), pp. 343–360. DOI: 10.1017/S0269888912000252.

[62] Philippe Granger. "Static analyses of congruence properties on rational numbers (extended abstract)". In: *Static Analysis*. Ed. by Pascal Van Hentenryck. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 278–292. ISBN: 978-3-540-69576-9. DOI: 10.1007/BFb0032748.

[63] Philippe Granger. "Static analysis of arithmetical congruences". In: *International Journal of Computer Mathematics* 30.3-4 (1989), pp. 165–190. DOI: 10.1080/00207168908803778.

[64] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*. 5.0.5. 2012. URL: http://gmplib.org/.

[65] Jun Gu. "Efficient local search for very large-scale satisfiability problems". In: *SIGART Bull.* 3.1 (Jan. 1992), pp. 8–12. ISSN: 0163-5719. DOI: 10.1145/130836.130837.

[66] Aarti Gupta, Malay K. Ganai, and Chao Wang. "SAT-Based Verification Methods and Applications in Hardware Verification". In: *Formal Methods for Hardware Verification*. Ed. by Marco Bernardo and Alessandro Cimatti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 108–143. ISBN: 978-3-540-34305-9. DOI: 10.1007/11757283_5.

[67] Pierre-Emmanuel Hladik, Hadrien Cambazard, Anne-Marie Déplanche, and Narendra Jussien. "Solving a real-time allocation problem with constraint programming". In: *Journal of Systems and Software* 81.1 (2008), pp. 132–149. ISSN: 0164-1212. DOI: 10.1016/j.jss.2007.02.032.

[68] IEEE Computer Society. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Std 610.12-1990. Institute of Electrical and Electronics Engineers. New York, NY, USA, Dec. 1990. DOI: 10.1109/IEEESTD.1990.101064.

[69] Franjo Ivančić, Zijiang Yang, Malay K. Ganai, Aarti Gupta, and Pranav Ashar. "Efficient SAT-based bounded model checking for software verification". In: *Theoretical Computer Science* 404.3 (2008). International Symposium on Leveraging Applications of Formal Methods (ISoLA 2004), pp. 256–274. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2008.03.013.

[70] Artur Jeż, Anthony W. Lin, Oliver Markgraf, and Philipp Rümmer. "Decision Procedures for Sequence Theories". In: *Computer Aided Verification*. Ed. by Constantin Enea and Akash Lal. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2023, pp. 18–40. ISBN: 978-3-031-37703-7. DOI: 10.1007/978-3-031-37703-7_2.

[71] Fredrik Johansson. "Calcium: computing in exact real and complex fields". In: *Proceedings of the 2021 on International Symposium on Symbolic and Algebraic Computation*. 2021, pp. 225–232.

[72] George Katsirelos and Fahiem Bacchus. "Generalized nogoods in CSPs". In: *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 1*. AAAI'05. Pittsburgh, Pennsylvania: AAAI Press, 2005, pp. 390–396. ISBN: 157735236x. DOI: 10.5555/1619332.1619396.

[73] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. "Frama-C: A software analysis perspective". In: *Formal Aspects of Computing* 27.3 (May 2015), pp. 573–609. ISSN: 1433-299X. DOI: 10.1007/s00165-014-0326-7.

[74] Laura Kovács and Andrei Voronkov. "First-Order Theorem Proving and Vampire". In: *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044*. CAV 2013. Saint Petersburg, Russia: Springer-Verlag, 2013, pp. 1–35. ISBN: 9783642397981. DOI: 10.5555/2958031.2958033.

[75] Daniel Kroening and Ofer Strichman. "Equality Logic and Uninterpreted Functions". In: *Decision Procedures: An Algorithmic Point of View*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 77–95. ISBN: 978-3-662-50497-0. DOI: 10.1007/978-3-662-50497-0_4.

[76] Daniel Kroening and Michael Tautschnig. "CBMC – C Bounded Model Checker". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Erika Ábrahám and Klaus Havelund. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 389–391. ISBN: 978-3-642-54862-8. DOI: 10.1007/978-3-642-54862-8_26.

[77] C. Y. Lee. "Representation of switching circuits by binary-decision programs". In: *The Bell System Technical Journal* 38.4 (1959), pp. 985–999. DOI: 10.1002/j.1538-7305.1959.tb01585.x.

[78] K. R. M. Leino and Clément Pit-Claudel. "Trigger Selection Strategies to Stabilize Program Verifiers". In: *Computer Aided Verification*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Cham: Springer International Publishing, 2016, pp. 361–381. ISBN: 978-3-319-41528-4. DOI: 10.1007/978-3-319-41528-4_20.

[79] 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. Dakar, Senegal: Springer-Verlag, 2010, pp. 348–370. ISBN: 3642175104. DOI: 10.5555/1939141.1939161.

[80] Dorian Lesbre, Matthieu Lemerre, Hichem Rami Ait-El-Hara, and François Bobot. “Relational Abstractions Based on Labeled Union-Find”. In: *Proc. ACM Program. Lang.* 9.PLDI (June 2025). DOI: 10.1145/3729298.

[81] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. “A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions”. In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. Cham: Springer International Publishing, 2014, pp. 646–662. ISBN: 978-3-319-08867-9. DOI: 10.1007/978-3-319-08867-9_43.

[82] J.P. Marques-Silva and K.A. Sakallah. “GRASP: a search algorithm for propositional satisfiability”. In: *IEEE Transactions on Computers* 48.5 (1999), pp. 506–521. DOI: 10.1109/12.769433.

[83] Joao Marques-Silva, Ines Lynce, and Sharad Malik. “Chapter 4. Conflict-Driven Clause Learning SAT Solvers”. In: *Handbook of Satisfiability*. IOS Press, 2021, pp. 133–182. DOI: 10.3233/FAIA200987.

[84] Bruno Marre, François Bobot, and Zakaria Chihani. “Real Behavior of Floating Point Numbers”. In: *The SMT Workshop*. SMT 2017, 15th International Workshop on Satisfiability Modulo Theories. Heidelberg, Germany, July 2017. URL: https://cea.hal.science/cea-01795760.

[85] John McCarthy. “Towards a Mathematical Science of Computation”. In: *Information Processing, Proceedings of the 2nd IFIP Congress 1962, Munich, Germany, August 27 - September 1, 1962*. North-Holland, 1962, pp. 21–28.

[86] John W. McCormick and Peter C. Chapin. *Building High Integrity Applications with SPARK*. Cambridge: Cambridge University Press, 2015. ISBN: 978-1-107-04073-1. DOI: 10.1017/CBO9781139629294.

[87] Kenneth L. McMillan, Andreas Kuehlmann, and Mooly Sagiv. “Generalizing DPLL to Richer Logics”. In: *Computer Aided Verification*. Ed. by Ahmed Bouajjani and Oded Maler. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 462–476. ISBN: 978-3-642-02658-4. DOI: 10.1007/978-3-642-02658-4_35.

[88] Antoine Miné, Xavier Leroy, Pascal Cuoq, and Christophe Troestler. *Zarith: Arbitrary-precision arithmetic in OCaml*. Accessed: 2025-06-05. 2010. URL: https://github.com/ocaml/Zarith.

[89] 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*. DAC ’01. Las Vegas, Nevada, USA: Association for Computing Machinery, 2001, pp. 530–535. ISBN: 1581132972. DOI: 10.1145/378239.379017.

[90] Leonardo de Moura and Nikolaj Bjørner. “Efficient E-Matching for SMT Solvers”. In: *Automated Deduction – CADE-21*. Ed. by Frank Pfenning. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 183–198. ISBN: 978-3-540-73595-3. DOI: 10.1007/978-3-540-73595-3_13.

[91] 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. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[92] Leonardo de Moura and Dejan Jovanović. “A Model-Constructing Satisfiability Calculus”. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–12. ISBN: 978-3-642-35873-9. DOI: 10.1007/978-3-642-35873-9_1.

[93] 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, pp. 625–635. ISBN: 978-3-030-79875-8. DOI: 10.1007/978-3-030-79876-5_37.

[94] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. “Generalized, efficient array decision procedures”. In: *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*. IEEE, 2009, pp. 45–52. DOI: 10.1109/FMCAD.2009.5351142.

[95] Alexander Nadel and Vadim Ryvchin. “Chronological Backtracking”. In: *Theory and Applications of Satisfiability Testing – SAT 2018*. Ed. by Olaf Beyersdorff and Christoph M. Wintersteiger. Cham: Springer International Publishing, 2018, pp. 111–121. ISBN: 978-3-319-94144-8. DOI: 10.1007/978-3-319-94144-8_7.

[96] Greg Nelson and Derek C. Oppen. “Fast Decision Procedures Based on Congruence Closure”. In: *J. ACM* 27.2 (Apr. 1980), pp. 356–364. ISSN: 0004-5411. DOI: 10.1145/322186.322198.

[97] Greg Nelson and Derek C. Oppen. “Simplification by Cooperating Decision Procedures”. In: *ACM Trans. Program. Lang. Syst.* 1.2 (Oct. 1979), pp. 245–257. ISSN: 0164-0925. DOI: 10.1145/357073.357079.

[98] Aina Niemetz and Mathias Preiner. “Bitwuzla”. In: *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*. Ed. by Constantin Enea and Akash Lal. Vol. 13965. Lecture Notes in Computer Science. Springer, 2023, pp. 3–17. DOI: 10.1007/978-3-031-37703-7_1.

[99] Robert Nieuwenhuis and Albert Oliveras. “DPLL(T) with Exhaustive Theory Propagation and Its Application to Difference Logic”. In: *Computer Aided Verification*. Ed. by Kousha Etessami and Sriram K. Rajamani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 321–334. ISBN: 978-3-540-31686-2. DOI: 10.1007/11513988_33.

[100] "5. The Rules of the Game". In: *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Ed. by Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 67–104. ISBN: 978-3-540-45949-1. DOI: 10.1007/3-540-45949-9_5.

[101] Marie Pelleau, Antoine Miné, Charlotte Truchet, and Frédéric Benhamou. "A Constraint Solver Based on Abstract Domains". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 434–454. ISBN: 978-3-642-35873-9. DOI: 10.1007/978-3-642-35873-9_26.

[102] Laurent Perron and Frédéric Didier. *CP-SAT*. Version v9.11. Google, May 7, 2024. URL: https://developers.google.com/optimization/cp/cp_solver/.

[103] Mathias Preiner, Hans-Jörg Schurr, Clark Barrett, Pascal Fontaine, Aina Niemetz, and Cesare Tinelli. *SMT-LIB release 2024 (non-incremental benchmarks)*. Version 2024.04.23. Zenodo, Apr. 2024. DOI: 10.5281/zenodo.11061097.

[104] Steven Prestwich. "Chapter 2. CNF Encodings". In: *Handbook of Satisfiability*. IOS Press, 2021, pp. 75–100. DOI: 10.3233/FAIA200985.

[105] "Quantifier-Free Equality and Data Structures". In: *The Calculus of Computation: Decision Procedures with Applications to Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 241–268. ISBN: 978-3-540-74113-8. DOI: 10.1007/978-3-540-74113-8_9.

[106] Jussi Rintanen. "Planning as satisfiability: Heuristics". In: *Artificial Intelligence* 193 (2012), pp. 45–86. ISSN: 0004-3702. DOI: 10.1016/j.artint.2012.08.001.

[107] J. A. Robinson. "A Machine-Oriented Logic Based on the Resolution Principle". In: *J. ACM* 12.1 (Jan. 1965), pp. 23–41. ISSN: 0004-5411. DOI: 10.1145/321250.321253.

[108] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*. USA: Elsevier Science Inc., 2006. ISBN: 9780080463803.

[109] J. Schimpf and K. Shen. "ECLiPSe - from LP to CLP". In: *Theory and Practice of Logic Programming* 12.1-2 (2011), pp. 127–156. DOI: 10.1017/S1471068411000469.

[110] Stephan Schulz. "E - a brainiac theorem prover". In: *AI Commun.* 15.2,3 (Aug. 2002), pp. 111–126. ISSN: 0921-7126. DOI: 10.5555/1218615.1218621.

[111] Bart Selman, Hector Levesque, and David Mitchell. "A new method for solving hard satisfiability problems". In: *Proceedings of the Tenth National Conference on Artificial Intelligence*. AAAI'92. San Jose, California: AAAI Press, 1992, pp. 440–446. ISBN: 0262510634. DOI: 10.5555/1867135.1867203.

[112] Natarajan Shankar and Harald Rueß. "Combining Shostak Theories". In: *Rewriting Techniques and Applications*. Ed. by Sophie Tison. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 1–18. ISBN: 978-3-540-45610-0. DOI: 10.1007/3-540-45610-4_1.

[113] Ying Sheng, Andres Nötzli, Andrew Reynolds, Yoni Zohar, David Dill, Wolfgang Grieskamp, Junkil Park, Shaz Qadeer, Clark Barrett, and Cesare Tinelli. "Reasoning About Vectors: Satisfiability Modulo a Theory of Sequences". In: *Journal of Automated Reasoning* 67.3 (Sept. 2023), p. 32. ISSN: 1573-0670. DOI: 10.1007/s10817-023-09682-2.

[114] Robert E. Shostak. "Deciding Combinations of Theories". In: *J. ACM* 31.1 (Jan. 1984), pp. 1–12. ISSN: 0004-5411. DOI: 10.1145/2422.322411.

[115] Mate Soos, Karsten Nohl, and Claude Castelluccia. "Extending SAT Solvers to Cryptographic Problems". In: *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*. Ed. by Oliver Kullmann. Vol. 5584. Lecture Notes in Computer Science. Springer, 2009, pp. 244–257. DOI: 10.1007/978-3-642-02777-2_24.

[116] Daniel A. Spielman and Shang-Hua Teng. "Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time". In: *J. ACM* 51.3 (May 2004), pp. 385–463. ISSN: 0004-5411. DOI: 10.1145/990308.990310.

[117] Geoff Sutcliffe. "The TPTP Problem Library and Associated Infrastructure". In: *Journal of Automated Reasoning* 59.4 (Dec. 2017), pp. 483–502. ISSN: 1573-0670. DOI: 10.1007/s10817-017-9407-7.

[118] Geoff Sutcliffe. "The TPTP World – Infrastructure for Automated Reasoning". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Edmund M. Clarke and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–12. ISBN: 978-3-642-17511-4. DOI: 10.1007/978-3-642-17511-4_1.

[119] Robert E. Tarjan and Jan van Leeuwen. "Worst-case Analysis of Set Union Algorithms". In: *J. ACM* 31.2 (Mar. 1984), pp. 245–281. ISSN: 0004-5411. DOI: 10.1145/62.2160.

[120] Robert Endre Tarjan. "Efficiency of a Good But Not Linear Set Union Algorithm". In: *J. ACM* 22.2 (1975), pp. 215–225. DOI: 10.1145/321879.321884.

[121] The Rocq Development Team. *The Rocq Prover*. Version 9.0. Apr. 2025. DOI: 10.5281/zenodo.15149629.

[122] Peter van Beek. "Chapter 4 - Backtracking Search Algorithms". In: *Handbook of Constraint Programming*. Ed. by Francesca Rossi, Peter van Beek, and Toby Walsh. Vol. 2. Foundations of Artificial Intelligence. Elsevier, 2006, pp. 85–134. DOI: 10.1016/S1574-6526(06)80008-8.

[123] Willem-Jan van Hoeve and Irit Katriel. "Chapter 6 - Global Constraints". In: *Handbook of Constraint Programming*. Ed. by Francesca Rossi, Peter van Beek, and Toby Walsh. Vol. 2. Foundations of Artificial Intelligence. Elsevier, 2006, pp. 169–208. DOI: 10.1016/S1574-6526(06)80010-6.

[124] George Varghese and Tony Lauck. "Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility". In: *Proceedings of the eleventh ACM Symposium on Operating systems principles*. Vol. 21. 5. New York, NY, USA: Association for Computing Machinery, 1987, pp. 25–38. DOI: 10.1145/37499.37504.

[125] R. Vemuri and R. Kalyanaraman. “Generation of design verification tests from behavioral VHDL programs using path enumeration and constraint programming”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 3.2 (1995), pp. 201–214. DOI: 10.1109/92.386221.

[126] Mark Wallace. “Practical applications of constraint programming”. In: *Constraints* 1.1 (Sept. 1996), pp. 139–168. ISSN: 1572-9354. DOI: 10.1007/BF00143881.

[127] Qinshi Wang and Andrew W. Appel. “A Solver for Arrays with Concatenation”. In: *Journal of Automated Reasoning* 67.1 (Jan. 2023), p. 4. ISSN: 1573-0670. DOI: 10.1007/s10817-022-09654-y.

[128] Ghiles Ziat. “A combination of abstract interpretation and constraint programming”. PhD Thesis. Sorbonne Université, July 2019. URL: https://theses.hal.science/tel-03987752.