12-2022

Uni-Prover: A Universal Automated Prover for Specificationally Rich Languages

Nicodemus Msafiri John Mbwambo
nmbwamb@clemson.edu

Uni-Prover: A Universal Automated Prover for Specificationally Rich Languages

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Nicodemus Msafiri John Mbwambo
December 2022

Accepted by:
Dr. Murali Sitaraman, Committee Chair
Dr. Brian C. Dean
Dr. Jason O. Hallstrom, Florida Atlantic University
Dr. Eileen T. Kraemer
Dr. Joan Krone, Denison University

Abstract

Formal software verification systems must be designed to adapt to growth in the scope and complexity of software, driven by expanding capabilities of computer hardware and domain of potential usage. They must provide specification languages that are flexible and rich enough to allow software developers to write precise and comprehensible specifications for a full spectrum of object-based software components. Rich specification languages allow for arbitrary extensions to the library of mathematical theories, and critically, verification of programs with such specifications require a universal automated prover. Most existing verification systems either incorporate specification languages limited to first-order logic, which lacks the richness necessary to write adequate specifications, or provide automated provers covering only a fixed collection of mathematical theories, which lack the compass to specify and verify sophisticated object-based software.

This dissertation presents an overall design of Uni-Prover, a universal automated prover for atomic sequents to verify software specified with rich languages. Such a prover is a necessary element of any adequate automated verification system of the future. The design contains components to accommodate changes or upgrades that may happen. The congruence class registry at the center of Uni-Prover handles all core manipulations necessary to verify programs, and it includes a multi-level organization for effective searching of the registry. The full functional behavior of the registry component is described mathematically, and a prototype implementation is given. Additionally, the "contiguous instantiation strategy," a strategy that requires neither user-supplied heuristics nor triggers when instantiating universally quantified theorems in any theory, is detailed to minimize verification steps by avoiding the proliferation of sequents in the instantiation process.

Dedication

To my mother Doroth, my wife Josephine, and my daughter Caitlyn. Words cannot explain my thankful heart for being there for me. Let this work be the true testament to your belief in me, encouragement, and support.

Acknowledgments

I want to thank my advisor Murali Sitaraman for his guidance, support, and inspiration throughout this process. Special thanks to William F. Ogden; his insights and ideas have helped me navigate the challenges this work presented. I thank Joan Krone, who has always been ready to advise, guide, and provide countless feedback on my writings. Special thanks to my committee members: Brian C. Dean, Jason O. Hallstrom, and Eileen T. Kraemer for their input on this work.

My appreciation also goes to Reusable Software Research Groups (RSRG) members at Clemson and The Ohio State University. They have contributed immensely to this work with suggestions and feedback, and I am forever grateful for their time.

I acknowledge the support of the U. S. National Science Foundation (NSF) grants CCF-1161916 and DUE-1914667 for this work. However,any opinions, findings, conclusions, or recommendations expressed here are those of the author, and they do not necessarily reflect the views of the NSF.

Outline

List of Tables

List of Figures

List of Source Listings

Chapter 1

Introduction

Software correctness is becoming increasingly important as advancements in computer technology have resulted in society's pervasive dependence on software. Unlike testing, that aims to detect errors in programs but cannot confirm their absence, formal verification can prove that software behaves correctly for all valid inputs, when that software is mathematically specified and specifications are well-formed. Formal verification of component-based software is the focus of this dissertation.

The foundations of formal software specification and verification date back to the work of James C. King [30] and Sir Charles Antony Richard Hoare [26]. King presented a prototype description of a program verifier, which translates programs to machine code and proves their correctness upon applying rules of logic. The work by Hoare described the use of axioms and rules to prove the correctness of computer programs written using a simplified higher-level language. Nevertheless, since the inception of the idea and a more recent 'verifying compiler' grand challenge [27], software verification remains a work in progress. The current state of the field in formal methods, programming language, and software engineering research communities is discussed in Chapter 2.

Formal verification of software produces substantial benefits in software engineering. However, specification and verification add an extra layer of complexity and costs to the traditional software development process. Therefore, software components must be designed and specified to be reusable so that their code is verified once, and the cost of its initial development and verification can be amortized over multiple uses.

Formal verification systems must be designed to manage the rapid growth in software com-

plexity resulting from the increase in the capabilities of computer hardware and continuing plethora of high level languages. As computer hardware becomes powerful, developers quickly create a grander and more elaborate software system that the hardware can support. One consequence is that software progressively becomes larger and more complex. Accordingly, any satisfactory software verification system must include powerful mechanisms for handling the scalability issue.

A central scalability challenge concerns how to describe complex software components comprehensibly and precisely. In software development, objects are typically named according to some metaphorical model of what they could be thought of (queues, stacks, trees, etc.), and the operations are named with verb phrases suggesting their metaphorical effect on their object parameters be (enqueue, advance_right, etc.). While a useful starting point, the metaphorical models themselves are not supportive of rigorous software specification and verification, so precisely formulated mathematical theories must be introduced to provide formal domains as the abstract spaces in which to describe the values of the objects. Such abstract spaces are "obvious" for built-in objects, such as integers, but not for more complex ones.

Furthermore, formal pre-conditions and post-conditions are required to describe operations that manipulate objects to support mathematical reasoning about the software. A postulate here is that if a piece of software or its design is intellectually tractable enough that software engineers can establish the requirements for it and implement it correctly, then there is a collection of mathematical theories in which formal specifications for this software can be formulated, so that, crucially these specifications will also be intellectually tractable. The specifications that meet this level of intellectual tractability are what we term "rich" in this dissertation. Mathematically-extendable rich specifications are necessary to scale up and describe the rich behavior of sophisticated object-based software components. In turn, formal verification needs to scale up to handle rich specifications. We explain this central idea in more detail in the following section.

1.1 Rich Specification of Object-Based Software Components

Object-based software components contain both internal and external details. A component's implementation details are considered internal. Implementation details are generally complex and likely to change [54], an observation that initiated the construction and use of objects in the sixties. On the other hand, the component's specification details are considered external and not likely

to change. The separation enables software developers to write correct programs where the specifications provide the necessary information for clients while employing alternative implementations for unavoidable efficiency tradeoffs.

A good software design should separate component specifications and implementations, as Parnas proposed in [54]. It is unnecessary to overload the client with details irrelevant to their understanding and use of the component. The two principles underlying this design are information hiding and abstraction. The principles are explained in the following section.

1.1.1 Information Hiding and Abstraction in Specification

Information hiding decouples the exterior details of the component from the internal details. A properly formulated software component specification achieves this separation by hiding complex implementation details from clients. The specification should provide a simpler conceptual view of the component with the state (types) of objects, operations, and their parameters. It should also describe how the software component will be used and what the expected results should be after use.

The component specification must be sufficient to communicate all necessary information to the client without revealing how it is implemented. This communication is effectively achieved through abstraction. Abstraction is a technique for developing component specifications that are comprehensible to users without the knowledge of the internals. It complements information hiding by describing the effects of using the component without revealing how those effects are achieved. Abstraction uses mathematical models and notations (abstractions) provided by the specification language.

The specification of a component can be informal or formal. However, only formal specifications are amenable to automated verification. Formal specification uses a traditional language of mathematics, which has a couple of millennia of refinement behind it. It is a universal international language for everyone. It has a rigorous foundation and has been proven adequate for all science and engineering. Mathematical language is also precise, extendable, and expressive.

While formal specifications offer more benefits than informal specifications, it does not mean that all formal specifications achieve the optimal level of abstraction. For example, a software component specified using a language limited to natural number theory may be formal, but will result in unreadable specifications with limited abstraction. Generally, if a specification language

includes just a fixed set of mathematical theories, such a language is still limited and is insufficient to achieve simple, precise, and comprehensible specifications of the behavior of all software components, i.e, such specifications are not rich.

Software specifications may also fail to meet the abstraction goal because they are written directly to describe the implementation code, even if they are formal. Such specifications are often more complex than necessary and limited to a single implementation. For example, a specification for a sorting algorithm written based on a specific implementation would require separate specification for each sorting setup. In contrast, a general, well-engineered prioritizing specification will be universal, and allow multiple implementations that use different data structures and sorting algorithms [66, 64].

1.1.2 Need for Specificationally Rich Languages

Just as programming languages had to advance from earlier high-level languages, such as C, FORTRAN, and Pascal, to modern-day object-based languages, such as C++ and Java, in order to accommodate the expanding software industry, specification languages have to advance too. Programming needs object-based languages to facilitate the construction of intellectually complex software components, so specification needs specificationally rich languages to write desirable component specifications that are comprehensible, simple, scalable, and amenable to automated verification.

When using languages with limited expressiveness to develop software specifications, the specifications quickly become unwieldy as the behavior of software gets complex. Specification languages must be rich, meaning extendable with abstract theories necessary to provide the expressiveness software engineers need to write specifications at an appropriate level of abstraction. The varying complexity of software means the specification language should be adaptable. The notion of richness can be viewed as an "insight optimization" capability of specification languages to allow software components to be described in a way that is most desirable.

1.1.3 Expressiveness in Specification Languages

Richness in a formal specification language includes expressiveness, which depends on the underlying mathematical logic. For example, a language system that extends to first-order logic

allows quantification over variables making it more expressive than propositional logic, which does not support quantification. Given its expressiveness over propositional logic, first-order specifications can be more abstract than specifications written in propositional logic. In this case, first-order logic provides more mathematical entities that can describe software behavior at a higher level of abstraction. We can say specification languages based on first-order logic are "richer" than those based on propositional logic.

Higher-order logic is more expressive than first-order logic. It permits quantification over functions and predicates, and it allows predicates to receive predicate premises. Therefore, specification languages that extend to higher-order logic are richer than those based on first-order logic. Operationally this means that they provide the full expressive power of set theory-based mathematical logic.

Specificationally rich languages extend higher-order logic and are not restricted to a fixed set of mathematical theories, which can offer only limited language constructs, but rather allow for additional mathematical theories as needed so developers can define and use any sophisticated mathematical entities to accurately and clearly describe a component's behavior. This flexibility means any arbitrary mathematical theory can be conceived, given that it provides suitable mathematical models and notations for describing the component in an optimally insightful way.

1.2 Verification Support for Specificationally Rich Languages

The challenge with the flexibility that comes from specificationally rich languages is the support needed to verify programs specified with such languages. The underlying mathematics used in specificatinally rich languages can be extended, and the verification system must include an automated prover that can work with arbitrary theories, which means any prover that relies on fixed underlying mathematical theories (e.g., use of specialized decision procedures) is inadequate. Chapter 2 discusses related existing verification systems to motivate the need for a universal automated prover that can work with specificationally rich languages.

1.3 Problem Statement

Current verification systems are either limited in their usage or the scope of programs they can verify. They are interactive or automated.

Interactive verification systems require humans with specialized knowledge and experience to guide the verification process, limiting their use to only experts.

Some automated verification systems incorporate specification languages limited to first-order logic, which lacks the richness necessary to write adequate specifications for a full spectrum of object-based software components. Others employ an automated prover subsystem that can only work with a fixed collection of mathematical theories, limiting the range of programs they can verify within the available computing resources—time and space.

The challenge is to design a universal automated prover that is not specialized in particular theories, yet is capable of effectively proving a full spectrum of programs that use specificationally rich languages.

1.4 Research Approach

This research is based on the RESOLVE, which is an integrated programming language for verification with a rich specification language. The verification approach is based on Uni-Prover—a Universal Automated Prover for atomic sequents that we have designed.

The Uni-Prover differs from the previous RESOLVE minimalist rewrite prover [58] in its verification capability and technical approach. Designed to explore the limits of a simple prover for verifying well-designed software, the minimalist prover rewrites terms repeatedly to arrive at a proof, and is shown to function well with the collection of theories that occur in components commonly used as classroom examples. It's approach may not scale up effectively when the verification conditions become more challenging for sophisticated components, such as ones presented in [66], and thus accommodate the full specification capabilities of the RESOLVE.

The minimalist prover is designed to work with a previous version of the RESOLVE Verification Condition (VC) generator. The VCs were not generated in a sequent form, so the prover could not leverage sequents. Since then, more developments have been done to the verifying compiler, and its current VC generator produces simplified VCs in sequent form [61]. The Uni-Prover is designed especially for VCs in sequent form. Nevertheless, the results of this work apply to any ver-

ification system that produces sequent VCs or can be extended in that direction. The sequent VCs for Uni-Prover are atomic, consisting of ground terms, and have no logical operators and quantifiers. A detailed discussion of the RESOLVE verification system, along with prior work, is presented in Chapter 3.

The Uni-Prover is designed to work with specificationally rich languages, and we have specified and implemented its central component, the congruence class registry. The following section summarizes the fundamental properties of Uni-Prover compared to existing provers, including those based on decision procedures.

1.4.1 A Summary of Uni-Prover

When programs with rich specifications are verified, the generated sequent VCs frequently lie outside the scope of decision procedures. While the theory domain of decision procedures can be extended to verify the additional sequent VCs, it can only be done with a significant development cost. Unlike decision procedures, in no case would it be necessary to modify or extend the universal prover whenever new mathematical theories are added to the library.

The prover designed in this work verifies atomic sequent VCs that are "obvious" relative to supporting theories that have been appropriately developed: A sequent VC is obvious if its correctness can be established from the available theorems in relatively few proof steps. The working hypothesis is that the resulting VCs would be comparatively straightforward if the software is well-engineered, with suitable specifications and code annotations [31].

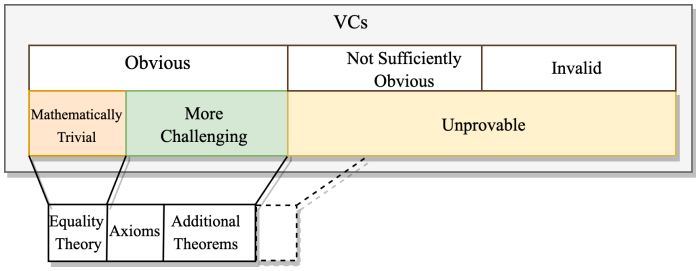


Figure (1.1) Provability scope for verification conditions

Sequent VCs generated from programs fall within the spectrum presented in Figure (1.1). They may be either provable or unprovable. Provable VCs are obvious relative to the current state of the library. Unprovable VCs are either invalid or not sufficiently obvious for automated verification. If a sequent VC is not sufficiently obvious, it might be because the available theorems in the library

are not sophisticated enough, or because it cannot be proved within the resource constraints of the prover.

While the goal of any verification system is to prove all VCs generated from a program, many verification systems are limited by the kinds of sequent VCs they can prove. We have illustrated the verification scope of our prover compared to decision procedures in Figure (1.2). The blue region denotes a space of all possible sequent VCs generated. A small subset of these sequent VCs are mathematically trivial and provable using uninterpreted function decision procedures shown in the green region.

Like uninterpreted function decision procedures, all other decision procedures are domain-specific and prove a limited set of sequent VCs. For example, the Pressburger arithmetic solver is limited to the Pressburger arithmetic theory. Even though it extends what can be proved beyond an uninterpreted function decision procedure, it is still limited by the sequent VCs it can prove within the available computing resources. Example verification coverage of a decision procedure is shown in the orange region. Sequent VCs outside this region may be proved if another decision procedure is developed or an existing one is extended.

The universal automated prover in this work is not limited to a specific domain of theorems. Unlike decision procedures, it can work with arbitrary theories. At its core is the congruence class registry with congruence property of equality. Any atomic sequent VC in an uninterpreted form can be verified based on the congruence property of equality. If our prover is only built with this capacity, its coverage will coincide with that of the uninterpreted function decision procedure in green. However, Uni-Prover is desigened to verify many more sequent VCs depending on the theories (`T`), available computing memory (`M`), and the duration (`D`) it takes to prove a sequent VC. Every added theory in the mathematical library gives our prover more power to prove more sequent VCs. Example coverage of our prover is shown in the diagram in purple containing all sequent VCs provable with respect to `T`, `M`, and `D`.

The math library contains all theories developed for the verification system. Most of these theories will be irrelevant for the sequent VC `S`. `T` will be a subset of theories that are relevant to `S`.

The universal automated prover in this dissertation works with an extendable mathematical library, and any arbitrary theory that can be conceived. Programmers and mathematicians are responsible for elaborating the mathematical library with suitable theorems and reformulating the specifications to produce sufficiently obvious VCs. When more suitable and well-developed mathe-

matical theorems (T) are introduced into the library, more sequent VCs can be proven. The prover must effectively keep the duration (D) minimum and use only the available memory (M). This flexibility is possible only if the prover can maintain a consistent performance when working with both the library's old and new theorems. This is a critical point, because the prover will not be tuned or rebuilt to accommodate what needs to be proved.

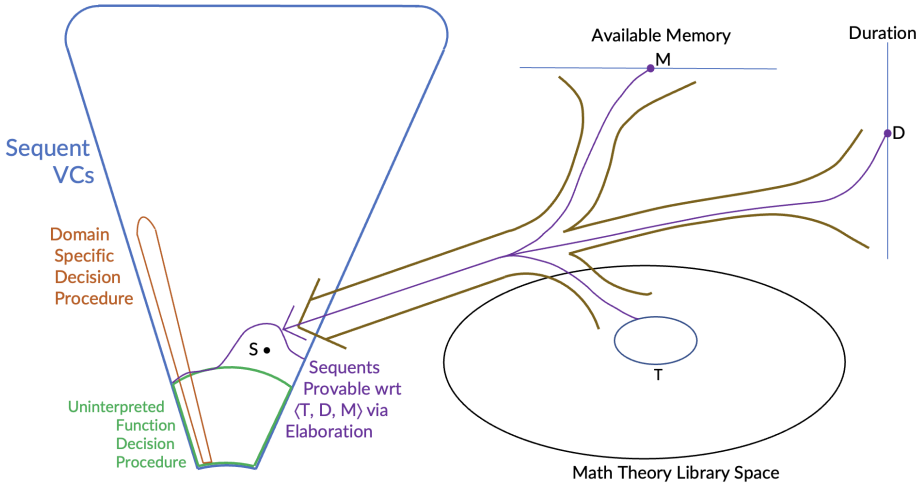


Figure (1.2) Verification scope for Uni-Prover relative to other verification systems

When VCs are complex, the number of verification steps (D) can be numerous and possibly unbounded. The Uni-Prover incorporates novel, generic, and well-engineered strategies and components to achieve the effectiveness needed to reduce the number of verification steps.

The central component we have developed is the congruence class registry, which records the sequent VC to be proved and handles all necessary manipulations to establish whether the sequent VC is correct. The congruence class registry is designed with effectiveness in mind. Among its features is the four-tiered organization that is aimed to support effective searching. The Congruence Class Registry is explained in Chapter 9.

Additionally, we have developed a contiguous instantiation strategy designed to work with any universally quantified theorem. The strategy is general, and it does not require users to provide customized heuristics or triggers. Its effectiveness is in keeping the sequent VC to be verified from branching into many sequents during the verification process. A comparison of the contiguous instantiation strategy to other existing strategies and a complete description of how it works is provided in Chapter 5.

1.5 Contributions

This research addresses the limitations of existing verification systems noted in the problem statement.

The central contribution is a design of Uni-Prover—a Universal Automated Prover for Atomic Sequents. The prover is not limited to a fixed set of theories. It is intended to verify automatically a full spectrum of object-based software components with rich specifications. Such a prover will be crucial for any full-fledged automated verification system of the future.

Contributions that support Uni-Prover include development of a full mathematical specification of the congruence class registry, the core component in the Uni-Prover. The specification describes its functional behavior, serves as an abstraction to support multiple implementations, and will ultimately allow the verification of the core of the prover itself. The registry includes a multi-level searching strategy designed to search congruence classes in the registry effectively. Development of a triggerless and non-heuristic contiguous instantiation strategy designed to work with any universally quantified theorem effectively is a key contribution. The strategy is intended to limit the number of sequents a prover has to verify to conclude whether a target sequent is correct or not, minimizing the number of verification steps it takes.

An illustrative rich specification that uses new mathematical developments to capture the behavior of a generic data abstraction for navigable exploration trees is a motivational contribution.

1.6 Dissertation Organization

The rest of the dissertation is organized as follows: Chapter 2 presents an overview of related verification systems. Chapter 3 provides a background of the RESOLVE verification system with an emphasis on prior prover efforts.

Chapter 4 presents an overview of the Uni-Prover designed in this work. Chapter 5 discusses the contiguous instantiation strategy for universal quantified theorems.

Chapter 6 presents an example of data abstraction in a specificationally rich language for object-based software components.

Chapter 7 discusses the motivation for congruence class registry and briefly describes its mathematical specification. Chapter 8 explains mathematical developments to facilitate the specification of the congruence class registry. Chapter 9 presents the mathematical specification of the

congruence class registry. Chapter 10 presents an overall design of the Uni-Prover, and a prototype implementation of the congruence class registry. Finally, Chapter 11 contains a summary and future directions.

Chapter 2

Related Work

This chapter describes the main characteristics of extant verification systems, their strengths, and their limitations in supporting specificationally rich languages. Prior work in the RESOLVE language literature is detailed in the next chapter.

2.1 Classification of Provers

A prover is an integral part of the verification system. It is ultimately responsible for establishing the correctness of programs, theorems, or both. They can be interactive, automated, or auto-active. Interactive provers involve users in directing the verification process. Users need training and experience to be able to utilize the tactics relating to the new theories effectively. In contrast, automated provers do not need user interactions to complete verification.

Theorem provers are mainly interactive requiring experts with specialized experience to guide the verification of mathematical theorems. Example interactive theorem provers include Lean Theorem Prover [16], which is recently updated to Lean 4 [47] with extensible implementation. Other interactive theorem provers are ACL2 [46], Coq [8], Isabelle [50], which we summarize in Table 2.1. Theorem provers are not the focus of this work.

The focus is on automated provers for programs. Section 2.2 contains a classification of automated provers and places the proposed work in the context of others. In Section 2.3, we discuss related automated verification systems to show the need and justification of the system selected for this work. The discussion in Section 2.4 summarizes our findings on existing related work.

2.2 Classification in Automated Provers

This section presents three classes of automated provers. The characteristics of each class is described to motivate this work.

2.2.1 Decision Procedures

Decision procedures are designed for a specific mathematical domain to maximize their verification efficiency in solving particular problems in their domains. While they achieve a high efficiency in their domains, decision procedures are the least flexible. They are typically hard-coded to utilize domain-specific information and only work on a limited class of decidable theories. New theorems can be accommodated in decision procedures, but they come with significant development and certification costs.

Many decision procedures exist in formal verification systems. One example is uninterpreted function solver, a decision procedure for uninterpreted functions. The second example is a fast linear arithmetic theory solver [19], which is used in Z3 [15] and Yices [18]. Fast linear arithmetic theory solver is specifically designed to provide efficient theory propagation and fast backtracking for only quantifier-free linear arithmetic equalities and inequalities. Z3's another decision procedure is a generalized efficient array procedure, which is presented in [13].

A RESOLVE-specific example decision procedure is the SplitDecision system [1] developed at The Ohio State University. SplitDecision procedure is developed for RESOLVE's widely used mathematical theories, such as sets, integers, and strings. The limitations allow it to employ specific heuristics to improve the performance, but leave it inadequate for proving programs specified using rich languages.

2.2.2 SMT Solvers

Satisfiability Modulo Theory (SMT) solvers are designed to operate with a fixed set of mathematical theories, making them fast and reliable for automated verification. SMT solvers generalize Boolean satisfiability (SAT) and establish program correctness by first expressing the verification problem as a SAT problem. The SAT problem itself is a well-known NP-complete problem [52]. Different branch-and-bound techniques (such as Davis-Putnam-Logemann-Loveland (DPLL) [49]) efficiently solve the problem by decreasing the search space for the most practical

formulae.

SMT solvers are limited to first-order theories, which are not expressive enough to capture the behavior of all object-based components at the level of abstraction needed. Z3 [15] for example, is a widely employed SMT solver in automated verification systems. It has been used in Dafny [42], HAVOC [39], spec#[7], Why3 [20] and others. Another SMT solver is Yieces [18]. Yieces has been used in program verification and model checking. It is mostly integrated as the primary decision procedure in theorem provers such as PVS [53]. Yieces employs a modern variant of DPLL as a SAT-solving decision procedure.

While SMT solvers are useful, their limitation to first-order theory restricts their usability to verification of programs with specifications based on first-order theory. However, their efficiency and reliability have influenced their application in many provers, which integrate them to simplify the verification process by dealing with formulas or proof obligations that prepositionally would not contribute to the proof.

The universal automated prover proposed in this work does not incorporate SMT solvers, though it does not mean that it could not be used in conjunction with such solvers.

2.2.3 Term Rewriting Provers

Automated provers based on term rewriting are flexible and general compared to SMT solvers. They are based on rewriting terms, and they go through a series of deductions, applying theorems at each step. Term rewriting provers start with facts (givens) and goals and transform them on each step until one of the goals is proven valid from the givens. This deduction process is similar to how humans would approach a verification problem. Example verification systems with provers that have employed term-rewriting include ACL2 prover [46] and the RESOLVE minimalist prover [58].

Term rewriting was proposed to solve the problem of reasoning about equality and has been one of the successful methods explained in [29]. Term rewriting works by replacing terms (subexpressions) with other terms. The prover proposed in this work focuses its techniques at the clause (expression) level, because such an approach is efficient. Clauses in this work are considered atomic, free from any logical operators. One example we discuss in a later section is the use of elaboration rules, which are defined and created from the theorems at the clause level. Even though elaboration rules are represented by clauses from the theorem, applying them involves working at a

term level.

A significant difference between the term rewriting provers and the prover in this work is in the rewriting process itself. When a theorem instance is added to the sequent as part of an elaboration process, an entire collection of congruence classes is rewritten and not just a term as it would be in term rewriting provers. It takes many rewriting steps to get the same results we intend to achieve in one congruence class rewrite.

2.3 Related Automated Verification Systems

The prover in this work does not fall in any of the classifications provided in Section 2.2. It bears some similarities to term rewriting provers, but most features are unique and set it apart from existing provers.

This section summarizes existing automated verification systems most related to our work. The focus is on their support of specificationally rich languages, higher-order assertions, extendable mathematical library, and automation, the four essential features at the core of the automated prover discussed in this work. Table 2.1 contains a characterization of existing systems using these four features.

2.3.1 KeY

KeY [2] is a verification system that uses Java Modeling Language (JML)[40] to specify the behavior programs through specialized classes and methods level comments. JML syntax and semantics extend that of Java programing language, tying the two languages together.

JML* [65] extends JML to support abstraction and modular verification. JML* specifications can hide the implementation details and achieve reusable abstractions and scalable verification. The two major abstraction support in KeY includes method contracts and loop invariants.

KeY does not support higher-order logic. It employs Dynamic Logic (DL) for Java, a first-order, multi-modal logic that extends Hoare Logic[26]. All proof obligations in KeY are represented in what are known as DL formulas before a theorem prover for logic can establish their correctness.

Verification in KeY is automatic for programs free from recursion and loops. For programs that fail to verify automatically, KeY offers an interactive proof assistant for users to guide the proof process manually. In this mode, users can apply inference rules (or tactics) in steps towards the

goal.

KeY has tailored sets of hard-coded sequent calculus rules are defined in "taclet" (i.e., tactics) language. The rules are applied by the user or a prover depending on whether the verification is interactive or automatic.

2.3.2 Dafny

Dafny [42] is a program verification system that uses Dafny, an object oriented, imperative specification and programming language. Its programming and specification languages are the same. The expressions have similar syntax and meaning. Dafny offers features from object-based, imperative, sequential, and functional programming. It supports class types, sets, sequences, algebraic data types, and has a limited collection of (parametric) data objects for which it has proof support. The programmer can use Dafny to specify programs through specification constructs built in the programming language, just like Eiffel [45, 44], JML [40], and Spec# [7], where specifications are not separated from program code.

Verification of a program in Dafny employs both Boogie [6] and Z3 [15]. The program to be verified is translated to an equivalent intermediate verification language, Boogie. Dafny then uses the Boogie tool to generate first-order proof obligations for the target program. The proof obligations are then sent to Z3 (a first-order SMT solver) prover for verification. Boogie is a layer on which verifiers for other languages can be built on top.

For non-trivial programs, Dafny may require assistance in the form of hints from the user to establish the program's correctness. The language provides constructs for the user to write, prove, and use lemmas. Lemmas in Dafny are implicitly ghost methods defined to take parameters, preconditions and postconditions (describing the behavior of each lemma), and statements (providing the proof of the defined behaviors) within the body.

2.3.3 Prusti

Rust [32] is a programming language designed for performance, reliability, and productivity. Rust is unique with its ownership type system, that sets it apart from many programming languages by eliminating problems caused by reference aliasing, data races, and dangling pointers.

Prusti [4, 3] is a formal verification system that leverages Rust's type system allowing

simplified specification and verification of Rust programs. In Prusti, Implicit Dynamic Frame (IDF), a variant of the traditional separation logic, is used. Prusti verifier is a general-purpose deductive verifier with the goals of verifying expressive program properties, alleviating programmers from annotation burden, and providing an easy integration to the Rust programmer's workflow.

Prusti verification is based on Viper [48], an intermediate language providing a verification infrastructure for tools based on separation logic and other permission logic. Verification in Viper, as in Dafny, employs both Boogie [6] and Z3 [15].

2.3.4 Why3

Why3 [20] is a verification platform for deductive programs that uses a logic language Why3 and a programming language WhyML for the specification and implementation of functional programs, respectively. The two languages are tied together, and any logical symbol can be used in both languages. Why3 is based on first-order logic and deals with higher-order logic assertions by converting them to first-order logic used by the backend provers.

WhyML allows programmers to annotate their implementations with loop invariants and progress metrics used for termination. Programs are verified by generating verification conditions (VCs), which are then sent to one of the external provers. The employed prover can be automatic or interactive. The set of provers currently used includes Coq[8], PVS[53], Vampire[34], and Z3[15].

Why3 is classified as an automatic verification system, and users have no direct control over how proof is carried out. Instead, Why3 automatically generates proof obligations and translates them to an acceptable format for external provers.

Why3 comes with a standard library of logical theories containing reusable theory modules with definitions, predicates, and lemmas helpful in specifying programs. In addition to built-in theories describing integers, lists, functions, and trees, the library can be extended with new theories written by the user. This feature sets Why3 apart from many existing verification systems, making it closer in spirit to the RESOLVE Uni-Prover discussed in this work.

A key difference between Why3 and Uni-Prover is that the latter employs a mathematical library that can deal with higher-order theories. Specifications in Why3 are based on first-order logic with few extensions to higher-order logic. In principle, the Uni-Prover in this work can be integrated as one of the external provers for Why3.

2.3.5 AutoProof

AutoProof [62] is an automated verification system to verify functional properties of sequential object-oriented programs. It works with Eiffel [45, 44], a language to specify and verify programs. Verification of programs in AutoProof is automated, but users are expected to provide annotations along with their program. The annotations use Eiffel language constructs, including preconditions and postconditions, invariants, variants, and inline assertions.

AutoProof provides a specification library known as the Mathematical Model Library (MML) usable in complex specifications. MML contains classes linked to background mathematical theories for sets, relations, functions, and other theories to abstractly model interfaces. MML is extendable with new classes wrapping a new mathematical construct written by a user through logic classes. The limitation is that, all the extensions have to be compatible with supported Boogie types.

Unlike Dafny and KeY, the model-based specification style followed in AutoProof allows users to model interfaces in an implementation-neutral way. This property provides the separation between specifications and implementation needed to achieve a higher level of abstraction than Dafny and KeY. Despite the extension capabilities available in AutoProof, the limitation to Boogie types precludes it from supporting full specificational richness.

Verification of programs in AutoProof works by converting the input program into a collection of verification conditions (VCs) in a two-step process, whereby Eiffel-specified programs are first converted to a Boogie program which is then used by the Boogie tool to generate VCs. The correctness of each VC is established with an SMT solver. Z3 is used as the default solver.

2.4 Discussion

A summary of existing verification systems is given in Table (2.1). They are separated into automated systems, such as AutoProof, Dafny, KeY, RESOLVE, VeriFast, Prusti, and Why3, and interactive systems, such as Coq, Lean, and PVC.

Automated verification systems require less interactive effort and expertise from the user. However, not all automated verification systems employ specificationally rich languages. Their specifications are limited to first-order logic. Some verification systems have an extensible mathematical library but within first-logic. Why3 and RESOLVE provers support higher-order specifications with some limitations. Why3 deals with higher-order assertions by converting them to first order(with

some limitations). The Uni-Prover does not directly accommodate quantifiers in code annotations and existentially quantified theorems in mathematical developments. Table (2.1) acknowledges these limitations with a * in the 4^{th} column. This research is based on the RESOLVE verification system, which has an extendable mathematical library and can support higher-order logic.

Interactive verification systems have extensible mathematical libraries and support higher-order logic. Their specification languages are rich and can describe software at the right level of abstraction. However, they lack automation in verification because users need to interact with the verification process to guide the proof.

Table (2.1) Summary of Current Full Verification Efforts

Program Verification Effort	**Fully Automated Verification**	**Has Extensible Mathematical Library**	**Supports Higher Order Specification**	**Supports Specification-ally Rich Language**
Dafny	Yes	No	No	No
Prusti	Yes	No	No	No
WhY3	Yes	Yes	Yes*	No
KeY	Yes	No	No	No
AutoProof	Yes	Yes	No	No
RESOLVE(with Uni-Prover)	**Yes**	**Yes**	**Yes***	**Yes**
Coq	No	Yes	Yes	Yes
Lean	No	Yes	Yes	Yes
PVS	No	Yes	Yes	Yes

Chapter 3

RESOLVE Background

This chapter summarizes prior foundational work on RESOLVE that forms the basis for this research. We include a brief history of the development of the RESOLVE language, with emphasis on work to establish correctness. We discuss various verification efforts and two existing RESOLVE provers, along with details of the current RESOLVE verifying compiler and its architecture. The chapter concludes with examples in the RESOLVE language as a prelude to the rest of the dissertation.

3.1 RESOLVE Verification System Foundations

The RESOLVE verification system is based on results from over three decades of research since the idea was conceived in the 1980's. An excellent summary of the RESOLVE research efforts can be found in [51, 57]. This section discusses verification research most related to this work.

3.1.1 The Role of Verification in Software Reusability

Joan Krone's work in 1988 [35] addresses the design and development of verifiable and reusable software. Reusability is a critical software design principle to amortize the cost of formal specification and verification. A reusable component must have a separate conceptual module (specifications) from the realization module (implementation). The separation is essential to verify programs solely based on their specifications and promote efficiency through a choice of implementation for the concept specifications.

Krone's work motivated the need for a system with a specification language independent from a programming language. The specification language should describe the component's functions precisely and concisely. Meanwhile, the programming language should make it possible to write modular, efficient, and reusable code. Most importantly, Krone's work provides a set of rigorous mathematical proof rules for programming language constructs to establish code correctness. These proof rules have motivated many subsequent RESOLVE efforts.

3.1.2 Computer Program Verification: Improvements for Human Reasoning

In 1995, Wayne Heym presented an indexed method for reasoning about modular imperative programs [25]. This method is an alternative to the traditional goal-directed formal reasoning method, such as the one presented in Krone's work. The indexed method is more natural and it fits how programmers reason about code from top to bottom. It formalizes the informal reasoning practice where programmers look at a statement and focus on how it affects the values of variables after its execution and conditions that hold on each branch for those values. The method has been used extensively in undergraduate computer science education [9].

The indexed method offers two benefits. First, it allows users to select the order in which groups of statements can be reasoned about independently. Second, the mathematical assertion structure built in the indexed method matches the static structure of the programs. However, the indexed method uses many names in the verification process to maintain the naturalness, increasing the number of steps needed in automated verification and generating assertions that may not be necessary for proving the goals.

3.1.3 Direct Reasoning

A key complication in reasoning about programs is the use of references. References are unavoidable for efficient computing. Copying references (as opposed to entire objects) allows constant-time data movement and parameter passing for all objects. The downside of copying references is complicated reasoning caused by aliasing when objects are mutable. The RESOLVE language includes swapping as an efficient alternative to copying [23].

Beginning with the prior work of Harms on swapping, Kulczycki presents the idea of direct

reasoning [36]. Direct reasoning is possible in languages with "clean semantics". Semantics for a language are considered clean if the representation of the program's state is a collection of abstract values of all defined variables, and invoking an operation only affects parameters accessible to the operation. Direct reasoning is achieved using the swap operation for data movement, avoiding parameter aliasing, and encapsulating data structures involving references. Kulczycki's work includes a specification of a data abstraction for capturing references on which other components can be layered.

3.1.4 Mechanical and Modular VC Generation for Object-Based Software

In 2011, Heather Harton conceived fully mechanizable proof rules for RESOLVE language constructs and developed an automated verification condition (VC) generator, a key subsystem of the RESOLVE verification system. Harton's work [24] leverages the principles outlined in Krone's work and uses goal-directed proof rules that are sound, complete, and amenable to automation for a language with clean semantics. Harton's VC generator has been intergrated into a Web IDE for RESOLVE and has been used extensively in CS education [10, 12, 11, 21].

The RESOLVE VC generator mechanically produces a collection of mathematical assertions that correspond to the correctness of the code. VCs are generated from annotated code (e.g., loops with invariants and progress metrics) using specifications and proof rules. The VC generator also incorporates simplification of generated VCs whenever possible. VC simplification is a topic investigated further in work by Sun [61] discussed in the next section.

3.1.5 Specification and Mechanical Verification of Performance Profiles of Software Components

Nighat Yasmin's work in [67] focuses on performance verification for component-based software. She extended the RESOLVE language by introducing performance specifications (profiles) layered on top of the existing functional specification language, and extended the formal RESOLVE proof system with necessary augmentations to support performance verification. The performance profiles only include duration (timing), but the results can be extended to displacement (space) bounds.

Yasmin's work provides a set of mechanizable proof rules usable in the performance VC

generation process and presents a prototype implementation of a performance VC generator for experimentation. The implementation only considers execution time performance profiles. Space constraints are left for a future extension of his work.

3.1.6 Towards Automated Verification of Object-Based Software With Reference Behaviour

Yu-Shan Sun's work focuses on automated verification of object-based software with (and without) reference behavior [61]. The work is motivated because not all reasoning about reference behavior is avoidable. Sun's work captures unavoidable acyclic reference behavior using automation-friendly abstractions. The work includes specified and implemented components encapsulating reference behavior where objects share a global state.

A specification and verification system that handles shared states among objects has been developed for experimentation. A VC generator employing sequent-based logical reduction rules that produce more simplified (parsimonious) VCs for verification has also been developed. The VC generator is currently not integrated into the current online RESOLVE verifying compiler that uses an automated prover designed and developed by Hampton Smith [58] to discharge VCs. Smith's work is summarized in 3.2.2.

3.1.7 Scaling Up Automated Verification

The most recent work on RESOLVE verification was by Daniel Welch in 2019 [66]. Welch investigated complexities for developers in specifying component-based software and writing annotations for implementations. The process is made easier through a Formalization Integrated Development Environment (F-IDE) developed to support the formal specification and automated verification of object-based software. The F-IDE is user-friendly, and it includes features to support specification writing, especially for complex software components. Some of the features include responsive editing, assistance for design-by-contract, the interplay of multiple artifacts, and the use of new mathematical models to specify object-based interfaces. Welch's work includes a non-trivial component-based case study involving multiple theory, specification, and implementation units.

3.2 Prior RESOLVE Work on Automated Provers

This section discusses two existing RESOLVE automated provers as a precursor to the design of the prover in this dissertation. The first prover is the SplitDecision procedure discussed in Section 3.2.1. A general, minimalist automated prover is discussed in Section 3.2.2. Both provers are based on a thesis that verifying a well-engineered and well-specified software accompanied by expressive and extensible mathematical library results in VCs that are straightforward for verification [31].

3.2.1 RESOLVE SplitDecision Procedure

The SplitDecision procedure either proves the generated VCs or simplifies them for another automated prover [1]. SplitDecision procedure was designed from the ground up and developed using RESOLVE/C++ [28] at The Ohio State University. The version of RESOLVE used, unlike the one in this dissertation, allows and uses only a fixed number of mathematical theories. initial prototype worked with only a subset of string theory, one of RESOLVE's commonly used theories. Limiting the SplitDecision procedure to string theory allowed more optimizations tailored to make it fast and capable of proving more VCs from programs involving strings. The earlier experiments showed SplitDecision is faster and more effective in proving the VCs compared to the in-house RESOLVE automated prover (that is not specialized for any theories) and Isabelle [50].

The initial success in the SplitDecision procedure led to extensions in its theory domain to handle finite sets, tuples, and integer theory. Such extensions come with substantial development costs, given that decision procedures are developed for a fixed set of theories. While SplitDecision procedure is efficient and fast, it cannot work with specificationally rich languages where mathematical theories are not fixed.

3.2.2 A Minimalist Automated Prover

The most recent version of the RESOLVE prover developed at Clemson University is a minimalist automated prover [58]. This rewrite prover is flexible compared to SplitDecision procedure presented earlier. Its ability to work with an extensible mathematical library broadens the types of VCs it can prove, and it is a first prototype on the efforts to develop a general prover.

The minimalist prover was developed to establish that for software designed for verification, many VCs are straightforward to prove. While the prover functions well with the collection of theories

that occur in components commonly used as classroom examples, it' approach cannot effectively scale for sophisticated components, such as ones presented in [66] and the navigable tree data abstraction in this dissertation. The techniques incorporated are not uniform in how they perform on different theorems, and many more rewriting steps may be required to conclude the correctness of more challenging VCs. The non-uniform performance causes the minimalist prover fail to find proofs even when they exist.

Additionally, the minimalist prover is designed to work with the previous version of the RESOLVE VC generator, which did not produce VCs in a sequent form. Since then, more development has been done to the verifying compiler. The current version produces more simplified VCs in sequent form [61].

The Uni-Prover designed in this work is not based on decision procedures, leverages VCs in sequent form, and employs effective strategies that reduce the number of steps taken to verify target sequent VCs. Critically, it provides a uniform and effective performance when dealing with arbitrary theories.

Table (3.1) summarizes the characteristics of three automated provers in the RESOLVE literature. SplitDecision contains decision procedures for a fixed set of mathematical theories, and hence, does not support a specificationally rich language. Its verification scope is also theory limited. Minimalist prover works with an extensible math library. However, it works effectively only with a set of theories routinely used in classroom examples, and its performance does not scale uniformly to be effective for new theories. Its verification capability is thus limited under reasonable resource constraints. The Uni-Prover supports an extensible math library, higher-order specification, and specificationally rich languages, and has been designed to be effective. Its verification scope is not as limited compared to Minimalist Rewrite Prover.

Table (3.1) Comparison of Automated RESOLVE Provers

RESOLVE Prover	**Supports Extensible Mathematical Library**	**Supports Higher Order Specification**	**Supports Specification-ally Rich Language**	**Provable VCs within Resource Bounds**
SplitDecision	No	No	No	Theory Limited
Minimalist Rewrite Prover	Yes	Yes*	Yes	Limited
Uni-Prover	**Yes**	**Yes***	**Yes**	**Not as Limited**

3.3 A Summary of the RESOLVE Framework

The RESOLVE verification system is composed of several subsystems presented in Section 3.3.1. Salient features of the language are discussed in Section 3.3.2.

3.3.1 The RESOLVE Verification System Architecture

Figure (3.1) shows the current RESOLVE verification system architecture [61]. On the left is a software specialist who writes component specifications using preferred mathematical models and notations that support automated verification. They also annotate their code with assertions such as loop invariants, progress metrics, representation invariants, and abstraction relations for verification.

The specifications and assertive code are then provided to a sequent VC generator, which uses a set of modular proof rules to generate a collection of mathematical assertions (sequent VCs) that correspond to the code's correctness. Harton developed the first VC generator, as discussed in Section 3.1.4. The most recent development is discussed in Section 3.1.6.

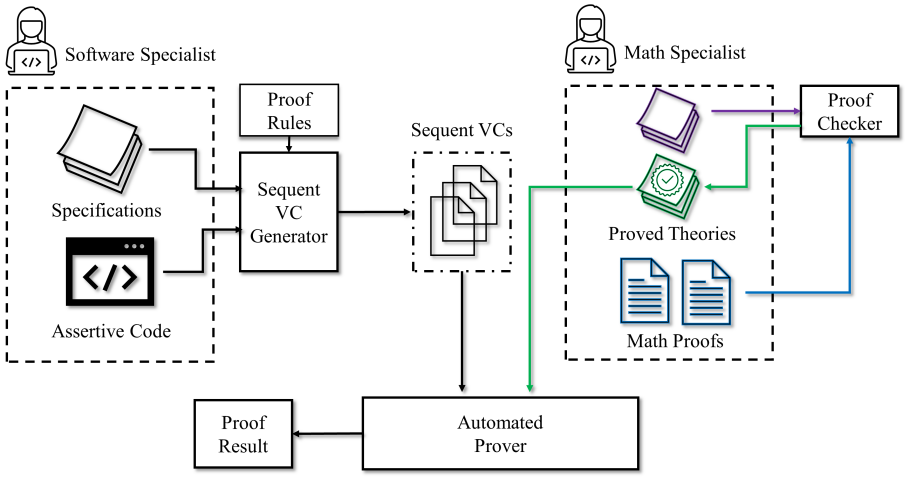


Figure (3.1) RESOLVE verification system architecture

The automated prover must establish the correctness of all sequent VCs before declaring the code is valid. Verifying some of the sequent VCs may require the prover to draw mathematical theorems from the library to assist with the verification process. The theorems are from the theories

developed by a math specialist who writes sound and sufficiently complete theories. The theories must be supported by proofs done once with human assistance and confirmed by a proof checker[59]. The prover will only consider proof-checked theories relevant to the domain of the annotated code, which means the theories must be developed in a modular form to achieve finer granularity.

3.3.2 The RESOLVE Language and Its Salient Features

RESOLVE is a rich language that supports programmers in specifying precisely and concisely software behavior and writing their implementations. A detailed description of RESOLVE's key features may be found in [37]. The features are summarized below.

First, RESOLVE integrates specification and programming languages but keeps the two languages independent of each other. The separation allows the specifications to be entirely mathematical, distinguishing RESOLVE language from many of its counterparts summarized in Section 2.3.

Second, the RESOLVE system incorporates an extensible mathematical library, allowing arbitrary mathematical theories and theory extensions to be added. This feature is necessary to make the specification language rich enough to write precise abstract specifications and verify a broad spectrum of programs.

Finally, the RESOLVE language has clean semantics [36] to deal with the unwanted effects of aliasing due to uncontrolled referencing and mutation, which in turn complicate reasoning. With clean semantics, the program's state and its effect on the variables are kept local to ensure that only values of explicitly mentioned list of state-space variables can change.

3.4 Specifying and Implementing a Stack Component

This section presents a concrete example of a specified and implemented component using RESOLVE language to illustrate the features described in Section 3.3.2.

3.4.1 A RESOLVE Component Specification

RESOLVE concepts are interfaces with specifications that formally describe the functional behavior of components. We present an example concept named `Stack_Template`, which describes

the behavior of stacks and their usable operations. A specification of `Stack_Template` is presented in Listing (1), followed by a discussion on key RESOLVE language constructs used in the specification.

```
Concept Stack_Template(type Entry; evaluates Max_Depth: Integer);
        uses String_Theory, Integer_Ext_Theory;
        requires 1 <= Max_Depth;
    Type Family Stack is modeled by Str(Entry);
        exemplar S;
        constraint |S| <= Max_Depth;
        initialization ensures S = Empty_String;
    end;
    Operation Push(alters E: Entry; updates S: Stack);
        requires 1 + |S| <= Max_Depth;
        ensures  S = <#E> o #S;
    Operation Pop(replaces R: Entry; updates S: Stack);
        requires 1 <= |S|;
        ensures #S = <R> o S;
    -- remaining operations omitted for brevity --
end;
```

Listing (1) An abstract specification for a Stack data structure

The stack concept in Listing (1) is parameterized with a generic type `Entry` and an integer value `Max_Depth`, which allows a stack of any valid type and size to be created. `Max_Depth` is preceded by **`evaluates`**, a parameter mode specifying its additional properties. In this case, **`evaluates`** mode specifies that `Max_Depth` can be an expression that evaluates to an integer value, including a constant.

Because theories used in the specification of the concept are kept separate (which emphasizes the modular approach used in RESOLVE [60]), the **`uses`** statement gives the concept access to the theories that are available for writing specifications and for verifying VC's. In `Stack_Template`, `String_Theory` and `Integer_Ext_Theory` are imported.

`Stack_Template` is generic and can be instantiated to create a stack of any particular type. RESOLVE uses the keyword **`Type Family`** to specify that the concept exports a family of types `Stack` modeled as a string of `Entry`. An **`exemplar`** keyword is used to introduce an example stack `S`

used in the subsequent assertions specifying more properties on a created stack. The first assertion **constraint** bounds the length of a stack S to Max_Depth, and the second assertion **initialization** specifies an initial value of every Stack S variable is an empty string.

Operations in the concept are specified using an optional **requires** clause (pre-condition) and an **ensures** clause (post-condition). As a part of specifications, operation parameters are preceded by modes. In principle, parameter modes simplify specifications by eliminating assertions describing the parameters in the **requires** and **ensures** clause. During verification, proof obligations are generated for each parameter mode. A list of parameter modes used in specifying Stack_Template are presented in Figure (3.2).

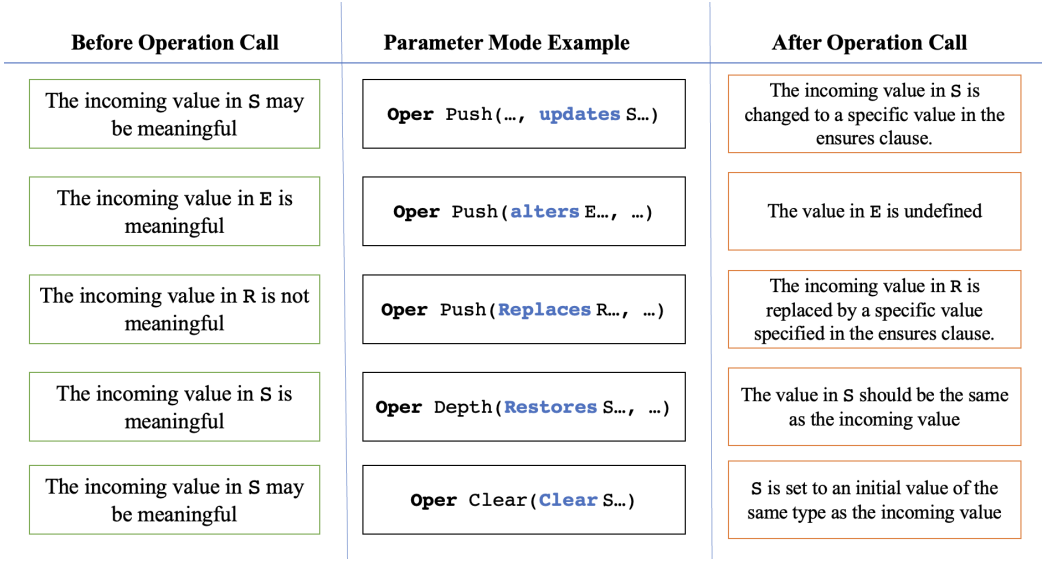


Figure (3.2) A list of parameter modes used in Stack_Template

The specifications for each operation in the concept serve as a contract between the client and an implementer. In the **requires** clause, responsibilities for the client are specified. The client must fulfill all specified obligations before calling the operation. For example, an operation Push in the Stack_Template adds a new entry into the stack. The **requires** clause, therefore, restricts the operation from being called when the stack is full. The client must ensure that the stack has room for at least one new entry before using operation Push. Mathematically, the restriction is stated in the **requires** clause as $1 + |S| \leq$ Max_Depth. The vertical bars around S is a string length operator defined in the string theory.

An implementer uses the specified behavior in the **ensures** clause to write code that achieves

the operation's goal. Mathematically, it is stated in the **ensures** clause as S = ⟨#E⟩ ∘ #S. Because it is often necessary to refer to both input and output values of parameters in the **ensures** clause, a # sign is used as a prefix to indicate the incoming value of a parameter before the operation call. Its use in the assertion S = ⟨#E⟩ ∘ #S means that the outgoing value of stack S is a concatenation of an incoming entry value (#E) and incoming stack (#S). The string concatenation operator ∘ and a singleton string constructor operator ⟨...⟩ are defined in string theory.

3.4.2 A RESOLVE Concept Implementation

A specified component can be implemented in multiple ways for different performance trade-offs. It is a modular approach used in RESOLVE, allowing clients to rely solely on specifications to use components and simplify their reasoning. This section describes how a concept is implemented in RESOLVE. In particular, we present an array implementation of `Stack_Template` in Listing (2).

```
Realization Array_Realiz for Stack_Template;
                            uses Integer_To_String_Function_Theory;
    Type Stack is represented by Record
            Contents: Array 1..Max_Depth of Entry;
            Top: Integer;
        end;
        convention
            0 <= S.Top <= Max_Depth;
        correspondence
            Conc.S = Reverse(Iterated_Concatenation(1, S.Top,
                    Stringify_Z_Entity(S.Contents))); -- Stringed_Z_Entity
    end;
    Procedure Push(alters E: Entry; updates S: Stack);
        S.Top := S.Top + 1;
        E :=: S.Contents[S.Top];
    end;
    Procedure Pop(replaces R: Entry; updates S: Stack);
        R :=: S.Contents[S.Top];
        S.Top := S.Top - 1;
    end;
```

```
        Procedure Depth(restores S: Stack): Integer;
            Depth := S.Top;
        end;
        Procedure Rem_Capacity(restores S: Stack): Integer;
            Rem_Capacity := Max_Depth - S.Top;
        end;
        Procedure Clear(clears S: Stack);
            S.Top := 0;
        end;
    end;
```

Listing (2) Array implementation of a `Stack_Template`

The specification in Listing (1) models a `Stack` mathematically as a string of entries. The implementation can use any data structure that can accommodate the behavior specified for a `Stack` for its representation. In this particular implementation, a `Stack` is represented in the code as a **`Record`** (similar to a struct in C) with two fields. The first field `Contents` is an array of entries with `Max_Depth` as its maximum size. The second field `Top` is an index in the array representing the top of the `Stack`. The **`convention`** clause specifies a realization invariant to restrict the `Top` of the `Stack` within specified bounds. The realization invariant is assumed true before each operation, except for the initialization. It should also be guaranteed to be true after each operation specified in the concept, at the end of initialization, and at the beginning of finalization.

The **`correspondence`** assertion in the RESOLVE realization in Listing 2 is the abstraction relation between the mathematical conceptualization of `Stack` and its representation in the implementation. It provides a mathematical interpretation as an abstract value of an internal representation. The relation has to be well-founded to represent all legitimate values of a `Stack` representation, and the abstraction should fit within the conceptual constraints. The correspondence presented in Listing (2) states that the user's conceptual stack `Conc.S`, which is a mathematical string of entries, is a reverse of concatenated elements in the array `S.Contents` from the first position to `S.Top` in the internal representation.

Each operation specified in the `Stack_Template` must be implemented by writing a procedure. All procedures in Listing (2) use the RESOLVE swap operator denoted by a symbol :=:. The swap operator is used to exchange values in two variables without copying their content, and

it is available for all programming types. It allows efficient movement of large arbitrary structures in constant time and without introducing aliasing [23]. If and when necessary, a copy operation can be imported explicitly.

The first procedure implements `Push` operation by incrementing `S.Top` to point to the next empty slot in the content stack `S.Contents` and then swapping the value in `E` into the content stack at position `S.Top`. The second procedure implements the operation `Pop`, and uses the swap operation to swap out the value at the top of the contents stack. Because the top value is removed from the content stack, `S.Top` is decremented by one to point to the top occupied index. For brevity, other procedures to complete the realization for the stack template are just included in Listing (2).

3.4.3 Enhancements

Only orthogonal and efficiently realizable primary operations are specified in the core concept to prevent specifications from becoming unwieldy. Any other helpful operation implemented using primary operations can be specified and implemented as a secondary operation.

In RESOLVE, a specification inheritance mechanism, called an "enhancement," permits a straightforward extension of a concept. An example enhancement operation for `Stack_Template`, one to flip a stack, is specified and realized in Listing (3).

```
Enhancement Flipping_Capability for Stack_Template;
    Operation Flip(updates S: Stack);
        ensures S = Reverse(#S);
end Flipping_Capability;
Realization Flipping_Realiz for Flipping_Capability of Stack_Template;
    Procedure Flip(updates S: Stack);
        Var Temp: Stack;
        Var Next_Entry: Entry;
        While ( 1 <= Depth(S) )
            maintaining #S = Reverse(Temp) o S;
            decreasing |S|;
        do
            Pop(Next_Entry, S);
            Push(Next_Entry, Temp);
        end;
```

```
            Temp :=: S;
        end Flip;
    end Flipping_Realiz;
```

Listing (3) Flippling capability enhancement specification and realization

The `Flipping_Capability` enhancement is specified with just an **`ensures`** clause, which guarantees a reversed stack at the end of an operation call. It uses a string reversal operator from string theory. Its implementation employs a while loop to pop the top element in the stack and push it onto a temporary local stack (`Temp`) iteratively. The while loop is annotated with an invariant in the **`maintaining`** clause, and a progress metric in a **`decreasing`** clause to support automated verification.

Assertions like invariants and progress metrics are the only assistance the programmer provides to the verification. No further interactions are needed during the verification process. All practical automated verification systems, including RESOLVE, demand these assertions be included in the code for automated verification [56].

Chapter 4

Uni-Prover Overview

This chapter presents an overview of the Uni-Prover and the steps it takes to verify sequent VCs. Unlike many existing provers described in the previous two chapters, this prover is designed for verification of atomic sequent VCs. The chapter begins with a summary of sequent VC basics.

4.1 RESOLVE Sequent VC Basics

Verification of correctness for the RESOLVE implementation involves the generation of Verification Conditions (VCs) as described in Section 3.3.1. VCs are proof obligations that are necessary and sufficient to establish code correctness.

Each VC is a logical statement with two main parts, **goals** and **givens**. In each part of the VC are ground clauses without any logical connectors. Currently, the RESOLVE VC generator produces VCs in a sequent form, as discussed in Sun's work [61]. They can be naturally reinterpreted as ground clause atomic sequents in the form of $\Gamma \Rrightarrow \Delta$, where Γ denotes a set of positive ground clauses called antecedents (**givens**) and Δ denotes a set of positive ground clauses called succedents(**goals**).

The arrow ($\Rrightarrow$) joining the two sides of the sequent is a set implication arrow, and it means that the *conjunct* of all clauses in the antecedent implies the *disjunct* of clauses in the succedent. Givens are conjoined and goals are disjoined, so the sequent is true iff one of the goals is provable using all of the givens.

Because the VCs generated in RESOLVE are naturally represented as sequents, we use the term "sequent VCs" throughout this work. A general representation of a sequent VC is shown in

(3.1), where ground clauses $\mathtt{A}_1, \mathtt{A}_2 ... \mathtt{A}_\mathtt{m}$ are the antecedents and $\mathtt{S}_1, \mathtt{S}_2 ... \mathtt{S}_\mathtt{n}$ are succedents.

$$\underbrace{\{\mathtt{A}_1, \mathtt{A}_2, ..., \mathtt{A}_\mathtt{m}\}}_{\Gamma} \Rrightarrow \underbrace{\{\mathtt{S}_1, \mathtt{S}_2, ..., \mathtt{S}_\mathtt{n}\}}_{\Delta} \tag{3.1}$$

To demonstrate how close the generated VC in RESOLVE is to a sequent, in Listing (4) is an example VC with three ground clauses as givens and one ground clause as a goal. This particular VC is generated from verification of code for a Stack Flip enhancement in Listing 3. The symbol $\circ$ is used for concatenation of two strings; a superscript `Rev` is a string reversal operator; the pair of angled brackets $\langle\rangle$ is a stringify operator; and Λ is an empty string. All these notations and operators are defined in string theory.

Goals:

$\mathtt{S} = (\langle \mathtt{E} \rangle \circ \mathtt{T})^{\mathtt{Rev}} \circ \mathtt{U}$

Givens:

1. $\mathtt{S} = \mathtt{T}^{\mathtt{Rev}} \circ \mathtt{R}$
2. $\mathtt{V} = \langle \mathtt{E} \rangle \circ \mathtt{U}$
3. $\mathtt{V} = \mathtt{R} \circ \Lambda$

Listing (4) A sample generated VC in RESOLVE

The ground clause sequent VC representing the VC in Listing (4) is shown in 3.2 below. The conversion from the VC to sequent VC is straightforward. All givens in the VC become the antecedents, and goals become succedents.

$$\{\mathtt{S} = \mathtt{T}^{\mathtt{Rev}} \circ \mathtt{R}, \mathtt{V} = \langle \mathtt{E} \rangle \circ \mathtt{U}, \mathtt{V} = \mathtt{R} \circ \Lambda\} \Rrightarrow \{\mathtt{S} = (\langle \mathtt{E} \rangle \circ \mathtt{T})^{\mathtt{Rev}} \circ \mathtt{U}\} \tag{3.2}$$

Having VCs in a sequent form allows us to take advantage of the sequent calculus rules to make the proof process more effective using a divide and conquer strategy. The prover in this work applies immediately to systems where VCs are already generated in a sequent format, but the benefits offered by the approach open the opportunity for systems that generate VCs in other formats to consider the direction in [61].

4.2 Central Role of a Congruence Class Registry

The Uni-Prover is designed to verify sequent VCs by optimizing the number of verification steps necessary to establish correctness. In this work, we have designed, specified, and implemented a prototype of its central component, the congruence class registry. The registry handles equalities effectively (refer to Section 4.3.5) and works with a contiguous instantiation strategy that requires neither user-supplied heuristics nor triggers when instantiating universally quantified theorems in any theory.

The congruence class registry stores a sequent VC that is to be verified in congruence classes and effectively handles all the manipulations necessary to establish that the sequent VC is correct. Figure 4.1 illustrates the effectiveness of the Registry in storing the sequent VC. On the left is the client side, including the initial target sequent VC to be proved. The entailed sequent accounts for equalities in the antecedent and contains all clauses created as a consequence of applying those equalities, making the entailed sequent very large. Additionally, the sequent continues to grow as theorems are applied.

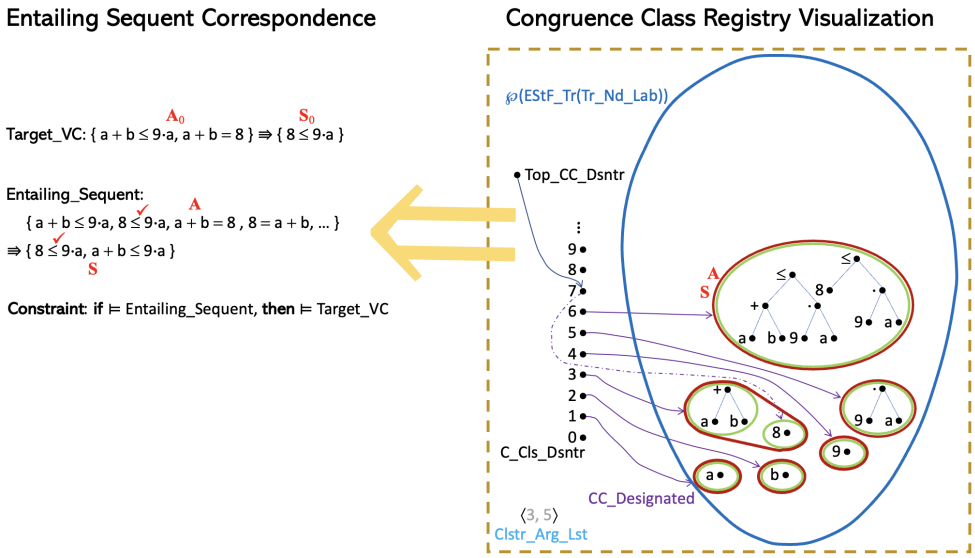


Figure (4.1) Conceptual visualization of a sequent VC in the Registry

The entailing sequent VC on the left is illustrated in the Congruence Class Registry on the right conceptually. The Registry is designed and specified such that an actual realization could store the information economically, with minimal replication such as of the ground terms and trees. In this sense, the duplication of subtrees in the illustration is strictly to ease *conceptual* understanding.

The Registry design makes it possible to fold together the antecedents and succedents. It uses congruence classes to deal with equality and allows the entailing sequent to be kept in a compressed form on each verification step. The two sides are related through a correspondence, and the sequent VC in the registry can be re-interpreted to the largely entailing sequent through it. The correspondence is indicated by the large arrow pointing from the registry.

The compression achieved by an implementation of the registry is central for effective verification, though at the conceptual level this compression is not visible in the illustration here or in subsequent ones in this Chapter. Compression details may be found in Chapter 10.

The verification process starts with the registration of the sequent VC. We explain the sequent registration process using an example in section 4.3. Using sequent calculus, the sequent VC can be declared correct if one clause registered from the succedent ends up in the same congruence class as another clause registered from the antecedent, meaning that a given matches a goal that proves the VC. Some sequent VCs are trivial, direct result of simple logic rules, and immediately provable. Therefore, once the registration is complete, those sequent VCs can be determined if they are correct. Many other sequent VCs involve drawing theorems from the mathematical library to support the verification process by enriching the sequent VC under verification with more antecedents or succedents.

The following section describes how a sequent VC is registered and proved without involving theorems. The illustrations shows step by step how the Registry stores the sequent VC shown in Figure (4.1). Section 4.4 explains how theorems are involved in the verification process as and when necessary.

4.3 Sequent VC Registration in a Congruence Class Registry

Figure (4.2) shows an example sequent VC presented in two forms. On the left is a traditional format using a set implication arrow. On the right is a nested list form, which is easy to process through a position indicated by the red arrow. A RESOLVE nested list component with all necessary operations for processing a sequent VC is provided in appendix E. From here onwards, we will use A to indicate the antecedent side of the sequent VC and S for succedent.

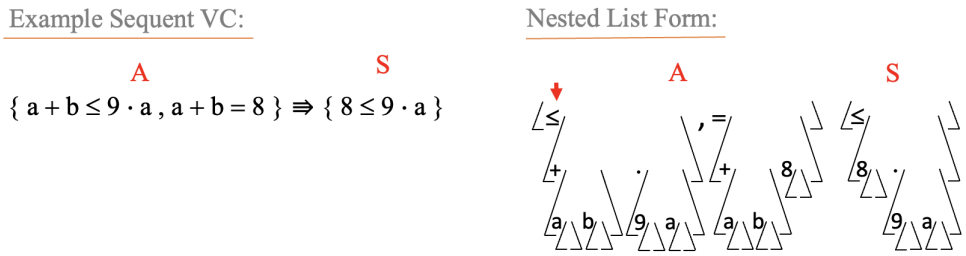


Figure (4.2) Two forms representing a sequent VC

4.3.1 Registering Variables and Constants

Before a sequent can be registered, elements that constitute the sequent must be registered. The congruence class registry concept specifies the operations necessary for a client to achieve the intended functionality. An implementer uses the concept to write realizations that achieve the specified behavior. The diagrams used in this explanation, as in Figure (4.3), have two sides. The client-side on the left illustrates the use of specified operations to register or search the registry. On the right is a visual representation of the congruence class registry (`CC_Reg`) illustrating how the called operations change its state. The right side still uses terms in the concept just for generality in our explanations, leaving a more specific discussion of an implementation to Chapter 10. The following is a discussion on how to register variables and constants in sequent VCs.

A clause in a sequent VC is registered by processing the nested list bottom-up starting with variables and constants followed by the operators. The list position is advanced until variables and constants are found and registered before pulling back to an outer list where operators are placed as they would be in a syntax tree.

Suppose the client first registers the left-most clause and the left-most term. For the sequent VC in Figure (4.2), the client must advance the position to the list labeled `a`, which is registered first. The registration of a variable `a` is shown in Figure (4.3).

The congruence class registry starts empty and gets populated as a sequent VC is registered. For each registered label from the sequent VC, a congruence class (shown in red) and a cluster (shown in green) are created. A cluster is a subclass that groups together all trees in the congruence class that have similar root node label. Initially, every class will only have one cluster. But they may contain more than one cluster, as more trees get added into the congruence class. The class and cluster currently being registered are shown in dotted circles, and those previously added are shown

in solid circles. The organization of trees in congruence classes and clusters is to ensure effective searching as described in a later Section 4.5.

Each created cluster has a label and an argument list containing congruence classes for the arguments. The cluster argument list (`Clstr_Arg`) holds the arguments relevant to the registered node label. The `Clstr_Arg` list is shown in Figure (4.3) on the bottom left of the registry. Variable and constants have zero arguments, and therefore, the cluster argument string is empty during their registration. However, when operators are registered, their arguments are appended to the `Clstr_Arg` list using registry operations.

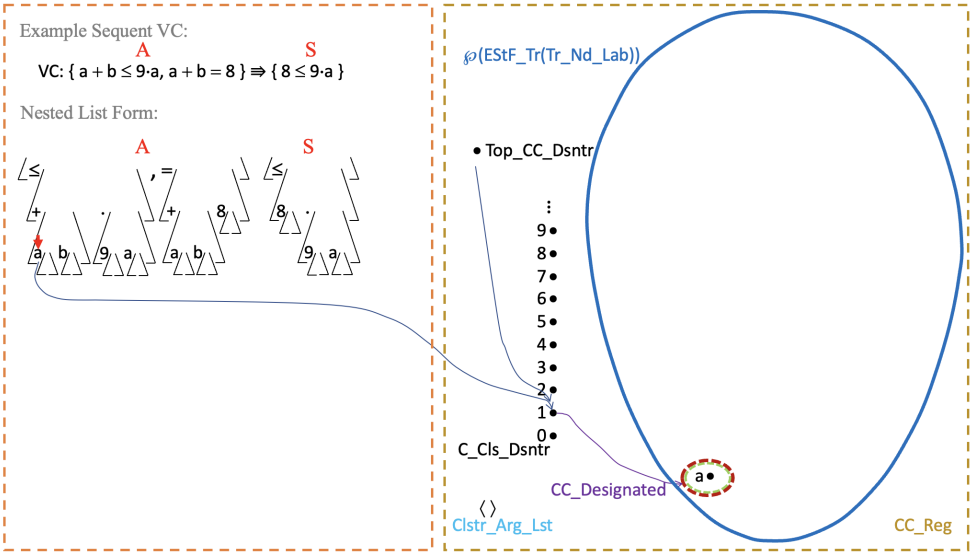


Figure (4.3) Registering a variable a into the registry

Every created class and cluster is designated by a unique number in the registry. Congruence classes have Congruence Class Designators (`C_Cls_Dsntr`), and clusters have Cluster Designators (`Clstr_Dsntr`). Only congruence class designators are included in the diagrams used for the discussion in this section. More details about cluster designators are found in section 4.5.

The first created class for a variable a is assigned a class designator of `1`. The designator is found after incrementing `Top_CC_Dsntr` that keeps track of the most recently used designator. Each class designator is mapped to its respective class through Congruence Class Designated (`CC_Designated`) function. Once a class is created, a client is provided with an accessor to the class. In Figure (4.3), the accessor to the created class is represented by the arrow from the label on the client-side to the class in the registry.

The variable b is registered next using exact same process above. A new cluster and class are created in the registry as shown in Figure (4.4). As for a, b also has no arguments. Therefore, `Clstr_Arg_Lst` is empty.

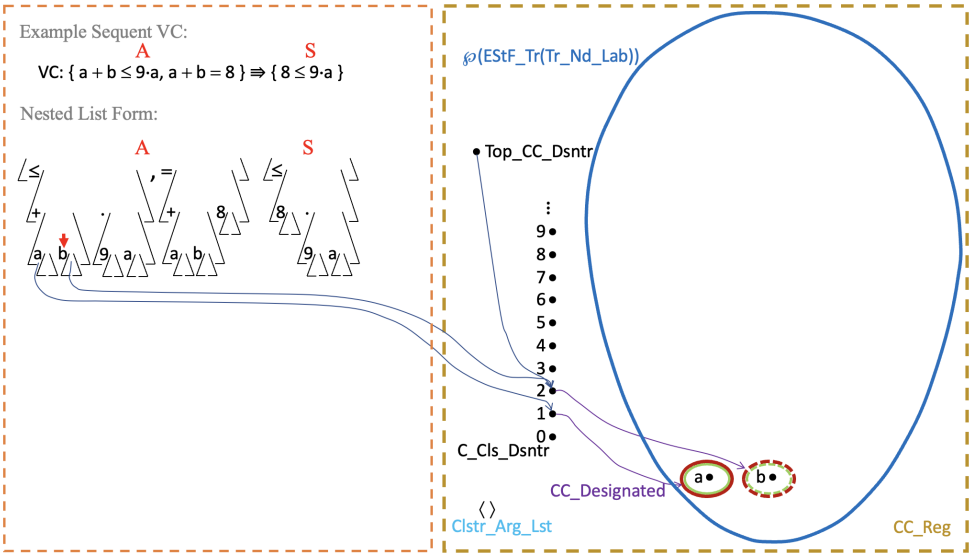


Figure (4.4) Registering a variable *b* into the registry

4.3.2 Registering a Cluster With Arguments

Now that `a` and `b` are registered, a + operator on the immediate outer list (see the position arrow in Figure (4.5)) can be registered. In contrast to variables and constants, the + operator has arguments. The congruence class designators for `a` and `b` are appended to the cluster argument list (`Clstr_Arg_Lst`) before a class and cluster for the + operator are created. In Figure (4.5), the created class for the + operator is designated by `3`, which is now our `Top_CC_Dsntr`.

4.3.3 Registering an Existing Cluster

For effective searching, the registry *does not store any duplicates*, as we demonstrate in Figure 4.6 when a subexpression $9 \cdot a$ is registered. Because a constant `9` is not yet in the registry, a new cluster and congruence class are created and assigned `4` as a designator. However, no registration happens when we get to variable `a`, as it already exists in the registry. Even though variable `a` is registered, the client must retrieve its accessor and append it to the argument list because `a` is one of the arguments for the dot operator.

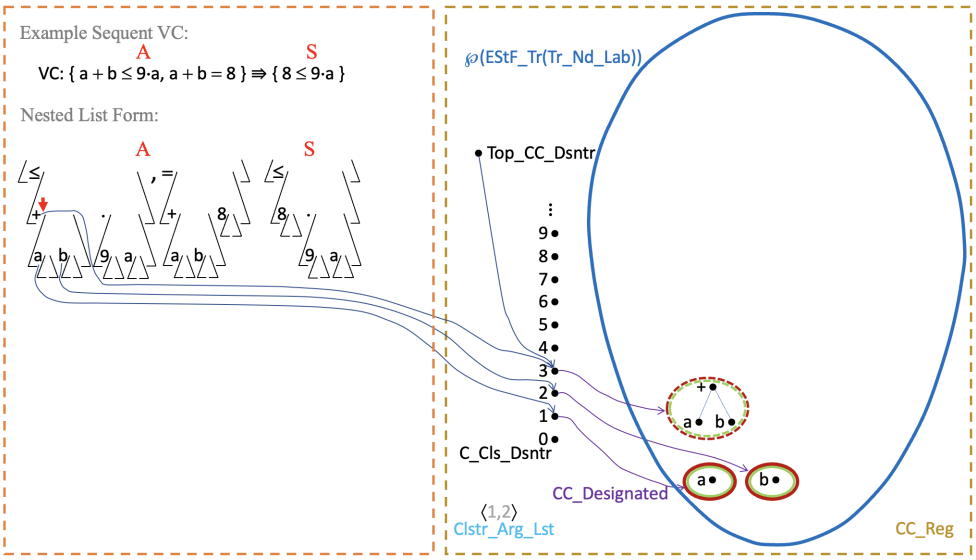


Figure (4.5) Registering an addition operator with a and b as arguments

4.3.4 Top-Level Class Registration

Now that we have registered both `a + b` and `9 · a`, the sequent clause `a + b ≤ 9 · a` can ultimately be registered. At the root of this clause is the operator ≤, and its registration requires congruence classes `3` and `5` as arguments. Figure (4.7) shows a new class and cluster created for the operator ≤. At this point, the registration for the clause is complete. This is a "top level" class, and an attribute A for antecedent is added to indicate which side of the sequent VC the registered clause came from. The role of this information is to determine the correctness of the sequent VC as we explain in the subsequent sections. Top-level classes for clauses from the succedent are supplied with S as an attribute.

All low-level classes are supplied with a default attribute determined by the client. The default attribute are not shown in the illustrations used in explaining the registration process in this chapter. Default attribute is explained further in Section 9.1.

4.3.5 Registering an Equality in the Antecedent

Equalities must be handled differently from other predicates for performance reasons. In the registry, they are handled through congruence classes, where classes with trees known to be equal are merged into a single class. For example, the equality predicate `a + b = 8` in our example sequent VC is recorded by merging the congruence class for `a + b` and the class for `8`. The registration process

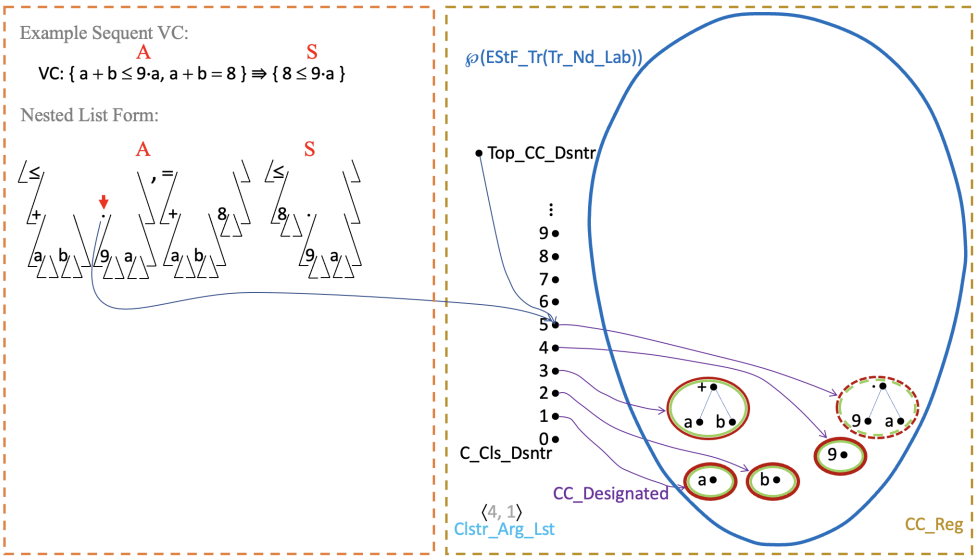


Figure (4.6) Registering a ground term `9 · a`

follows the usual bottom-up approach, and only a constant `8` needs to be registered, as the term `a + b` in `a + b = 8` is already registered. The registration of `8` is shown in Figure (4.8).

With `a + b` and `8` registered, their classes are merged into a single class when we get to the registration of = operator. The state of the registry after merging the two classes is shown in Figure (4.9). The new class is assigned the minimum of two designators `3` and `7` of the merged classes. The designator `7` also points to the newly created class as the client still holds its accessor. Thus, if the client uses `7` at any point, it is going to access the new class. This design ensures a clean separation between accessors and designators, which is critical in reasoning, as explained in section 7.3.1.

In most cases, as a consequence of having two classes merged, a cascade of updates on other congruence classes follows. After the updates, all congruence classes in the registry with trees containing either `a + b` or `8` will now include both trees. Figure 4.10 shows this expansion in congruence class `6`. The initial tree with a subtree `a + b` is now equal to a tree with a subtree `8`. The full-scale effect of merging two classes may cause more classes to collapse, resulting in more congruent classes in the registry. While the importance of the collapse is not striking in this simple example, its general cascading impact is at the heart of the registry.

The sequent VC is proved correct in the registry if clauses from both the antecedent and succedent are in the same congruence class. This is concluded from the registry when trees are in the same class tagged by both attributes, A and S. For our example sequent VC, once the succedent

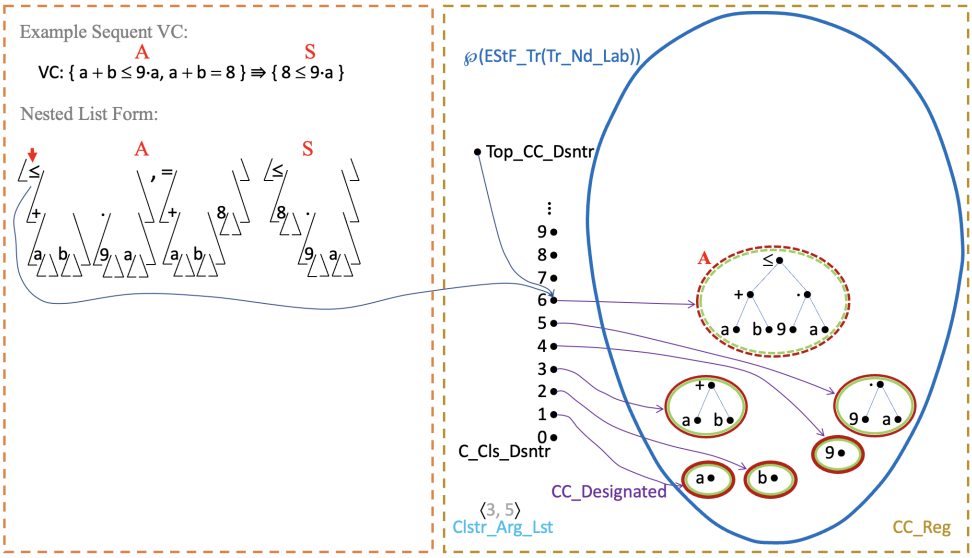


Figure (4.7) Adding attributes to top-level class during registration

clause is ultimately determined to exist in the registry, the succedent attribute is added to class 6, and we can conclude that the sequent VC is correct.

This example illustrates the simpler case in verification where the target sequent VC is immediately provable to be correct given its antecedents. The more general case occurs when the antecedents are insufficient to verify the sequent VC and require an elaboration of mathematical theorems on the sequent VC before establishing its correctness.

4.4 Employing Theorems to Verify Sequent VCs

The theorems are stored in the mathematical library and must be already proven true independently to support verification in a reasonable number of steps. The library houses many theories put together in theory units, and only theorems relevant to the target sequent VC are selected for effectiveness in the verification process. Literature is replete with theorem selection using heuristics, as discussed further in Chapter 5.

Once theorems are selected, they are instantiated and used in the verification process. Several instantiation techniques exist, and a few are described in chapter 5. Some techniques can be costly. For instance, a straightforward instantiation strategy explained in section 5.1.1 can lead to an exponential number of sequents to be proved, in order to prove a given initial sequent VC. It is too costly to use, in general.

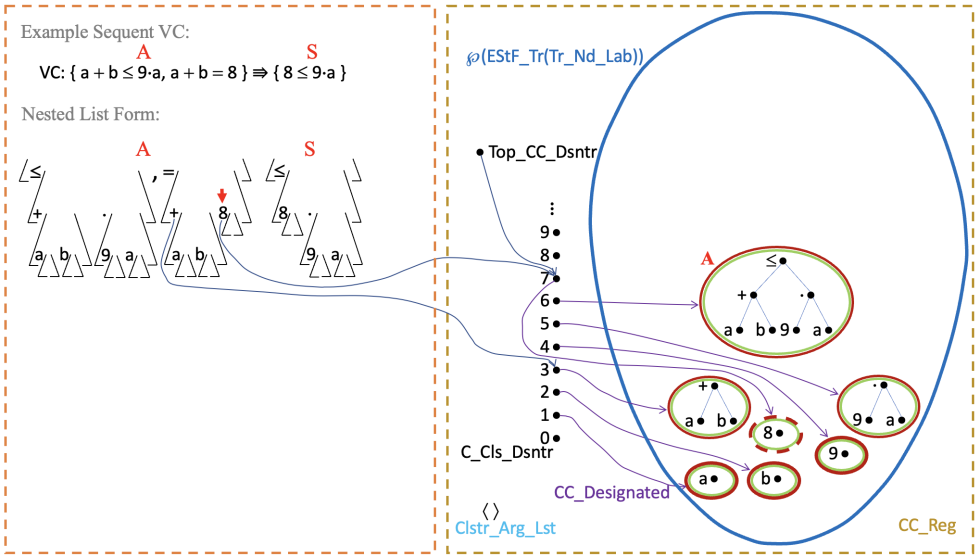


Figure (4.8) Registering a constant *8* into the registry

Other instantiation techniques such as E-matching [14] and those used in [68, 55] are based on heuristics and not applicable in a general case where arbitrary theorems can be conceived. The Uni-Prover is designed to work with arbitrary theorems, and specialized instantiation strategies are inadequate.

We employ contiguous instantiation that does not require human intervention or hints. This strategy is described in detail in Section 5.2. It works with elaboration rules generated from the relevant theorems and involves counter-matching clauses in the rules to those in the registry, a process that is central to achieving contiguous instantiation and avoiding one sequent exploding into many. The following subsection discusses the idea of elaboration rules and how contiguous instantiation employs a counter-matching process for effectiveness.

4.4.1 Contiguous Instantiation Strategy for Uni-Prover

Elaboration rules are created to ensure only one instantiated clause of the theorem is introduced in the sequent VC being verified, making the instantiation effective by ensuring only one sequent is processed for verification.

Elaboration rules are generated automatically from theorems written in Universally Disjunctive Form (UDF), for example $\mathtt{T}_1$ in Figure 4.11. In general, if given $\mathtt{UDF}_i = \forall \mathtt{x}_1 : \mathtt{S}, \cdots, \forall \mathtt{x}_j : \mathtt{S}, \mathtt{C}_{1,1} \vee \cdots \vee \mathtt{C}_{i,k}$ with a set of clauses $\{\mathtt{C}_{i,1} \cdots, \mathtt{C}_{i,m}, \cdots, \mathtt{C}_{i,k}\}$. For each determinate clause $\mathtt{C}_{i,m}$ in the set $\{\mathtt{C}_{i,1} \cdots, \mathtt{C}_{i,m}, \cdots, \mathtt{C}_{i,k}\}$,

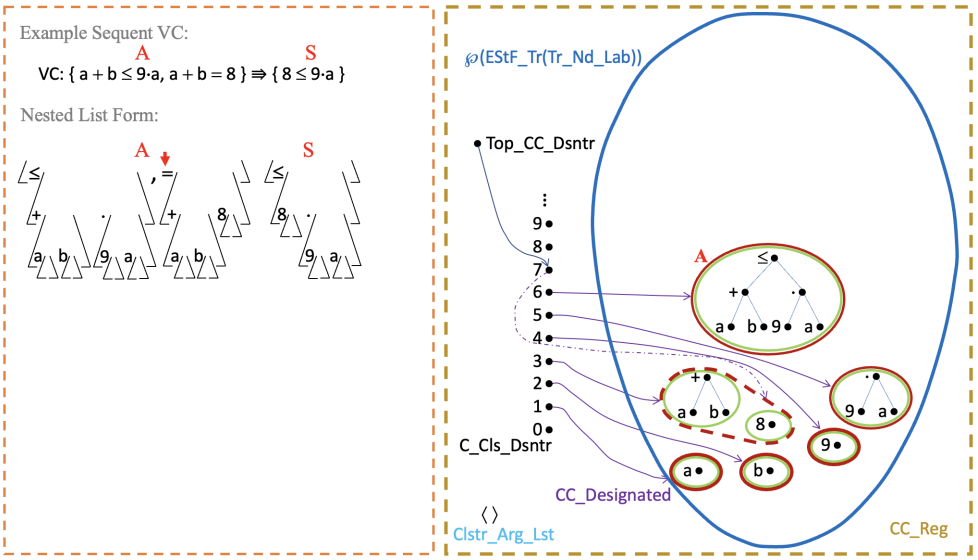


Figure (4.9) Merging two classes known to be equal in the registry

an elaboration rule is created where $\{C_{i,1},\cdots,C_{i,k}\} \sim \{C_{i,m}\}$ are precursor clauses, and $\{C_{i,m}\}$ is a resultant clause. The theorem UDF_i could produce up to k distinct elaboration rules determined by the number of determinate clauses in the theorem.

A theorem of the form $p \implies q$ reduces to the two disjuncts $\neg p \vee q$. In the example theorem T_1 here, there are three disjuncts. So for T_1, three elaboration rules (R_1, R_2, and R_3) are created. An elaboration rule has a precursor part on the left of the arrow and a resultant part on the right. The rules are created by picking one clause at a time to be the resultant clause, and the remaining clauses become precursor clauses. For instance, R_1 has $n \leq m$ as a resultant clause, which makes $m = n$ and $m \leq n$ precursor clauses. The number of elaboration rules created out of a theorem depends on the number of clauses. Details about elaboration rules are presented in Section 5.2.4.

Counter-matching precursor clauses is what drives the contiguous instantiation strategy. A precursor clause is counter-matched if it matches a clause in the sequent VC that is on the side indicated by the attribute on that precursor clause. There are two attributes used, A is for antecedent, and S is for succedent. For example, the first clause in the rule R_1 is tagged with S, which means its counter-match must be in the succedent of the sequent VC. Similarly, the second clause should counter-match a clause in the antecedent. Once all precursor clauses in a rule are counter-matched, an instance of the resultant clause is added to either the antecedents or succedents of the target sequent VC. The attribute attached to the resultant clause determines which side of

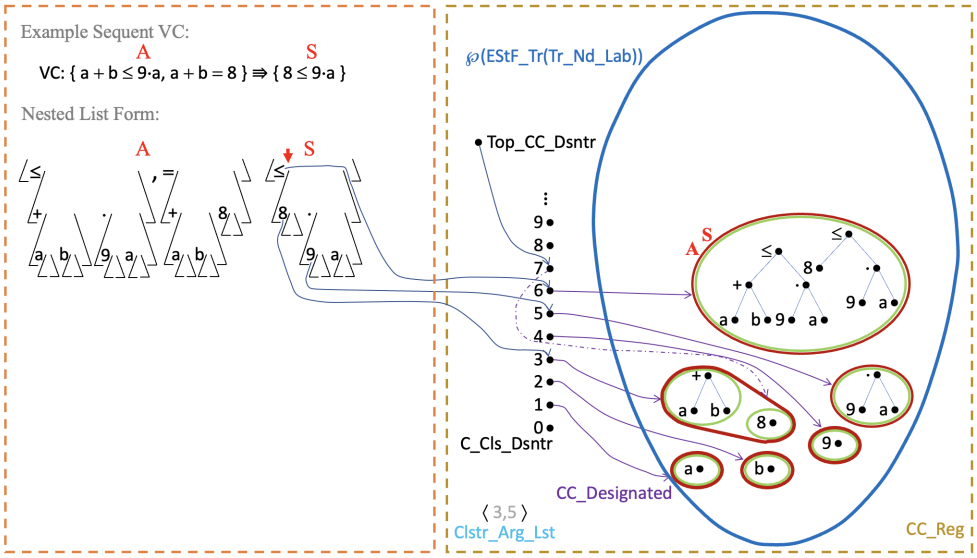


Figure (4.10) Updating class *6* after merging class *3* and *7*

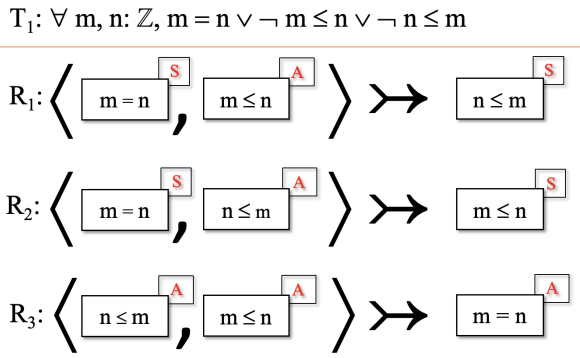


Figure (4.11) Creating elaboration rules from theorem $\mathtt{T}_1$

the target sequent the instance should be added. Details on how clause attributes are decided can be found in section 5.2.6.

Figure 4.12 presents a counter-matching example for a rule $\mathtt{R}_1$ used in the verification of a sequent VC provided above the dotted line. The sequent includes all relevant integer theorems ($\Theta_{\mathbb{Z}}$) necessary to prove its correctness. These theorems are converted to elaboration rules, which are then applied one after the other to elaborate the sequent VC. One of these rules is $\mathtt{R}_1$ from the integer theorem in Figure (4.11). To find if any of the rule's precursor clauses counter-match a clause in the sequent VC, ground terms from the sequent likely to result in a counter-match are selected. In this case, $3 \cdot \mathtt{b}$ and $3 \cdot (\mathtt{a} + 1)$ are selected for `m` and `n` respectively. The substitution results in an instance of the rule $\mathtt{R}_1$ ($\mathtt{IR}_1$). To find if the first clause resulted in a counter-match, we check if there is a

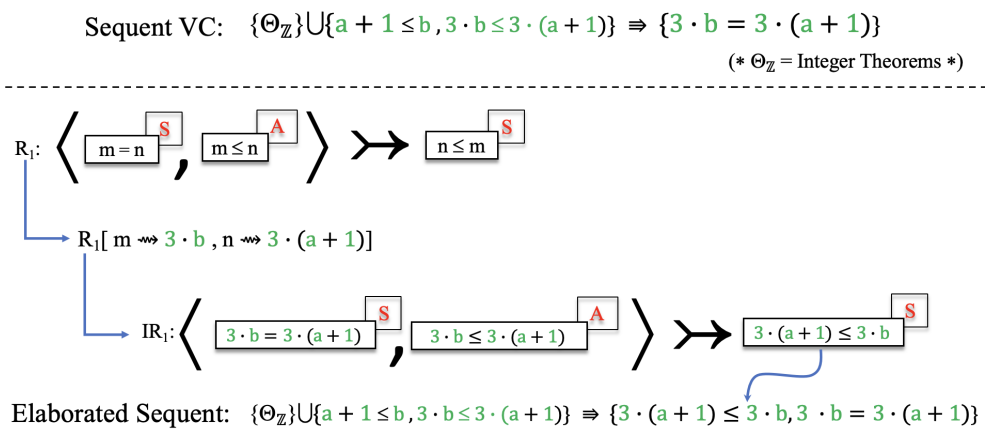


Figure (4.12) Elaboration rule counter-matching example

similar clause in the succedent of the sequent VC, and here, we find one. Similarly, we can find a matching clause in the antecedent of the sequent for the second precursor clause.

Once the precursor clauses in this rule are counter-matched, the next step is to include an instance of the resultant clause on the succedent of the sequent. This step is shown in the elaborated sequent VC in Figure 4.12. The resultant VC is "more" provable, because there is an additional goal in it; recall that the goals is succedent are disjuncts.

4.4.2 Registering a Non-Trivial Congruence Class

In registering clusters, some arguments may consist of simple classes containing one cluster, and others may involve non-trivial classes with more than one cluster. The registration of clusters in earlier sections used arguments with simple classes. This section describes the case involving arguments with non-trivial classes.

We start with a sequent VC registration involving a non-trivial class `3` shown in Figure (4.13), followed by a similar case of how resultant trees are registered in Section 4.4.3. The sequent VC used in this explanation is equivalent to the earlier one, with an equality predicate coming first.

The equality is registered starting from the innermost lists with the variables `a` and `b`. The term `a + b` is then registered followed by a constant `8`. When we get to the = operator, a class for the term `a + b` and for the constant `8` are merged to a single class, as illustrated in Figure (4.13).

The second clause requires the registration of the second term `9 · a` shown in Figure (4.14), before we can ultimately register it. The arguments needed for the registration of ≤ at the root of the second clause are `3` and `6`. While a class `6` is simple, class `3` is considered non-trivial as it

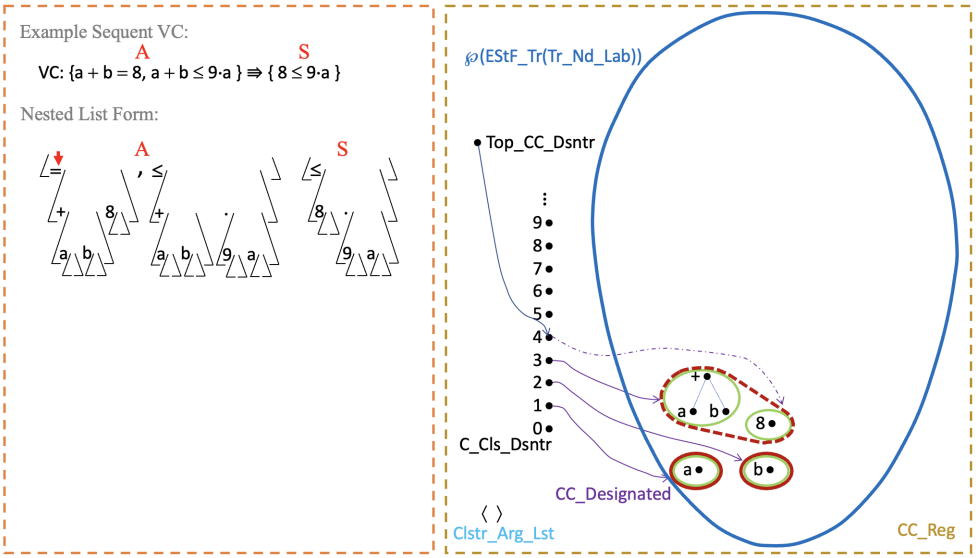


Figure (4.13) Registering equality as the first clause in a sequent VC

contains more than one cluster. Figure (4.15) demonstrates the difference of creating a cluster with a non-trivial class in the argument. The trees formed in the new cluster consist of a combination of all trees existing in the non-trivial class causing an expansion observed when class 3 is used in the argument.

4.4.3 Resultant Clause Registration in a Congruence Class Registry

Figure (4.16) shows the registration of the elaboration rule's resultant clause. On the left is theorem $\mathtt{T}_2$, which is first rewritten in the disjunctive form before being used to create elaboration rules. One of the rules created is $\mathtt{R}_5$. To facilitate searching and registration, $\mathtt{R}_5$ is represented in a tree form. The two trees before the arrow are precursor trees, and the one after is the resultant tree. All precursor trees must be counter-matched for the resultant tree to be registered. A successful counter-match will translate the elaboration rule's constants (0) and variables $(\mathtt{m}, \mathtt{n}, \mathtt{p})$ to registry's congruence classes $(\mathtt{C}_\mathtt{m}, \mathtt{C}_\mathtt{n}, \mathtt{C}_\mathtt{p})$. If the rule is deterministic, constants and variables in the resultant tree will be a subset of those in the precursor tree. Therefore, the classes needed for the resultant clause can be determined easily.

The registration of a resultant tree is similar to sequent registration in section 4.4.2. Non-trivial classes are involved, and a similar expansion happens as a cluster is created. See class 30 in Figure (4.16).

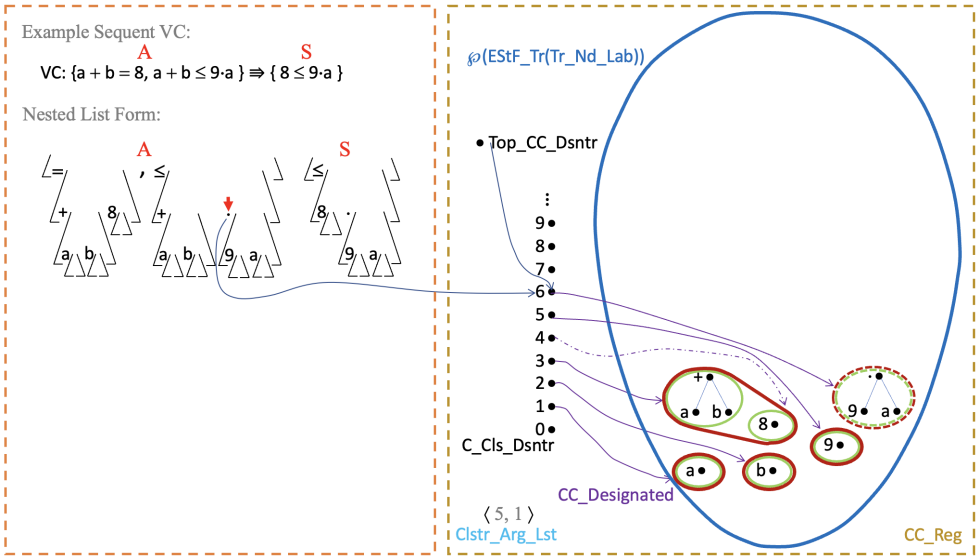


Figure (4.14) Registering a dot operator with 9 and a as arguments

In Figure (4.16), the resultant tree registration begins with the constant `0`. A class `29` is created and becomes our first argument for the ≤ operator. The second argument is a class $\mathtt{C_p}$ designated by `25`. As observed in the diagram, $\mathtt{C_p}$ is non-trivial and contains many trees. The two classes are used as arguments for the new cluster created in class `30`.

For a given rule, all of its matching instances in the registry are considered before moving to the next rule. Not finding a match for a rule does not preclude it from future matches. Therefore, to ensure that each rule is entirely considered in the process, we will effectively cycle through the rules until: (1) the sequent VC is proved, or (2) the rules do not contribute anything new to the registry, or (3) the allocated computing resources to the prover are exhausted.

4.5 Counter-Matching Process in the Registry

Counter-matching a precursor tree is a top-down process starting from the root node and searching for matches in the registry. The matching process is straightforward when matching trees to trees. However, if trees in the registry were stored distinctly, searching for matches would be highly inefficient as each entity in the registry would have to be investigated to establish a match. To make the searching process effective, the registry in this work stores trees in congruence classes, which makes the matching problem more complicated than simple tree matching. It takes additional registry organization and more sophisticated strategies to search the registry and perform counter-

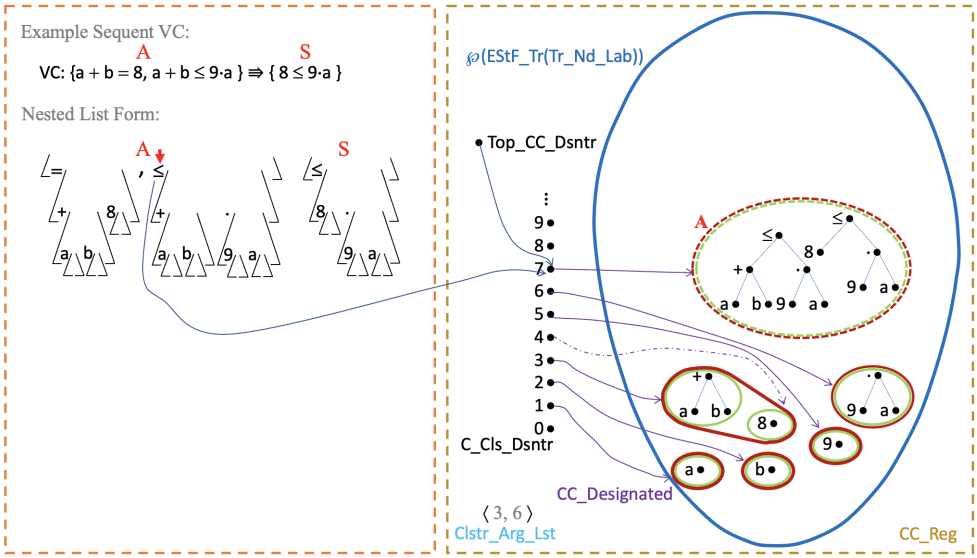


Figure (4.15) Registering ≤ operator with a non-trivial class in the argument list

matching effectively.

4.5.1 Registry Organization for Searching

The registry is organized in a four-tiered representation with `varieties` at the highest level and followed by its refinement, the congruence `classes`. The next level is termed `stands`, which refine the congruence classes, and finally, at the lowest level are congruence `clusters`. Using the precursor tree on the left of Figure (4.17), we illustrate how counter-matching is achieved in the registry and describe how the four-tiered organization makes the searching process effective.

4.5.2 Searching for Potentially Matching Clusters

The counter-matching of a precursor tree in Figure (4.17) starts at the top by looking in the registry for a class tagged with S that contains trees with ≤ as the root node label. In practice, the registry will have many congruence classes, and exhaustively searching all classes will be expensive. We need a better way to narrow our search to only classes that can lead us to a match.

We utilize congruence class varieties to narrow down the search for the root node label to a few potential classes. Varieties are shown in Figure (4.18), and they contain an ordered list of classes with at least one tree having the root node label required. For example, searching for the first precursor tree in Figure (4.16) starts with the root node ≤, and the first variety in Figure (4.18)

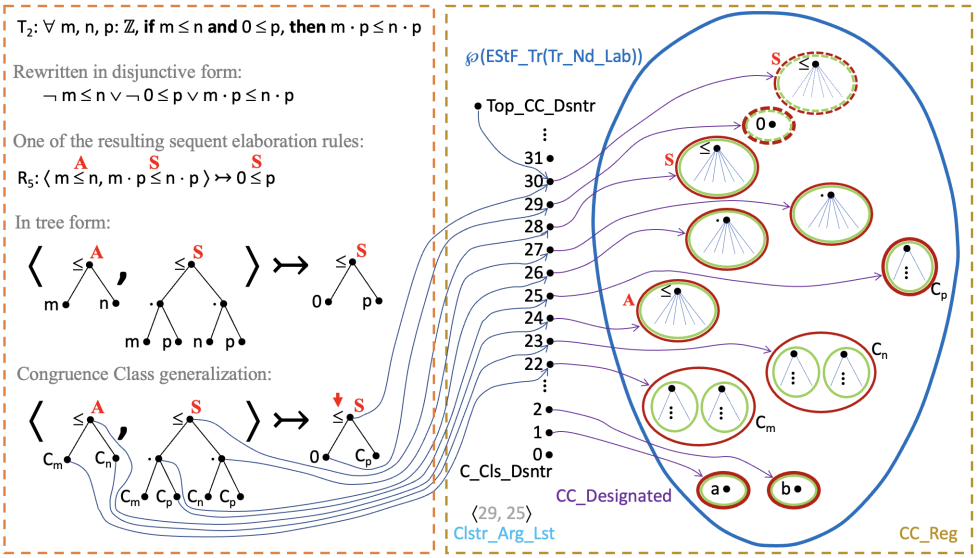


Figure (4.16) Registering a resultant tree with classes on the leaves

helps to eliminate all other classes in remaining varieties. The search in the variety goes through classes one by one in a specific order. The class selected must be tagged with an attribute matching the one in the precursor tree. The attribute match is a requirement to achieve counter-matching.

In practice, congruence classes in the registry contain many trees, and searching within classes presents a similar challenge where trees are searched exhaustively to establish a match. Our solution is to refine the congruence classes into clusters, the subclasses containing trees with the same root label. In a registry's organization, congruence clusters are the finest refinement, making them easy to manipulate.

The number of clusters in a congruence class can be large, which presents a problem in searching. In Figure (4.17), class `30` has three clusters. While this example does not show the scale of the problem, we can still observe that cluster `54` has only trees with $<$ operator as a root and should be eliminated from the search for $\leq$ operator. We have introduced stands in the registry to solve this problem.

Stands are shown in Figure (4.19). They keep all clusters of trees having the same root node label. For each class, there will be a stand for each root node. In our search of $\leq$ operator, the first stand with cluster `55` and `53` is enough. The clusters in the stand are kept in order, and the search in Figure (4.17) starts with the lowest cluster. All other stands are therefore eliminated from the search.

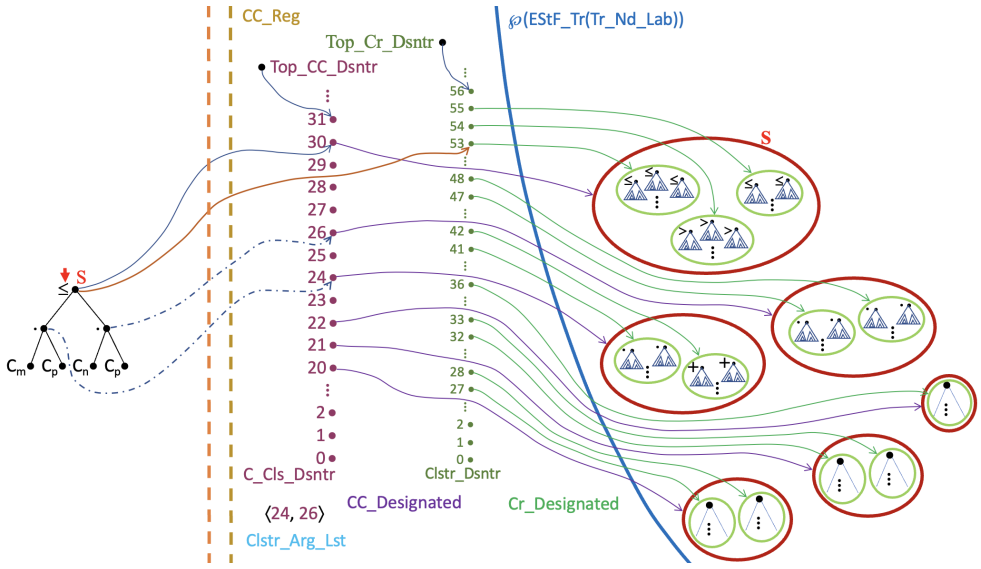


Figure (4.17) Using clusters to find label instances in congruence classes effectively

4.5.3 Counter-Matching Within a Cluster

Now that we have the list of potential clusters, searching can proceed to the next level down from the root node of the precursor tree in Figure 4.17. The requirement here is to get a match in the registry for the two product operators in the root branches. In the registry organization, we have clusters, which keep two crucial pieces of information: the root label and an argument string containing class designators for the root arguments. Therefore, instead of searching the entire registry to find the operators in the root branches, we can follow the cluster's argument string classes and determine if they contain trees with the product as a root label.

The first cluster to check in class 30 is cluster 53, which comes first in its stand. Suppose its argument string contains class 24 and 26, the task would be to find if these two classes contain trees with the product as root label. In this example, both classes contain the product operator. Otherwise, the search should consider the next cluster in the stand. The two classes with product operators match our precursor tree from its root down to the arguments in the first level. The classes that match the two product operators are shown with the dotted lines in Figure (4.17).

When the matching process is complete, the last level in precursor trees contains variables and constants translated to congruence classes. The translation uses clusters found in the classes for the operators discovered on the level preceding the variables and constants. For each cluster, argument classes are selected for the variables and constants to match their occurrence in the tree.

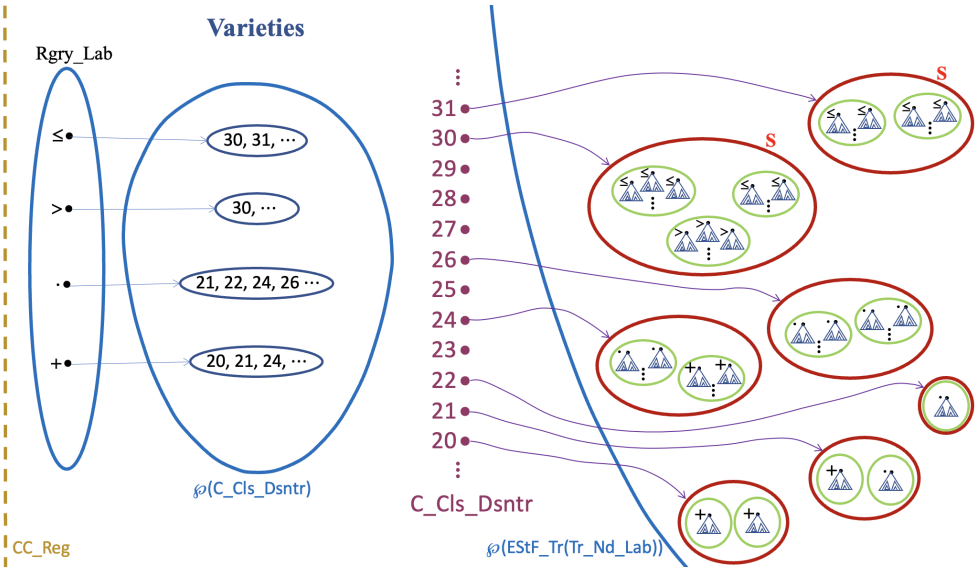


Figure (4.18) Using varieties to find classes in the registry effectively

The precursor tree in Figure (4.17) has all variables already translated to congruence classes. To describe the translation, we should start from classes 24 and 26 found for product operators in the previous level. Each product operator has two variables. The goal is to match them to congruence classes while ensuring that any variable occurring in more than one place in the tree is matched to the same congruence class.

To match the first two variables, we start with the first cluster in the stand for the product operator inside class 24. The cluster will provide us with classes for the arguments. Assuming the classes in the argument string for the selected cluster are C_m and C_p, the translated tree will have the two classes, which replace the variable m and p respectively. Since the variable p is the second argument for both the first and second product operator, the clusters we choose in class 24 and 26 must have the same class in their arguments for p. We have selected a cluster in class 24 with C_p for a variable p, which forces the cluster we select in class 26 to contain C_p for the variable p.

Because the cluster selected in class 26 must have C_p as one of its classes in the arguments, if a cluster is found, it concludes a counter-match for the respective precursor tree. Otherwise, it is a miss-match, and another cluster in class 24 is selected. A counter-match for the entire rule is reached only when congruence classes for the variables match their occurrences in all precursor trees. If the rule is determinate, successful translation of variables in the precursor trees means that the resultant tree can also be translated. The resultant tree is then registered to elaborate the sequent

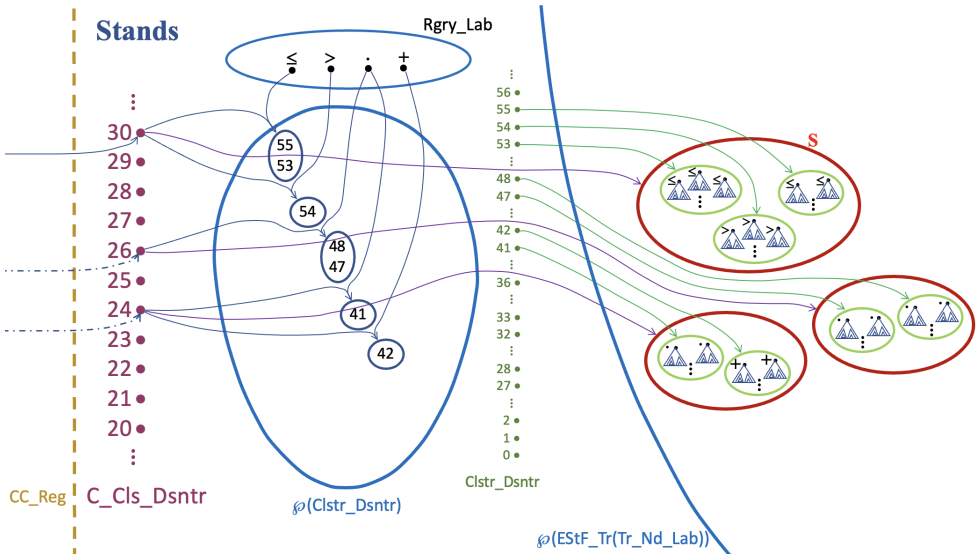


Figure (4.19) Using stands to find clusters in congruence classes effectively

VC in the registry. A detailed explanation of determinate and indeterminate rules is the topic of Section 5.3.1.

4.5.4 Partitioned Search State for Precursor Clauses

The elaboration rules may be applied more than once during instantiation. Both uncounter-matched rules and counter-matched rules must be re-explored as the state of the registry changes with each sequent VC elaboration. If a rule is revisited, starting over by matching a precursor clause that was processed before is ineffective. The matching process can be made more effective by keeping the node's state information usable when a node in the clause is revisited.

At every point in the search, two accessors representing the state of the search are recorded at the node on the precursor tree at the client side: One accessor for a class and one for a cluster. These two accessors make it clear which class or cluster was last searched in the registry, when backtracking on a search or when revisiting a rule, and the new search can start from there.

4.6 Summary

This chapter has presented an overview of the Uni-Prover. We have discussed the central role of the congruence class registry in verifying sequent VC. The discussion has involved descriptive examples and diagrams showing how sequent VCs are registered, how equality in the antecedent is handled, and how elaboration rules are instantiated through searching of precursor clauses for counter-matching and registration of resultant clauses once counter-matched is achieved. The elaboration rules are going to be applied likely many times during the verification process of the sequent VC. The search state is intended to keep the search effective when elaboration rules are revisited.

This chapter serves the additional purpose of setting the stage for the discussion in later chapters that contain a rigorous description of the mathematics involved and a formal specification of the congruence class registry.

Chapter 5

Contiguous Instantiation Strategy

This chapter describes the contiguous instantiation strategy, a technique designed and developed as a part of this work with the goal of providing effective instantiation of universally quantified theorems. The strategy is fully automatic and requires neither user-supplied heuristics nor triggers when instantiating universally quantified theorems in any theory, yet avoids proving of one sequent VC from expanding into proving many.

5.1 Motivation

Various quantifier instantiation techniques have been developed and utilized in different verification systems. While the goal remains the same, how instantiation takes place may have a costly consequence that limits the system's scalability and effectiveness. This section presents two existing instantiation techniques and their challenges to motivate the need for the contiguous instantiation technique developed in this work.

5.1.1 Straightforward Automated Instantiation

Straightforward automated instantiation for universally quantified theorems uses arbitrarily selected ground terms from the sequent VC to instantiate the theorem. Even if the selection is based on relevant heuristics, the process can become expensive. Arbitrary selection of ground terms means the resulting theorem instance can cause the target sequent VC to fork into multiple sequents as explained below.

If a theorem instance with either a conjunction or a disjunction of ground clauses is added to a sequent VC, either the $\wedge$`Left` or $\vee$`Right` sequent deduction rule is applied to remove the logical connectives. The rules are stated in (5.1) below, where added disjunctive clauses $(\psi \vee \phi)$ or conjunctive clauses $(\psi \wedge \phi)$ to the conclusion of the rule result in the branching of the sequent into two premises, each with one of the added clauses—causing a challenge in applying these rules to the target sequent VC, which may fork into many sequents if the added theorem instance is a conjunction or a disjunction of ground clauses.

$$\vee\texttt{Left}: \frac{\Gamma, \psi \Rrightarrow \Delta \qquad \Gamma, \phi \Rrightarrow \Delta}{\Gamma, \psi \vee \phi \Rrightarrow \Delta} \qquad\qquad \wedge\texttt{Right}: \frac{\Gamma \Rrightarrow \Delta, \psi \qquad \Gamma \Rrightarrow \Delta, \phi}{\Gamma \Rrightarrow \Delta, \psi \wedge \phi} \tag{5.1}$$

Generally, if an instance of `n` disjunctively or conjunctively connected ground clauses is added to the sequent VC, the sequent branches into `n` premises to be proved separately. The additional sequents VCs increase the number of steps in the proof process and minimize the possibility of verifying the original sequent in the allotted time.

The following example demonstrates how an arbitrary selection of ground terms from the sequent can create an instance of the theorem that leads to an expensive proof process. Suppose we want to establish the correctness of sequent VC 5.2 below generated from verifying a sum of squares (`SS`) program. Because the two clauses in the antecedent are not sufficient to prove this sequent VC, the automated prover must draw relevant theorems from the mathematical library to proceed with the verification process.

$$\{\texttt{SS} = \sum_{\texttt{i}=1}^{\texttt{CBd}-1} \texttt{i}^2, \texttt{CBd} \leq \texttt{Bd} + 1\} \Rrightarrow \{\texttt{SS} = \sum_{\texttt{i}=1}^{\texttt{Bd}} \texttt{i}^2, \texttt{CBd} \leq \texttt{Bd}\} \tag{5.2}$$

The library houses all mathematical theorems developed in the verification system, and only a smaller set relevant to the target sequent VC is extracted and used to support the proof. Theorems are relevant if and only if they exclusively include the operators available in the target sequent VC. Below are three example theorems relevant to sequent VC 5.2 expressed in Universally Disjunctive

Form (UDF).

$$\mathtt{T}_1 : \forall \mathtt{m}, \mathtt{n} : \mathbb{Z}, \mathtt{m} = \mathtt{n} \vee \neg \mathtt{m} \leq \mathtt{n} \vee \neg \mathtt{n} \leq \mathtt{m}$$

$$\mathtt{T}_2 : \forall \mathtt{m}, \mathtt{n} : \mathbb{Z}, \mathtt{m} \leq \mathtt{n} \vee \mathtt{n} + 1 \leq \mathtt{m}$$

$$\mathtt{T}_3 : \forall \mathtt{m}, \mathtt{n} : \mathbb{Z}, \mathtt{m} = \mathtt{n} \vee \neg \mathtt{m} + \mathtt{p} = \mathtt{n} + \mathtt{p}$$

Suppose $\mathtt{T}_2$ is arbitrarily selected for straightforward instantiation where ground terms SS and Bd are arbitrarily chosen and substituted for the variables in $\mathtt{T}_2$, $[\mathtt{m} \rightsquigarrow \mathtt{SS}, \mathtt{n} \rightsquigarrow \mathtt{Bd}]$. After substitution, the resulting instance of $\mathtt{T}_2$ is $\mathtt{SS} \leq \mathtt{Bd} \vee \mathtt{Bd} + 1 \leq \mathtt{SS}$. The problem with this arbitrarily created instance is manifested when added to the sequent VC as part of the sequent VC elaboration process.

Figure (5.1) illustrates what happens when $\mathtt{SS} \leq \mathtt{Bd} \vee \mathtt{Bd} + 1 \leq \mathtt{SS}$ is added to the sequent VC 5.2. The target sequent VC forks into two premises, which must be proved before we can conclude the original sequent VC is correct. Our hypothesis is that if ground terms are carefully chosen, it is possible to avoid forking of the target sequent VC. The contiguous instantiation strategy presented in Chapter 5 achieves just that.

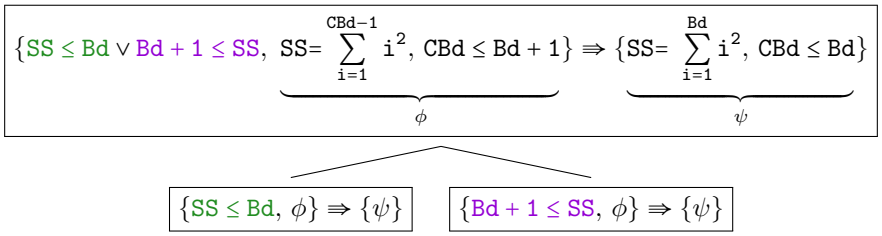


Figure (5.1) Branching of a target sequent VC in straightforward instantiation

Figure (5.2) provides a complete general illustration of a verification system employing a straightforward instantiation technique. The diagram starts with a collection of generated sequent VCs provided to the prover for verification. The target sequent VC is the one currently considered for verification. Each sequent VC has the antecedents represented with $\mathtt{A}_\mathtt{i}'\mathtt{s}$ and succedents represented with $\mathtt{S}_\mathtt{j}'\mathtt{s}$. The two sides are separated by a vertical line representing the set implication arrow ($\Rrightarrow$) underneath.

If we assume the target sequent VC's current state has insufficient antecedents ($\mathtt{A}_1$ and $\mathtt{A}_2$) to prove it, then relevant theorems must be selected from the math library. In this case, T1, T2, and T3 are selected as shown in Figure (5.2). From the three theorems, T2 is arbitrarily selected and

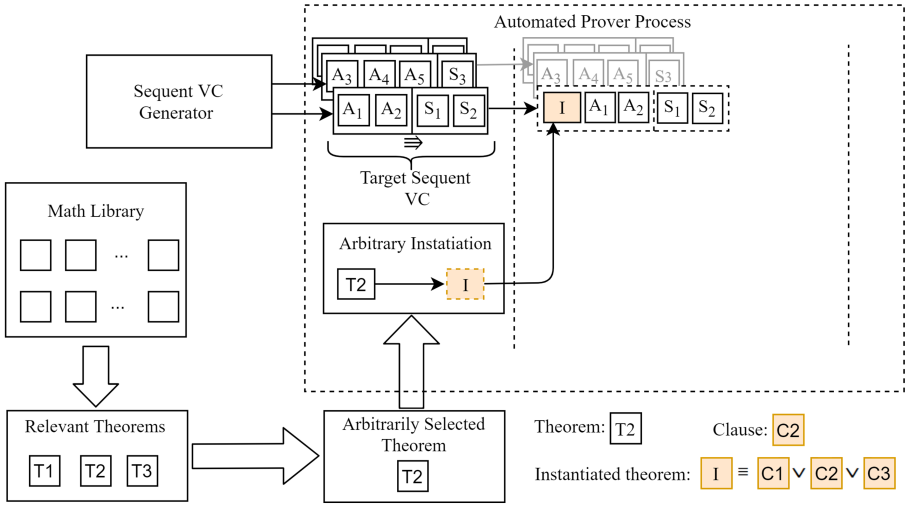


Figure (5.2) Elaborating a target sequent VC by adding an instance I

arbitrarily instantiated. A straightforward automated instantiation of `T2` will involve an arbitrary selection of ground terms from the target sequent VC, followed by ground terms substitution to create an instance `I` of the theorem.

Once `I` is created, it is added to the sequent VC as part of an elaboration process shown in Figure (5.2). The resulting sequent VC now includes $\mathtt{I}, \mathtt{A}_1, \mathtt{A}_2$ as antecedents. An instance `I` is considered to contain three disjunctively connected ground clauses $\mathtt{C}_1$, $\mathtt{C}_2$, and $\mathtt{C}_3$ (see the right lower corner of Figure (5.2)), and when added to the sequent VC, the VC forks into three premises each containing one of the clauses in `I`. We demonstrate this step in Figure (5.3), where the first sequent VC contains $\mathtt{C}_1$, the second contains $\mathtt{C}_2$, and the last one contains $\mathtt{C}_3$. Each resulting sequent VC is now considered one at a time as the new target sequent for the next verification process. The proliferation of sequent VCs as the theorem instance is added to the target sequent makes this strategy inadequate for this work as it bogs down the verification process.

5.1.2 E-Matching

E-matching [14] is a common strategy used for quantifier instantiation in verification systems, such as Simplify [17] and Z3 [15]. It is a pattern-driven instantiation technique, and it works by embedding a pattern to a universally quantified formula, which acts as a trigger for the respective formula to be instantiated. The pattern contains a set of terms used in pattern matching. A

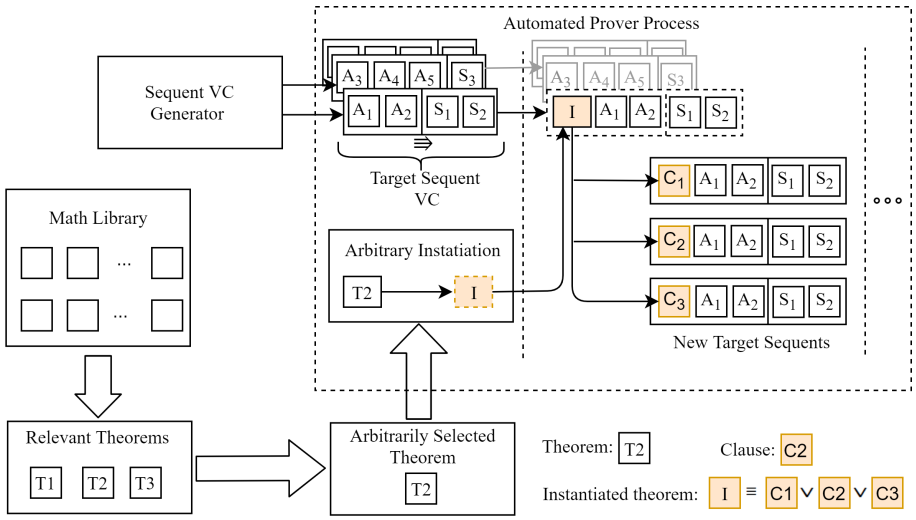


Figure (5.3) Proliferation of target sequent VCs following inclusion of I

pattern match determines the formula to be instantiated and the respective ground terms from the E-graph (Equality-graph) to be used for instantiation. E-graph is a congruence-closed representation of ground facts, and it is used to compute equalities between terms efficiently. The E-matching technique works with expressions and faces multiple challenges that make it prone to matching loops. Importantly, it needs human assistance.

To show how E-matching uses triggers for instantiation, let us consider an example quantifier formula $\forall \mathtt{x} : \mathtt{Int}, \mathtt{f(x)} = 45 * \mathtt{g(x)} + 99$. A possible trigger pattern for this formula is $\{\mathtt{f(x)}\}$. When a ground term such as $\mathtt{f(Y)}$ is encountered in the E-graph, it will trigger the entire quantifier body to be instantiated through as substitution $\mathtt{x} = \mathtt{Y}$. The result is an instance of the formula $\mathtt{f(Y)} = 45 * \mathtt{g(Y)} + 99$. Similar results can be found with $\{\mathtt{g(x)}\}$ as a trigger pattern.

Consider another quantified formula $\forall \mathtt{i} : \mathtt{Int} :: 0 < \mathtt{i} \implies \mathtt{f(i)} = \mathtt{i} * \mathtt{f(i-1)}$. One possible trigger pattern for this formula is $\{\mathtt{f(i)}\}$. When a ground term $\mathtt{f(x)}$ is encountered in the E-graph, it triggers a full instantiation of the formula, which includes a term $\mathtt{f(x-1)}$. The problem with this pattern match is the term $\mathtt{f(x-1)}$, which yields another possible instantiation on the same formula causing an indefinite process in a *matching loop*. If $\{\mathtt{f(i-1)}\}$ is selected as trigger pattern instead, the instantiation still ends in a matching loop.

Ongoing research efforts have been directed toward solving the matching loop problem in E-matching. Among them is the work by Ge et al. in [22] and the work by Barbosa in [5], both efforts illustrate how to deal with matching loops automatically.

A related problem is caused by trigger selection when a too-conservative trigger is picked, causing essential formulas needed by the proof to be skipped. On the other hand, if a pattern chosen is not sufficiently conservative, it may lead to an overwhelming number of formula instantiations leading to performance issues. Trigger patterns must be constructed carefully, which requires experience and tuning. Currently, automated trigger creation techniques such as one presented in [41] do exist. Though the technique achieves performance gain and improves predictability, in the general case, human assistance is necessary to create triggers successfully.

5.1.3 Discussion

The instantiation technique needed for this work must be effective to reduce the number of steps taken to verify the correctness of the target sequent VC. While the straightforward instantiation technique is simple, it leads to a more expensive proof process by introducing more sequent VCs that have to be proved. Because the computing resources are limited, these additional steps minimize the chances of verifying the target sequent VC. Hence the need for a technique that keeps the target sequent VC from forking.

E-matching uses triggers as programming tactics applied to reduce the search space. However, mathematicians develop theorems without considering triggers that may be created later. The creation and use of triggers lack the sophistication needed for a general case. There is a need for an instantiation technique that, unlike E-matching, eliminates the need to create triggers while achieving the necessary sophistication.

The contiguous instantiation strategy presented in Section 5.2 is triggerless and uses theorems that are broken down into elaboration rules. Additionally, unlike triggers that are specific to a rule resulting in a single instantiation, the strategy proposed here finds all instances for the elaboration rule at hand until all instantiation options are exhausted.

5.2 The Contiguous Instantiation Strategy in Detail

Contiguous instantiation is a novel non-heuristic strategy for instantiation of universally quantified theorems. It is engineered to address the fundamental problems in E-matching and straightforward instantiation strategies explained in Section 5.1. Together with other components and strategies designed in this work, contiguous instantiation is intended to optimize the verification

process by reducing the number of steps taken to prove a sequent VC.

Unlike E-matching, contiguous instantiation is triggerless, and it avoids sequent VC forking caused by a Straightforward instantiation of theorems. Being "contiguous" means the instantiation steps are taken systematically through a purpose-driven selection of ground terms from the sequent VC. In this strategy, every ground term selection is motivated to achieve a "counter-matching" of clauses in elaboration rules and sequent VC. Counter-matching was introduced in Section 4.5, this section presents more details.

5.2.1 Theorem Presentation for the Continguous Instantation Strategy

The contiguous instantiation strategy works with theorems presented in a Disjunctive Normal Form (DNF). DNF theorems naturally correspond to sequents and are suitable for automated verification. DNF theorems can be combined with universal quantifiers to get Universally quantified Disjunctive Formulas (UDF). In UDF theorems, quantifiers are at the front and succeeded by a disjunct of positive and or negative predicate clauses. The general form of theorems in UDF is shown in Fig. 5.4(a). Each theorem contains predicate clauses represented by $\mathtt{P_m}$. The clauses have terms $(\mathtt{T_i}, \mathtt{T_j}, \mathtt{T_k})$ with quantified variables $\mathtt{x_n}$. We have also provided integer theorems presented in UDF as examples in Fig. 5.4(b).

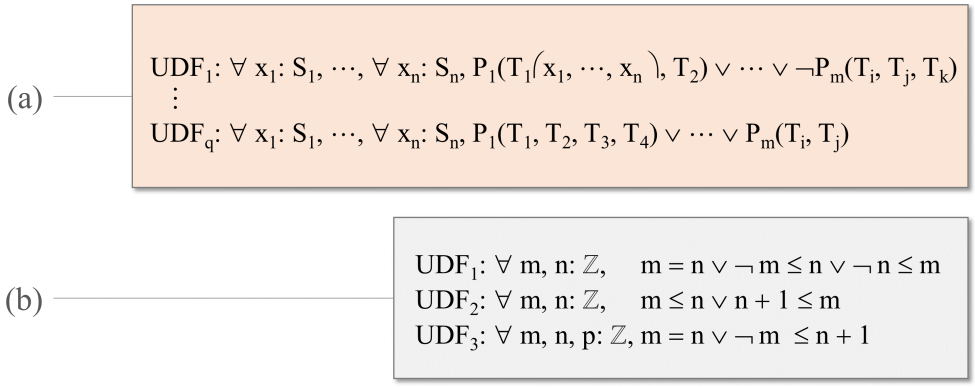


Figure (5.4) (a). General form of theorems presented in Universally Disjunctive Form (UDF) (b). Example integer theorems in UDF

5.2.2 Using UDF Theorems in Verification

Once theorems are expressed in universally disjunctive form, using them to verify sequents involves a left universal quantifier rule ($\forall\texttt{Left}$). To demonstrate its use, suppose a sequent $\Gamma \Rrightarrow \Delta$ requires theorems from the library to be verified. Theorems relevant to the sequent will be extracted and instantiated using the $\forall\texttt{Left}$ rule. The $\forall\texttt{Left}$ rule shown in (5.3) illustrates how relevant theorems $\Theta, (\forall \texttt{x} : \texttt{T}, \psi)$ are incorporated in the antecedent of the sequent in the rule's conclusion. Θ represents a collection of all relevant theorems to this sequent from the library, and $(\forall \texttt{x} : \texttt{T}, \psi)$ is the target theorem currently considered. The rule's premise shows an instantiation of the target theorem through ground term substitution $\psi[\texttt{x} \rightsquigarrow \texttt{t}]$ and elaboration of sequent by including the instantiated theorem.

$$\forall\texttt{Left} : \frac{\Theta, \psi[\texttt{x} \rightsquigarrow \texttt{t}], \Gamma \Rrightarrow \Delta}{\Theta, (\forall \texttt{x} : \texttt{T}, \psi), \Gamma \Rrightarrow \Delta} \tag{5.3}$$

5.2.3 Counter-Matching in Contiguous Instantiation Strategy

Consider an example theorem $\texttt{UDF}_2$ shown in Fig. 5.5(a). $\texttt{UDF}_2$ contains two predicates $\texttt{P}_1$ and $\texttt{P}_2$ with terms $\texttt{T}_1$, $\texttt{T}_2$, and $\texttt{T}_3$. If $\{\texttt{A}_1(\texttt{G}_1, \texttt{G}_2, \texttt{G}_3), \texttt{A}_2(\texttt{G}_1, \texttt{G}_2)\} \Rrightarrow \{\texttt{S}_1(\texttt{G}_1, \texttt{G}_2, \texttt{G3}, \texttt{G4}), \texttt{S}_2(\texttt{G}_1, \texttt{G}_2)\}$ is the sequent VC to be proved, and $\texttt{UDF}_2$ is among the relevant theorems extracted to support with the verification process, then $\texttt{UDF}_2$ is included in the antecedent of the sequent VC as explained in Section 5.2.2. The sequent VC has ground clauses $\texttt{A}_1, \texttt{A}_2$, $\texttt{S}_1, \texttt{S}_2$ and ground terms $\texttt{G}_1$ to $\texttt{G}_4$, all expressed in general terms to demonstrate how counter-matching works generally.

This discussion also uses an actual example theorem and sequent VC shown in Figure 5.5(b). This sequent VC is generated from verifying a sum of squares (SS) program and was presented in earlier chapters.

$\texttt{UDF}_2$ must be instantiated to elaborate our sequent VC using the $\forall\texttt{Left}$ rule. Ground terms from the sequent VC are selected and substituted accordingly. Suppose a predicate $\texttt{P}_1(\texttt{T}_1, \texttt{T}_2)$ in $\texttt{UDF}_2$ instantiate to a ground clause $\texttt{S}_1$ in the succedent of the sequent VC via substitution of ground terms $\texttt{G}_1, \texttt{G}_2, \texttt{G}_3$, and $\texttt{G}_4$ as illustrated in Figure 5.6(a). Similarly, a predicate clause $\texttt{m} \leq \texttt{n}$ Figure 5.5(b) is instantiated to $\texttt{CBd} \leq \texttt{Bd}$ as shown in Figure 5.6(b).

Because the variables involved in the second predicate clause $\texttt{P}_2(\texttt{T}_3)$ are a subset of variables in $\texttt{P}_1(\texttt{T}_1, \texttt{T}_2)$, after $\texttt{P}_1$ is instantiated, the instantiation of $\texttt{P}_2(\texttt{T}_3)$ is deterministic and straightforward.

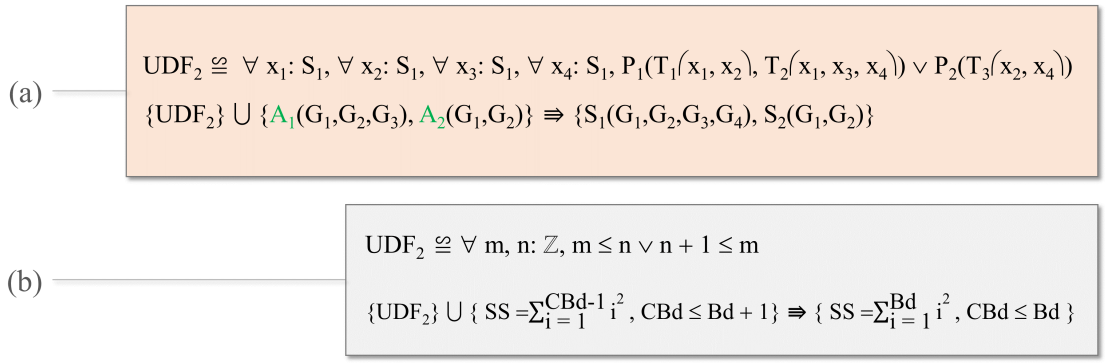


Figure (5.5) (a). Adding $\texttt{UDF}_2$ in a sequent represented in general terms. (b). Example integer theorem added to a sequent VC

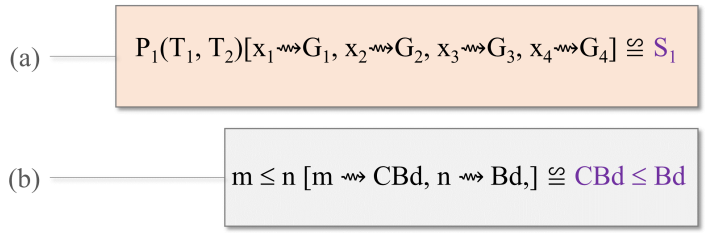


Figure (5.6) (a). Ground terms substitution for the predicate $\texttt{P}_1$. (b). Example ground terms substitution for the predicate $\texttt{m} \leq \texttt{n}$

Let us assume $\texttt{P}_2$ has instantiated to a ground clause $\texttt{D}$ as shown in Figure 5.9. An instance of $\texttt{P}_2$ does not counter-match any clause in the sequent VC, and in this work, it is called uncounter-matched clause.

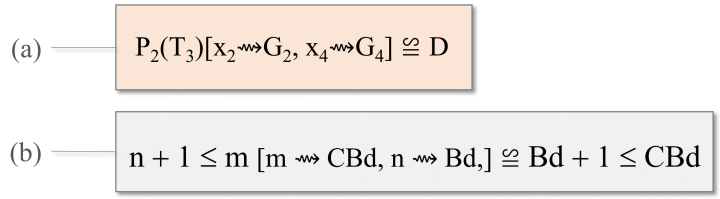


Figure (5.7) (a). Ground terms substitution for the predicate $\texttt{P}_2$. (b). Example ground terms substitution for the predicate $\texttt{n} + \texttt{1} \leq \texttt{m}$

At this point, we have created an instance $\boxed{\texttt{S}_1 \vee \texttt{D}}$ of the theorem $\texttt{UDF}_2$ with two disjunctive ground clauses, $\texttt{S}_1$ (after instantiating $\texttt{P}_1$), and $\texttt{D}$ (after instantiating $\texttt{P}_2$). Next, $\boxed{\texttt{S}_1 \vee \texttt{D}}$ is added to the sequent VC's antecedent to form an elaborated sequent $\texttt{A}_1 \wedge \texttt{A}_2, \boxed{\texttt{S}_1 \vee \texttt{D}} \Rightarrow \texttt{S}_1 \vee \texttt{S}_2$. It is critical

to note that our choice to instantiate $\mathtt{P}_1$ to a clause $\mathtt{S}_1$ was intentional. As we can observe, the elaborated sequent VC now has a ground clause $\mathtt{S}_1$ on both sides demonstrating the fundamental idea of counter-matching. The choice of the word "counter" is to insist on the fact that the goal is for the instantiated clause to match a ground clause on the opposite side of the sequent VC. Therefore, the ground terms are chosen to achieve that goal. In the case of our example, the ground terms were purposely selected to create an instance $\mathtt{S}_1$, which counter-matches $\mathtt{S}_1$ in the succedent of the sequent VC.

5.2.4 Effectiveness of the Contiguous Instantiation Strategy

The effectiveness of contiguous instantiation as compared to straightforward instantiation and E-Matching is demonstrated when either $\vee$`Left` or $\wedge$`Right` rule is applied to the elaborated sequent VC with a counter-matched clause like $\mathtt{A}_1 \wedge \mathtt{A}_2, \boxed{\mathtt{S}_1 \vee \mathtt{D}} \Rrightarrow \mathtt{S}_1 \vee \mathtt{S}_2$ found above. The $\vee$`Left` rule is applied to the sequent VC $\mathtt{A}_1 \wedge \mathtt{A}_2, \boxed{\mathtt{S}_1 \vee \mathtt{D}} \Rrightarrow \mathtt{S}_1 \vee \mathtt{S}_2$, which branches into two premises shown in Figure (5.8).

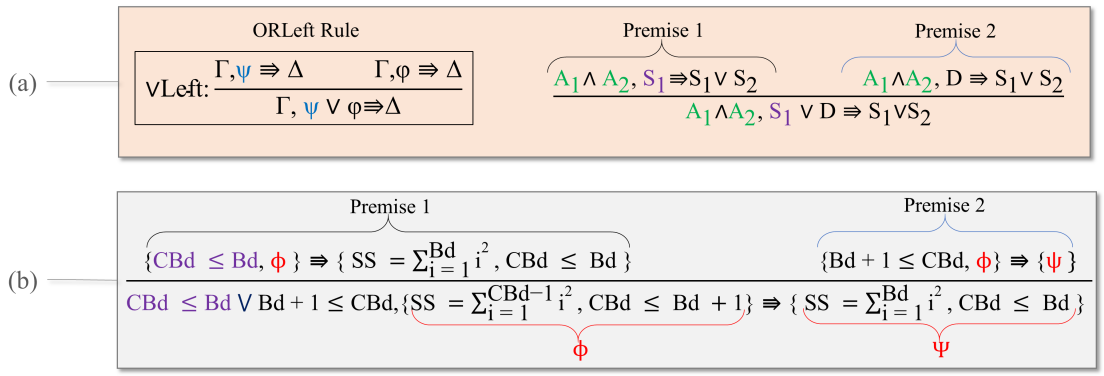


Figure (5.8) (a) Applying the ORLeft rule in a sequent with counter-matched clauses. (b). An example illustrating an application of the ORLeft rule on a sequent with counter-matched clauses

Among the two premises, $\mathtt{A}_1 \wedge \mathtt{A}_2, \mathtt{S}_1 \Rrightarrow \mathtt{S}_1 \vee \mathtt{S}_2$ is an instance of an axiom rule with a ground clause $\mathtt{S}_1$ appearing on both sides of the sequent. This premise reduces to **`true`** by the rules of the sequent calculus. Axiom instances do not split any further, and their correctness can be established immediately. Therefore, only the correctness of the second premise needs to be established.

Contiguous instantiation strategy is designed around the counter-matching strategy, which starts when ground terms are being selected. The selection is systematic, and motivated to ensure

that theorem clauses are counter-matched with ground clauses in the sequent VC. As a result, the proof process is going to focus on a single target sequent by having the others reducing to **true**. Therefore, we can deduce that:

— Any counter-matched clause results in a premise that is an instance of an axiom rule.

— Only the premise with an uncounter-matched ground clause remains to be verified.

The two findings above allow us to create elaboration rules from the UDF theorems and use them instead in the instantiation process. To demonstrate how elaboration rules are created, consider $\texttt{UDF}_\texttt{i} = \forall \texttt{x}_1 : \texttt{S}, \cdots, \forall \texttt{x}_\texttt{j} : \texttt{S}, \texttt{C}_{1,1} \vee \cdots \vee \texttt{C}_{\texttt{i,k}}$ with a set of clauses $\{\texttt{C}_{\texttt{i},1} \cdots, \texttt{C}_{\texttt{i,m}}, \cdots, \texttt{C}_{\texttt{i,k}}\}$. For each determinate clause $\texttt{C}_{\texttt{i,m}}$ in the set $\{\texttt{C}_{\texttt{i},1} \cdots, \texttt{C}_{\texttt{i,m}}, \cdots, \texttt{C}_{\texttt{i,k}}\}$, an elaboration rule is created where $\{\texttt{C}_{\texttt{i},1}, \cdots, \texttt{C}_{\texttt{i,k}}\} \sim \{\texttt{C}_{\texttt{i,m}}\}$ are precursor clauses, and $\{\texttt{C}_{\texttt{i,m}}\}$ is a resultant clause. The theorem $\texttt{UDF}_\texttt{i}$ could produce up to $\texttt{k}$ distinct elaboration rules determined by the number of determinate clauses in the theorem. Once elaboration rules are created, the instantiation of theorems is reduced to an instantiation of elaboration rules. Each rule is instantiated by counter-matching all the precursor clauses and elaborating the sequent VC using only the resultant clause. As concluded above, all counter-matched clauses create instances of the axiom rule and reduce to true. Therefore, the proof continues by dealing with a single sequent VC containing the added resultant clause. The use of elaboration rules in instantiation is explained in Section 5.2.6.

5.2.5 Counter-Matching of Negative Predicate Clause

The theorems used in the discussion above involved positive predicate clauses only. However, UDF theorems can contain negative clauses. This section explains how negative clauses are counter-matched.

Consider $\texttt{UDF}_3$ in Figure (5.9) with a positive predicate $\texttt{P}_1$ and a negative predicate $\texttt{Q}_1$. If $\{\texttt{A}_1(\texttt{G}_1, \texttt{G}_2, \texttt{G}_3), \texttt{A}_2(\texttt{G}_1, \texttt{G}_2)\} \Rrightarrow \{\texttt{S}_1(\texttt{G}_2, \texttt{G3}), \texttt{S}_2(\texttt{G}_1, \texttt{G}_2)\}$ is the sequent VC to be proved and $\texttt{UDF}_3$ is among the relevant theorems extracted to support the verification process, then $\texttt{UDF}_3$ is included in the antecedent of the sequent VC as explained in Section 5.2.2. The sequent VC has ground clauses $\texttt{A}_1, \texttt{A}_2$, $\texttt{S}_1, \texttt{S}_2$ and ground terms $\texttt{G}_1, \texttt{G}_2$, and $\texttt{G}_3$.

Suppose we instantiate $\texttt{Q}_1(\texttt{T}_2, \texttt{T}_3)$ in the theorem to a ground clause $\texttt{A}_1$ in the sequent VC. The instantiation of $\texttt{Q}_1$ is shown in Figure (5.10), with $\texttt{G}_1, \texttt{G}_2$ and $\texttt{G}_3$ used as ground terms. $\texttt{G}_1$ and $\texttt{G}_3$ are also used to instantiate $\texttt{P}_1$ to create an instance of $\texttt{UDF}_3$. Figure (5.11) shows the sequent VC

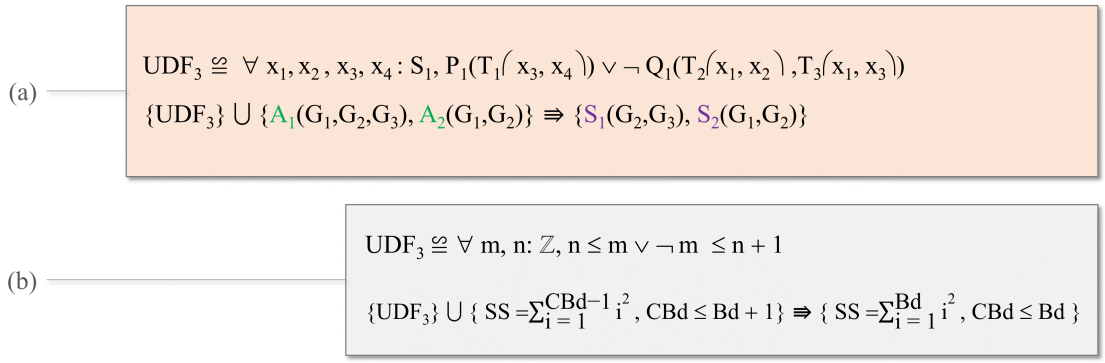


Figure (5.9) (a). Adding `UDF3` with a negative predicate to a sequent. (b). An example integer theorem with a negative predicate added to a sequent

elaboration and the application of the ∨`Left` rule. It should be noted that, because `Q1` is negative, after being added to the antecedent of the sequent VC, the ¬`Left` rule is applied to the premise that contains `Q1`, as shown in Figure (5.12). As an effect of this rule, `Q1` ends up in the succedent as a positive clause. Any negative clause in the theorem must end up in the succedent as a positive clause, and its counter-match must be in the antecedent of the sequent VC. Our choice of the ground terms `G1`, `G2`, and `G3` were intentional to create a counter-match between an instance of `Q1`, which will end in the succedent with a ground clause in the antcedent of the sequent VC.

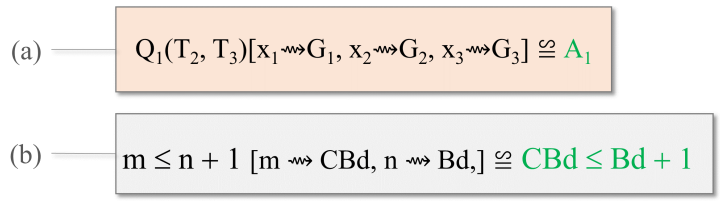


Figure (5.10) (a). Ground terms substitution for a negative predicate `Q1`. (b). An example ground terms substitution for a negative predicate `m ≤ n + 1`

5.2.6 Counter-Matching With Elaboration Rules

The discussion in Sections 5.2.4 and 5.2.5 suggested the use of elaboration rules in contiguous instantiation. Instead of dealing with an entire UDF theorem for the counter-matching of clauses, elaboration rules created from the theorems are utilized. However, to directly capture all cases encountered when dealing with theorems, elaboration rules are annotated with extra information to

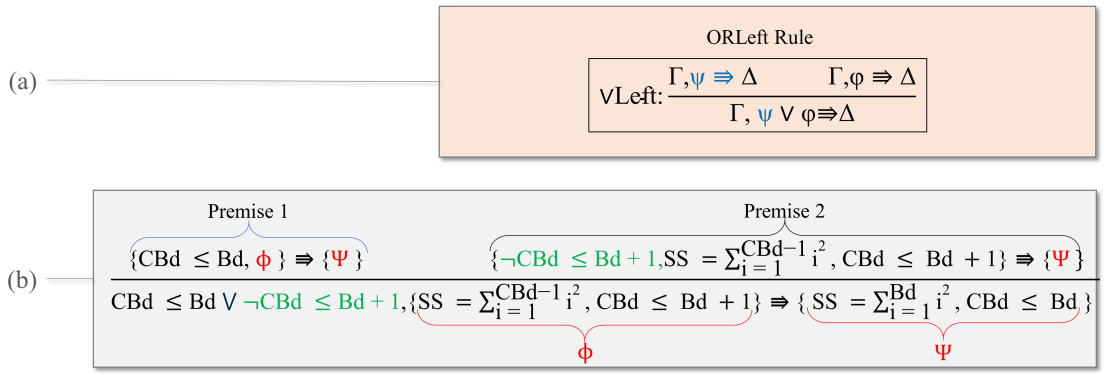


Figure (5.11) (a) The ORLeft rule. (b). Applying ORLeft rule to a sequent with a negative clause

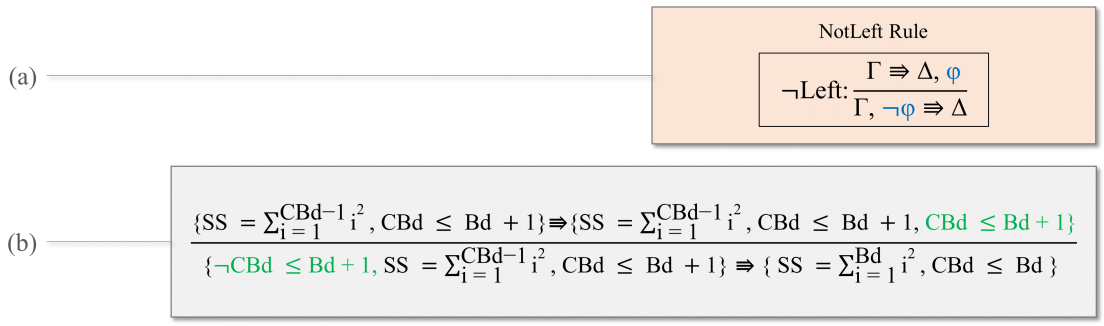


Figure (5.12) (a). The NotLeft rule of sequent calculus. (b). Applying NotLeftRule to a sequent with a negative clause

support the counter-matching and sequent VC elaboration process.

The following are two key pieces of information that should be included in elaboration rules. First is which side of the sequent VC should the counter-matching ground clause be. This information is annotated to the precursor clauses depending on whether they were positive or negative. For example, consider a UDF theorem $\mathtt{T}_1$ with three predicate clauses, two of which are negative, and one is positive. Fig. 5.13 shows three rules that are generated from $\mathtt{T}_1$. For rule $\mathtt{R}_1$ first precursor clause is positive, and it is annotated with S to mean its counter-match ground clause should come from the succedent. The second precursor clause is negative and annotated with A. Its counter-match should come from the antecedent.

The second piece of information is attached to the resultant clauses. This information helps determine where the resultant clause should be added to the sequent as part of the sequent VC

elaboration step. The annotation also depends if the clause is negative or positive. A negative clause is annotated with S and goes to the succedent. Positive clauses are annotated with A and go into the antecedent.

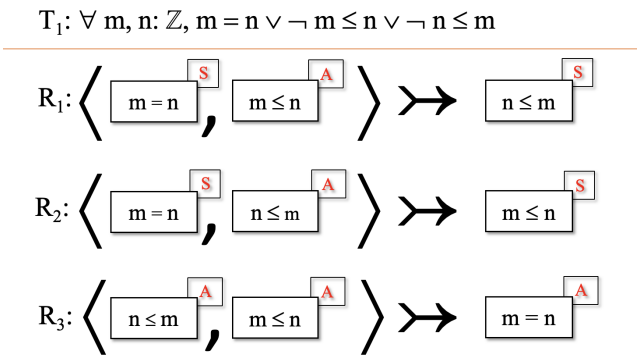


Figure (5.13) Creating elaboration rules from a theorem $\texttt{T}_1$

The goal of using elaboration rules is to ensure all precursor clauses are counter-matched, then only the resultant clause is added to the sequent VC. Adding one resultant clause to the target sequent VC keeps the verification process single-threaded.

5.3 Visualizing Contiguous Instantiation Strategy Process

Figure (5.14) demonstrates the entire process involved in contiguous instantiation of an elaboration rule. From the left, relevant theorems and a collection of respective elaboration rules are selected from the math library. One rule after the other is selected from the collection and contiguously instantiated. In the diagram, the rule $\texttt{P}_\texttt{a}\texttt{R}_\texttt{a}$ is selected for instantiation. $\texttt{P}_\texttt{a}$ is considered to contain two precursor clauses `C1` and `C2` (see the bottom right of the diagram). Before instantiating the rule, a counter-match must be established between precursor clauses (`C1` and `C2`) in the rule and ground clauses in the target sequent VC. No triggers are necessary for the matching process, and no matching loops are possible.

Once a counter-match is established, an instance of the resultant clause ($\texttt{I}_{\texttt{Ra}}$) is created and used to elaborate the target sequent VC, which becomes the **only** new target in the next process. No forking of the sequent VC happens in contigous instantiation, which keeps the verification process focusing on a single sequent.

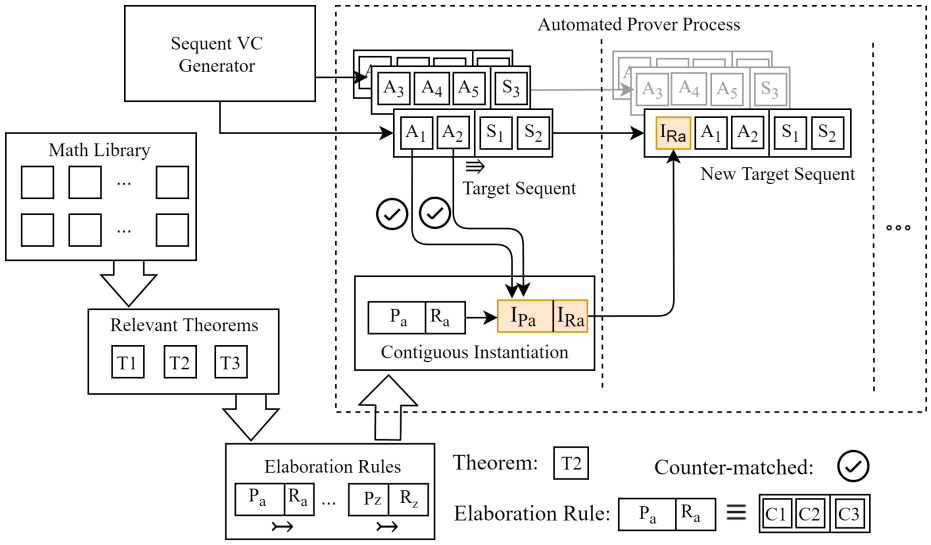


Figure (5.14) General diagram illustrating contiguous instantiation process and its effectiveness

5.3.1 Contiguous Instantiation Cases

Contiguous instantiation involves two prominent cases, **determinate** and **indeterminate** cases. In the determinate case, the free variables in the uncounter-matched predicate clause are a subset of the free variables found in the counter-matched predicate clauses of the disjunctive formula $\texttt{UDF}_\texttt{i}$. It makes the instantiation of the uncounter-matched predicate clause completely determined by the ground terms used in the counter-matched predicate clauses. The instantiation process discussed above is determinate, and section 5.3.1.1 describes the same process using general terms and later defines a general sequent elaboration rule for any determinate case instantiation.

When variables in the uncounter-matched clause are not in the set of variables in counter-matched clauses, the instantiation becomes indeterminate. Further investigation is required to determine other instantiation paths. Otherwise, the uncounter-matched clause is abandoned. The indeterminate case in contiguous instantiation is explained in section 5.3.1.2.

5.3.1.1 Determinate Case

Consider $\texttt{UDF}_\texttt{i}$ below expressed in a traditional implication statement. Its contiguous instantiation involves finding a substitution $\texttt{Sb} = \texttt{x}_1 \rightsquigarrow \texttt{G}_1, \texttt{x}_2 \rightsquigarrow \texttt{G}_2, \cdots \texttt{x}_\texttt{k} \rightsquigarrow \texttt{G}_\texttt{k}$ such that all but one of the predicates $\texttt{P}_1, \texttt{P}_2, \cdots, \texttt{Q}_1, \cdots \texttt{Q}_\texttt{n}$ counter-match appropriate ground clauses in Γ and Δ. The one unconter-matched predicate clause can either be positive or negative.

$$\mathtt{UDF_i} \equiv \forall \mathtt{x_1} : \mathtt{S_1}, \cdots \forall \mathtt{x_k} : \mathtt{S_k}, \mathtt{Q_1} \wedge \mathtt{Q_2} \wedge \cdots \wedge \mathtt{Q_n} \Rightarrow \mathtt{P_1} \vee \mathtt{P_2} \vee \cdots \vee \mathtt{P_m}$$

For the positive case, suppose $\mathtt{C_u} \equiv \mathtt{P_u}[\mathtt{Sb}]$ is the uncounter-matched ground clause, and all the counter-matched ground clauses are denoted by $\mathtt{C_j} \equiv \mathtt{P_j}[\mathtt{Sb}]$ for $\mathtt{j} : (_{\mathbb{N}}[1 \cdots \mathtt{m}] \sim \{\mathtt{u}\})$ and $\mathtt{D_j} \equiv \mathtt{Q_j}[\mathtt{S_b}]$ for $\mathtt{j} :_{\mathbb{N}} [1 \cdots \mathtt{n}]$, and $\Gamma' \equiv \Gamma \sim \{\mathtt{D_1}, \cdots, \mathtt{D_n}\}$ and $\Delta' \equiv \Delta \sim \{\mathtt{C_1}, \cdots, \mathtt{C_{u-1}}, \mathtt{C_{u+1}}, \cdots, \mathtt{C_m}\}$. Then the general sequent VC elaboration rule will be as follows:

$$\frac{\Gamma' \cup \{\mathtt{D_1}, \cdots, \mathtt{D_n}, \mathtt{C_u}\} \Rrightarrow \Delta' \cup \{\mathtt{C_1}, \cdots, \mathtt{C_{u-1}}, \mathtt{C_{u+1}}, \cdots, \mathtt{C_m}\}}{\Gamma' \cup \{\mathtt{D_1}, \cdots, \mathtt{D_n}\} \Rrightarrow \Delta' \cup \{\mathtt{C_1}, \cdots, \mathtt{C_{u-1}}, \mathtt{C_{u+1}}, \cdots, \mathtt{C_m}\}}$$

For the negative case, suppose $\mathtt{D_u} \equiv \mathtt{Q_u}[\mathtt{S_b}]$ is the uncounter-matched negated ground clause, and all the counter-matched ground clauses are denoted $\mathtt{C_j} \equiv \mathtt{P_j}[\mathtt{S_b}]$ for $\mathtt{j} :_{\mathbb{N}} [1 \cdots \mathtt{m}]$ and $\mathtt{D_j} \equiv \mathtt{Q_j}[\mathtt{S_b}]$ for $\mathtt{j} : (_{\mathbb{N}}[1 \cdots \mathtt{n}] - \{\mathtt{u}\}$, and $\Gamma' \equiv \Gamma \sim \{\mathtt{D_1}, \cdots, \mathtt{D_{u-1}}, \mathtt{D_{u+1}}, \cdots, \mathtt{D_n}\}$ and $\Delta' \equiv \Delta \sim \{\mathtt{C_1}, \cdots, \mathtt{C_m}\}$. Then the general sequent VC elaboration rule is:

$$\frac{\Gamma' \cup \{\mathtt{D_1}, \cdots, \mathtt{D_{u-1}}, \mathtt{D_{u+1}} \cdots, \mathtt{D_n}\} \Rrightarrow \Delta' \cup \{\mathtt{C_1}, \cdots, \mathtt{C_m}, \mathtt{D_u}\}}{\Gamma' \cup \{\mathtt{D_1}, \cdots, \mathtt{D_{u-1}}, \mathtt{D_{u+1}} \cdots, \mathtt{D_n}\} \Rrightarrow \Delta' \cup \{\mathtt{C_1}, \cdots, \mathtt{C_m}\}}$$

5.3.1.2 Indeterminate Case

For the indeterminate case, a basic example would involve just a single predicate clause. Consider $\mathtt{UDF_3} \equiv \forall \mathtt{x_1}, \mathtt{x_2} : \mathbb{Z}, (\mathtt{x_1} \cdot \mathtt{x_2}) = (\mathtt{x_2} \cdot \mathtt{x_1})$ and $\mathtt{UDF_4} \equiv \forall \mathtt{x_1}, \mathtt{x_2} : \mathbb{N}, \mathtt{x_1} \leq (\mathtt{x_1} + \mathtt{x_2})$. A contiguous instantiation of $\mathtt{UDF_3}$ would involve finding a subterm (e.g., $\mathtt{G1} \cdot \mathtt{G2}$) in the sequent $\Gamma \Rrightarrow \Delta$, which would lead to a ground substitution $\mathtt{Sb_3} \equiv \mathtt{x_1} \rightsquigarrow \mathtt{G_1}, \mathtt{x_2} \rightsquigarrow \mathtt{G_2}$, creating an instance $\mathtt{G1} \cdot \mathtt{G2} \equiv (x_1 \cdot x_2)[Sb_3]$, a subterm in the sequent $\Gamma \Rrightarrow \Delta$. This case has a term level determinacy whichever ground terms are used for the first term, the other can be determined. The following is a sequent VC elaboration rule is:

$$\frac{\Gamma' \cup \{(\mathtt{G_1}, \cdot \mathtt{G_2}) = (\mathtt{G_2} \cdot \mathtt{G_1})\} \Rrightarrow \Delta}{\Gamma \Rrightarrow \Delta}$$

The theorem $\mathtt{UDF_4}$ raises a term level variant of the indeterminacy issue. First, let's consider $\mathtt{x_1}$ as a targeted term to be the uncounter-matched one. If a subterm $(\mathtt{G_1} + \mathtt{G_2})$ of the sequent $\Gamma \Rrightarrow \Delta$ is identified to match the theorem term $(\mathtt{x_1} + \mathtt{x_2})$, then $\mathtt{x_1}$ can be instantiated, and the sequent VC elaboration rule is:

$$\frac{\Gamma' \cup \{G_1 \leq (G_1 + G_2)\} \Rrightarrow \Delta}{\Gamma \Rrightarrow \Delta}$$

However, if the term $(x_1 + x_2)$ is considered as the target uncounter-matched subterm, finding ground term for x_1 makes $(x_1 + x_2)$ indeterminate. Suppose a ground term G_1 of the sequent $\Gamma \Rrightarrow \Delta$ is selected to match the theorem term x_1, then for a prospective additional antecedent clause $G_1 \leq (G_1 + G_2)$ there would be a second indeterminate choice of a subterm G_2 of the sequent $\Gamma \Rrightarrow \Delta$, and the prospective additional clause would have multiple subterm attachments to the sequent $\Gamma \Rrightarrow \Delta$.

The general indeterminate case for $UDF_i \equiv \forall x_1 : S_1, \cdots, \forall x_k : S_k, P_1 \vee \neg Q_1 \vee \cdots, \neg Q_n$ can be described by considering a ground term substitution $Sb \equiv x_1 \rightsquigarrow G_1, x_2 \rightsquigarrow G_2, \cdots, x_k \rightsquigarrow G_k$ such that all but one of the predicates $P_1, P_2, \cdots, P_m, Q_1, \cdots, Q_m$ counter-matches appropriate ground clauses in Γ or Δ. Suppose P_u is the uncounter-matched predicate clause. If P_u is indeterminate, the set of its variables will not be found within counter-matched clauses. That is, for $V, W : \wp(\{x_1, \cdots, x_k\})$, $P_1, \vee \cdots \vee P_{u-1} \vee P_{u+1} \vee \cdots \vee \neg Q_n\lceil V \rceil$, and $P_u\lceil W \rceil$, $W \sim V \neq \varnothing$. Additionally, if $P_u \equiv R(T_1, T_2, \cdots, T_h)$, and T_t is a legitimate target term for being unmatched then for $i :_{\mathbb{N}} [1 \cdots h]$ with $Y_i : \wp(W)$ and $T_i\lceil Y_i \rceil$, $\bigcup_{i :_{\mathbb{N}} [1 \cdots h] \sim \{t\}} Y_i \equiv W$.

Suppose all counter-matched ground clauses are denoted by $C_j \equiv P_j[S_b]$ for $j : (_{\mathbb{N}}[1 \cdots m] \sim \{u\})$, and $D_j \equiv Q_j[S_b]$ for $j :_{\mathbb{N}} [1 \cdots n]$ then $\Gamma' \equiv \Gamma \sim \{D_1, \cdots, D_n\}$ and $\Delta' \equiv \Delta \sim \{C_1, \cdots, C_{u-1}, C_{u+1} \cdots, C_m\}$. For the uncounter-matched predicate clause P_u, all the ground terms $F_i \equiv T_i[S_b]$ for $i : (_{\mathbb{N}}[1 \cdots h] \sim \{t\})$ must occur somewhere in the sequent $\Gamma \Rrightarrow \Delta$ (but not necessarily the target ground term $F_t \equiv T_t[S_b]$). Then the VC sequent rule is:

$$\frac{\Gamma' \cup \{D_1, \cdots, D_n, R(F_1, F_2, \cdots, F_t, \cdots, F_h\} \Rrightarrow \Delta' \cup \{C_1, \cdots, C_{u-1}, C_{u+1}, \cdots, C_m\}}{\Gamma' \cup \{D_1, \cdots, D_n\} \Rrightarrow \Delta' \cup \{C_1, \cdots, C_{u-1}, C_{u+1}, \cdots, C_m\}}$$

So, in this less constrained situation, the sequent extension adds a new relationship R between subterms F_i, all but one of which were at least already under consideration.

5.4 Summary

Universally quantified theorems need to be routinely instantiated in proving VCs. From among the relevant theorems, picking ones that avoid proving of one sequent VC becoming proving of many is critical. The contiguous instantiation strategy discussed in the chapter is designed to achieves this goal, automatically and without relying on heuristics.

Chapter 6

A Specificationally Rich Data Abstraction

The specification and use of generic data abstractions make scalable software verification feasible, and most modern programming languages support the construction of reusable software components to encapsulate data abstractions. Several automated verification systems, such as those summarized in [33], have been developed to facilitate automated verification of component-based software.

A central argument in this dissertation is that for a component-based approach to scale, specifications of data abstractions that components encapsulate need to be written in a rich language. This chapter presents a compelling example. When specifications such as the one presented here are routinely developed and used, specialized procedures for verification for particular theories would be of limited use. A prover, such as the Uni-Prover in this dissertation, is required.

This chapter explains a formal specification of a novel, generic data abstraction for manipulating a *tree position* using RESOLVE specification language. The *tree position* facilitates navigation and modification of a tree structure while avoiding explicit references, making automated verification of code built using the specified trees plausible. The tree abstraction we present is generic, parameterized by the type of information the tree contains and the number of children for each node. It contains operations to modify and navigate the tree but does not include a search operation. However, provided operations in the abstraction can be used to realize various search trees

and implement a map data abstraction, as the authors have shown elsewhere in a thesis [43].

Figure (6.1) contains a UML diagram that illustrates relationships between different tree components. However, only the artifacts highlighted in gray are discussed in this chapter.

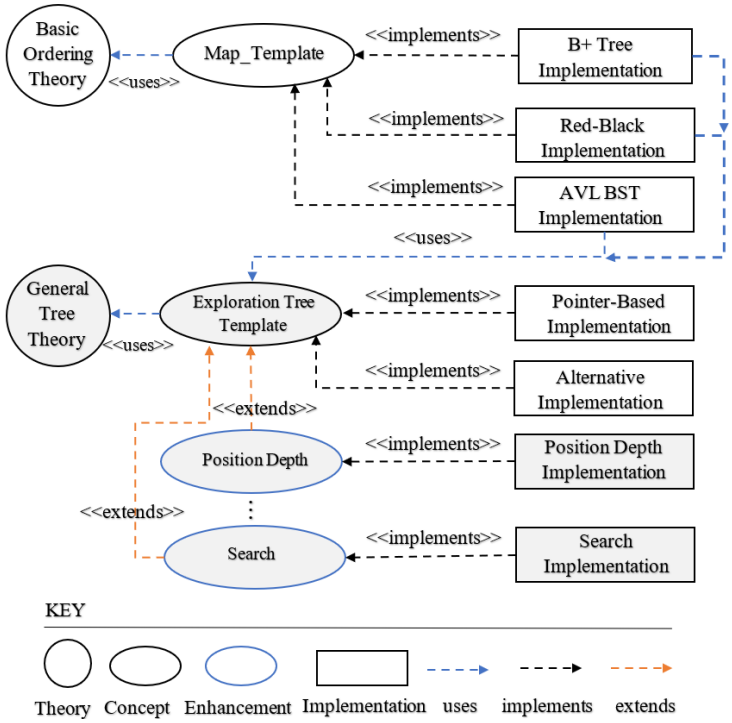
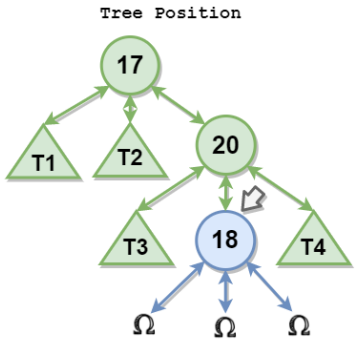


Figure (6.1) A UML diagram showing relationships of components in RESOLVE

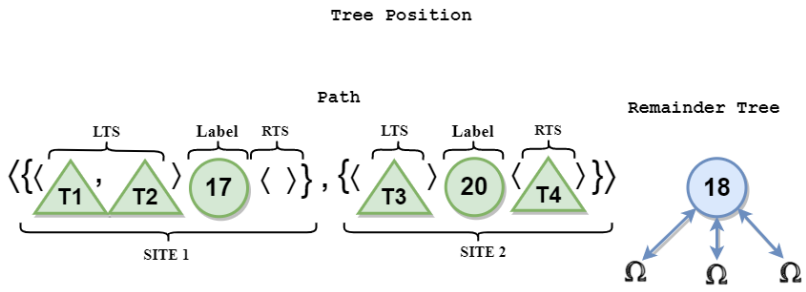
6.1 Encapsulating a Navigable Tree Position in a Data Abstraction

In imperative programming practice and literature, trees are realized via pointers—structural pointers between nodes and external pointers to nodes as tree positions. However, explicit manipulation of pointers is inherently complex and difficult to reason about [63, 38, 61]. To locally encapsulate this complexity, the use of reusable and comprehensible data abstractions is essential. The navigable tree data abstraction proposed here is surprisingly simple and presents a labeled tree and a single position within that tree where manipulations can be performed. While navigational operations allow access to other parts of the tree, other operations support changing the tree's shape and labeling. Verification of code underlying these operations do indeed involve pointers, but once verified,

the abstraction's reusability means that the upfront cost can be amortized over many deployments. In such deployments, the programmer's code is relieved from reasoning about explicit references, as would be necessary if traditional tree realizations were being manipulated.



(a)



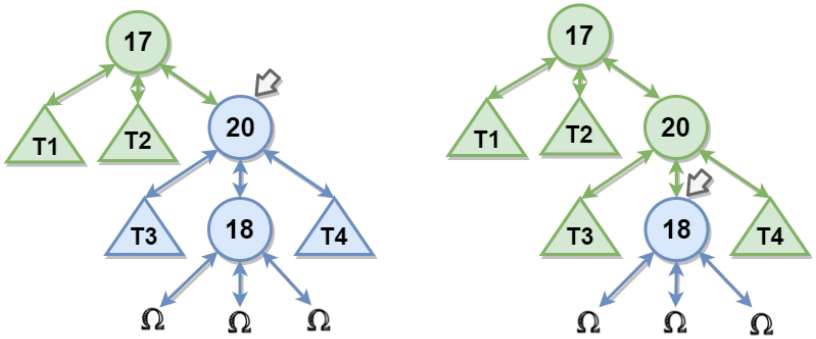
(b)

Figure (6.2) (a). An informal presentation of a *tree position*. (b). Formalization of a *tree position* showing a *path* with two *sites* and *remainder tree*

Figure (6.2) shows an example *tree position* for an instance of a generic tree abstraction we propose. The tree abstraction has been instantiated with integers as node labels and `3` as the maximum number of children for each node. The fat arrow in Figure 6.2(a) is used as an indicator of the current tree position. By the nature of trees, any particular position factors a tree into two disjoint parts—the *remainder tree* below the position indicator and a *path* before the position indicator. If a *path* includes all side branches to its left and right, then the underlying tree can be recovered. Further, movements between nearby positions can readily be specified. A tree position *path* contains *sites*. Each *site* records the label for that *path* position, the string of subtrees that lie to the left of the *path* (Left Tree String), and the string of subtrees that lie to the right of the *path* (Right Tree String). Figure 6.2(b) presents a formal illustration of the *tree position* with

the *remainder tree* shown in blue and the *path* in green. A formal discussion on a *tree position* is provided in Section 6.2.

A navigable tree data abstraction uses operations `Advance` and `Retreat` to navigate the exploration tree. These operations can move the current *tree position* to or from any one of the `k` immediate subtrees of the *remainder tree*. Operation `Advance` moves the current *tree position* to the next with an effect of adding one *site* to the *path*. Operation `Retreat` has an opposite effect to `Advance`. It reconstructs a new *remainder tree* using the last added *site* in the *path* and the current *remainder tree*, which is similar to navigating back to the previous *tree position*. For example, from a *tree position* in Figure 6.3(a), advancing to the next *tree position* in direction 2 will land us to a *tree position* in Figure 6.3(b), and we can retreat from *tree position* in Figure 6.3(b) to Figure 6.3(a).



(a) (b)

Figure (6.3) (a). A current *tree position* before advancing. (b). Updated *tree position* after advancing

The other operations included in the tree abstraction are `Add_Leaf` and `Remove_Leaf`. They create or modify trees by adding or removing a node from a tree. A new node is added to a tree using the operation `Add_Leaf`. For this operation to work, the *tree position* should be at the end, where the remainder tree is an empty tree (Ω) (see Figure 6.4(a)). Adding a new node, say with a label `16` at this position, results in a tree in Figure 6.4(b).

Only leaves are removable from a tree using the operation `Remove_Leaf`. While `Add_Leaf` and `Remove_Leaf` may seem restrictive, another operation, `Swap_Rem_Trees` (Swap Remainder Trees) allows the *remainder tree* of two different *tree positions* to be swapped in and out, essentially making it possible to add new nodes in the middle of the tree. Swapping is used rather than copying of references or values [23]. Swapping does not compromise abstract reasoning, and is efficient

for data movement as tree content in the generic abstraction can be of arbitrary type (not just `Integers`).

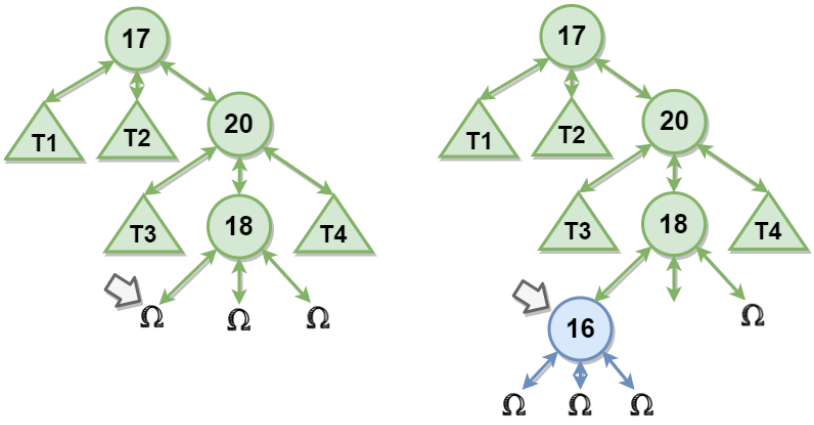


Figure (6.4) (a). An example *tree position* at an end. (b). Updated *tree position* with an added leaf

Boolean operations `At_an_End` (At an End) and `At_a_Leaf` (At a Leaf) are useful in checking the current *tree position* if it is at the end or at a leaf, respectively.

The specification for an exploration tree containing operations discussed above is provided in Listing (5). For space consideration, only the essential parts of the interface are shown. Defined as a **`Concept`** in RESOLVE, the interface describes the behavior of all primary operations necessary to make it useful. The parameters on each operation have a type and preceded by specification modes. For example, the parameter `dir` (direction) used in the `Advance` operation is of type `Integer`, and `P` is of type `Tree_Posn` (*tree position*). The operation `Advance` updates the *tree position* along the specified direction. The parameter mode **`updates`** is used along with `P` to explicitly specify its behavior before and after the operation is called. Other parameter modes in RESOLVE were discussed in Section 3.4.1.

```
Concept Exploration_Tree_Template( ... );
    ⋮
    Operation Advance( evaluates dir: Integer; updates P: Tree_Posn );
    Operation At_an_End( restores P: Tree_Posn ): Boolean;
    Operation Retreat( updates P: Tree_Pos );
    Operation Add_Leaf( (alters Labl: Node_Label;
                                          updates P: Tree_Posn );
```

```
    Operation Remove_Leaf( replaces Leaf_Labl: Node_Label;
                                          updates P: Tree_Posn )
    Operation At_a_Leaf( restores P: Tree Posn );
    Operation Swap_Label(updates Labl: Node_Label; updates P: Tree_Posn);
    Operation Swap_Rem_Trees( updates P, Q: Tree_Posn );
    ⋮
end Exploration_Tree_Template;
```

Listing (5) A skeleton version of the RESOLVE specification of an exploration tree

6.2 The Exploration Tree Data Abstraction

This section presents an exploration tree data abstraction written using a general tree theory containing mathematical theorems with definitions, notations, and predicates necessary to author succinct specifications. The general tree theory is explained in the next section.

6.2.1 A General Tree Theory

A significant challenge in writing theory for structures such as trees is the wide range of applications trees have in computing. Developing and verifying theories for each application will be costly. A practical solution is to develop a general theory for all applications. Generic theories are complex, and their development is not a trivial task. Nevertheless, their continuous reuse amortizes the up-front development and verification effort once developed and verified.

General Tree Theory used in this work is presented fully in Appendix D. This section covers a few definitions, predicates, and notations used in specifying the concept and operations in the coming section.

The first definition we present is a root label function `Rt_Lab`, which returns a root node label from a non-empty tree. Here is how it works, suppose we are given the *remainder tree* in Figure 6.3(a). `Rt_Lab` function will return an integer label 20 as a root node label.

A second function is root branches (`Rt_Brhs`), which extract all branches from the root node and return them as a string of trees. For the *remainder tree* in Figure 6.3(a), the root branches returned by this function will include `T3`, a tree with root node `18`, and `T4`.

A third function is Uniform Tree Positions (`U_Tr_Pos`), which returns a collection of all *tree positions* in a specified tree with `k` maximum number of children in a node and `Node_Label` as a node type.

A fourth function height (`ht`) is inductively defined to return an integer value representing the tree's maximum depth. For example, the height of an empty tree is 0.

Finally, Listing (6) defines a *site* and a *tree position*. A *site* is mathematically modeled as a Cartesian Product (`Cart_Prod`) of a node label (`Lab`) and a string of trees (`Str(Tr)`). A *tree position* is modeled as a `Cart_Prod` of a *path*, and the *remainder tree* (`Rem_Tr`).

```
Def. Site = Cart_Prod
        Lab: El;
        LTS, RTS: Str(Tr);
    end;
Def. Tree_Posn = Cart_Prod
        Path: Str(Site);
        Rem_Tr: Tr;
    end;
```

Listing (6) The mathematical definition of a *site* and a *tree position* in the theory

6.2.2 A Formal Specification of An Exploration Tree

The first part of the specification describing the behavior of an exploration tree is presented in Listing (7). The `Exploration_Tree_Template` is parameterized by a generic node type `Node_Label`, and integer values `k` and `Initial_Capacity`. The value `k` sets the maximum number of children for each node, and the `Initial_Capacity` places a shared upper bound on the number of nodes for all trees created within a single instantiation of the template.

```
Concept Exploration_Tree_Template( type Node_Label;
            evaluates k, Initial_Capacity: Integer);
   uses General_Tree_Theory....;
   requires 1 ≤ k and 0 < Initial_Capacity ..;

   Var Remaining_Cap: ℕ;
      initialization ensures Remaining_Cap = Initial_Capacity;
```

```
    Type Family Tree_Posn ⊆ U_Tr_Pos(k, Node_Label);
       exemplar P;
       initialization ensures P.Path = Λ and P.Rem_Tr = Ω;
       finalization ensures Remaining_Cap = #Remaining_Cap +
                                N_C(P.Path Ψ P.Rem_Tr);
            ⋮
end Exploration_Tree_Template;
```

Listing (7) RESOLVE concept for exploration tree

The specifications in Listing (7) are written using mathematical notations and definitions from different RESOLVE theories. The **uses** statement provides the concept with access to the `General_Tree_Theory` and any other theories used.

The **requires** clause next is specified to provide concept level restrictions on the values to be supplied by the client as parameters during an instantiation of the concept. The first clause ensures `k` is within the valid bounds, and no tree is created with zero children. The second clause is to ensure `Initial_Capacity` can never be zero.

The `Remaining_Cap` is a concept variable shared across all instantiated trees, and not each tree to be constrained individually to allow safe memory sharing [61]. The `Remaining_Cap` is specified as a natural number and initialized to `Initial_Capacity`.

The **Type Family** clause specifies `Tree_Posn`, a type exported by the template, as a subset of all uniform tree positions defined by `k` and `Node_Label`. The exported type is a whole family of types, each with different contents and not just one type, which emphasizes the generic nature of this abstraction.

The **exemplar** clause introduces a name `P` as an example *tree position* used in specifying **initialization** and **finalization ensures** clause. Initially, a created *tree position* `P` will have its *path* equal to an empty string (Λ) and its *remainder tree* equal to an empty tree (Ω). At the end of its use, the **finalization** clause guarantees that a count of tree nodes belonging to the tree object is added back to the *remaining capacity*. The initial or incoming *remaining capacity* is differentiated from the outgoing or final *remaining capacity* using a # symbol used as a prefix for the former.

A complete specification for an exploration tree should have all primary operations specified.

Some of these operations were mentioned in Listing 5, and their details are provided in [43]. The following is a detailed mathematical description for an operation `Advance`.

The operation `Advance` takes in the current *tree position* `P` and updates it to a new *tree position* depending on the advancing direction `dir`. The tree position `P` and direction `dir` are supplied as parameters preceded by modes specifying their behavior before and after the operation is called.

A mathematical specification describing the behavior of operation `Advance` is provided in Listing (8). The **`requires`** clause uses two conditions to specify the operation's behavior before behavior before the user calls it. The first condition restricts the operation to only nonempty *remainder trees* because advancing to an empty tree is impossible. The second condition requires the supplied integer value `dir` to be within valid bounds as only advancing from a node to one of its branches is possible.

```
Operation Advance( evaluates dir: Integer; updates P: Tree_Posn );
   requires P.Rem_Tr ≠ Ω and 1 ≤ dir ≤ k
   ensures P.Rem_Tr = ⧙( Prt_Btwn(dir − 1, dir, Rt_Brhs(#P.Rem_Tr)))
        and P.Path = #P.Path o ⟨( Rt_Lab(#P.Rem_Tr),
        Prt_Btwn(0, dir − 1, Rt_Brhs(#P.Rem_Tr)),
        Prt_Btwn(dir, k, Rt_Brhs(#P.Rem_Tr)) )⟩;
```

Listing (8) Formal specification of an `Advance` operation

The **`ensures`** clause specifies two postconditions describing what happens after the operation is called. Two parts are updated, the first condition updates the *remainder tree*, and the second condition updates the *path*. Each part is described separately next.

The first part of **`ensures`** clause uses a function `Rt_Brhs` to return a string containing all branches of a root node for a given tree. A *Part Between* operator (`Prt_Btwn`) will return a substring between two specified intervals of a given string. In the first part, the `Prt_Btwn` operator extracts a needed tree branch between two directions, `dir - 1` and `dir`. The tree branch extracted is in a string format, and a *destring* (⧙[1]) operator is used to produce the entry in a singleton string.

The second part updates the *tree position* by adding a new *site* to its existing *path*. The new

[1]Current RESOLVE compiler parses only ASCII characters, so all mathematical characters used are converted to ASCII equivalents

site is a result of advancing into the *remainder tree* leaving behind a root node label specified by `Rt_Lab(#P.Rem_Tr)` and the string of branches on the left and right of the advancing direction. The left branches are specified by `Prt_Btwn(0, dir – 1, Rt_Brhs(#P.Rem_Tr))` and the right branches are specified by `Prt_Btwn(dir, k, Rt_Brhs(#P.Rem_Tr))`.

6.2.3 Concept Enhancements

We present `Position_Depth_Capability` specified in Listing (9) as one of the enhancements that extend the application of our tree data abstraction. The goal of this enhancement is to find the longest path from the root node to one of the empty trees in the *remainder tree*.

```
Enhancement Position_Depth_Capability for Exploration_Tree_Template;
   Operation Position_Depth ( restores P: Tree_Posn): Integer;
        ensures Position_Depth = ( ht(#P.Rem_Tr) );
end Position_Depth_Capability;
```

Listing (9) An example specification a RESOLVE enhancement

The mathematical description in Listing (9) only specifies the postcondition as the operation is called on *tree position* `P`. The operation returns an integer value representing the height of `P`'s *remainder tree*, as specified in the **`ensures`** clause.

The `Position_Depth_Capability` enhancement is implemented in Listing (10). The implementation uses both recursion and iteration, and to establish their correctness, the implementer annotated the code with termination (**`decreasing`**) clause and loop invariant (**`maintaining`**).

```
Realization Obv_Rcsv_Realiz for Position_Depth_Capability
                                of Exploration_Tree_Template;
   Recursive Procedure Position_Depth(restores P: Tree_Posn): Integer;
      decreasing ht(P.Rem_Tr);
      Var PrevDir, NextHeight, MaxBrHeight: Integer;
      If (At_an_End(P)) then
         Position_Depth := 0;
      else
         PrevDir := 0;
         NextHeight := 0;
         MaxBrHeight := 0;
```

```
            While ( PrevDir < k )
               maintaining P.Path = #P.Path and P.Rem_Tr = #P.Rem_Tr and
                   MaxBrHeight = Ag(max, 0, ht[[Prt_Btwn(0, PrevDir,
                   Rt_Brhs(P.Rem_Tr))]]) and PrevDir <= k;
               decreasing k - PrevDir;
            do
               Increment(PrevDir);
               Advance(PrevDir, P);
               NextHeight := Position_Depth(P);
               If (MaxBrHeight < NextHeight) then
                  MaxBrHeight := NextHeight;
               end;
               Retreat(P);
            end;
            Position_Depth := MaxBrHeight + 1;
         end;
      end Position_Depth;
   end Obv_Rcsv_Realiz;
```

Listing (10) Implementing `Position_Depth_Capability` enhancement

The **`maintaining`** clause is specified as a conjunction of four main parts. The first two parts guarantee that the *tree position* is returned to its initial state at the end of the operation. The third part maintains the Maximum Branch Height (`MaxBrHeight`) on every iteration. This part is defined using an aggregate function (`Ag`), which has three arguments. The first argument, `max`, is a binary operator that returns a maximum value between two given values. The second argument is 0, a base height for an empty string. The third argument is `ht` is the height function applied to a string of trees found through a function `Rt_Brhs`. The extra square brackets [[...]] used with the function `ht` implies `ht` is applied to each tree in the string.

6.3 VC Generation Progress

The RESOLVE verification system has made considerable progress towards VC generation of implementations such as one in Listing (10). The VCs are not discharged as yet, and this work is

a progress towards the verification object based software components such as one presented in this chapter.

For the implementation provided in Listing (10), 40 VCs were successfully generated for verification. After a visual inspection of every VC, Table (6.1) was created to illustrate their distribution. We classified 24 VCs as mathematically trivial for verification. These VCs can quickly be verified from the antecedents (**Givens**) in a few steps using the theory of equality. The second class has 10 VCs; these are more challenging, requiring algebra and additional theorems to establish their correctness. The last 6 VCs are too costly to establish their correctness in few steps. A detailed classification on sequent VCs and their provability is presented in Section 1.4.1.

Table (6.1) A Breakdown of Generated VCs for Listing 10

Reason for VCs	Generated VCs	Mathematically Trivial VCs	More Challenging VCs	Too Costly to Establish
requires Clause	8	5	3	0
ensures Clause	8	5	3	0
Base Case	12	12	0	0
Inductive Case	8	0	2	6
Termination	4	2	2	0
Total	**40**	**24**	**10**	**6**

Two sequent VCs below are selected to illustrate the results of our experimentation. The first VC is shown in Listing 11 and it corresponds to an inductive case of one loop invariant conjuncts. In the verification process, the generated VCs are supplied to an automated prover. With the assistance of theorems from the library, VCs are formally verified by the prover. The validity of the received VC is established when any of the **Goal** assertions can be proved by the known facts from the **Given** assertions (which includes relevant theorems from the library). For the VC in Listing 11 it turns out to be trivial, and its correctness can be established by **Given** #3, which proves the first goal assertion.

```
Goal:
((1 + PrevDir'') ≤ k) or
    (1 + Ag(max, 0, ht[[ Prt_Btwn(0, PrevDir'',
    Rt_Brhs(P'''.Rem_Tr))]]) ≤ ht(P.Rem_Tr)) or (P.Rem_Tr = Empty_Tree)
```

```
Given:
1. (P''.Rem_Tr = DeString(Prt_Btwn(((1 + PrevDir'') − 1), (1 + PrevDir''),
    Rt_Brhs(P'''.Rem_Tr))))
```

```
2. (P''.Path=(P'''.Path o ⟨Rt_Lab(P'''.Rem_Tr)⟩ o ⟨Prt_Btwn(0,(1+PrevDir''−1),
    Rt_Brhs(P'''.Rem_Tr))⟩ o ⟨Prt_Btwn(1 + PrevDir'',k,Rt_Brhs(P'''.Rem_Tr)))))
3. ((1 + PrevDir'') ≤ k)
4. (P'''.Path = P.Path)
5. (P'''.Rem_Tr = P.Rem_Tr)
6. (PrevDir'' ≤ k)
7. (1 ≤ k)
8. (k > 0)
```

Listing (11) A VC generated from the inductive case of the loop invariant

The second VC presented is generated from the **decreasing** clause of the **While** loop to show termination. Compared to the previous one, this VC shown in Listing 12 would require simple algebra to the first assertion in the **Goal** to establish its correctness.

```
Goal:
((1 + (k − (1 + PrevDir''))) ≤ (k − PrevDir'')) or
    (1 + Ag(max, 0, ht[[ Prt_Btwn(0, PrevDir'',
    Rt_Brhs(P'''.Rem_Tr))]]) ≤ ht(P.Rem_Tr)) or (P.Rem_Tr = Empty_Tree)
```

```
Given:
1. (P''.Rem_Tr = DeString(Prt_Btwn(((1 + PrevDir'') − 1),
    (1 + PrevDir''), Rt_Brhs(P'''.Rem_Tr))))
2. (P''.Path = (P'''.Path o ⟨Rt_Lab(P'''.Rem_Tr)⟩) o
    ⟨Prt_Btwn(0, (1 + PrevDir'' − 1), Rt_Brhs(P'''.Rem_Tr))⟩ o
    ⟨Prt_Btwn((1 + PrevDir''), k, Rt_Brhs(P'''.Rem_Tr))))
3. ((1 + PrevDir'') ≤ k)
4. (P'''.Path = P.Path)
5. (P'''.Rem_Tr = P.Rem_Tr)
6. (PrevDir'' ≤ k)
7. (1 ≤ k)
8. (k > 0)
```

Listing (12) A VC generated to establish termination of the loop

6.4 Summary

Generation of sequent VCs is a big step toward verification of the code. The challenge faced by verification systems when proving implementation, such as one in Listing (10), is the use of `Exploration_Tree_Template` specified using the non-trivial General_Tree_Theory for which there are no special-purpose solvers. The current version of the RESOLVE automated prover cannot handle the verification of all sequent VCs either due to its limitations, motivating the universal automated prover for atomic sequents discussed in this work.

Chapter 7

Congruence Class Registry Overview

Congruence Class Registry is at the core of the Uni-Prover. This chapter motivates its need, explains its design, and presents a brief discussion on the concept that describes its behavior. The mathematics necessary to describe the concept formally and a formal specification are the topics of subsequent chapters.

7.1 Motivation

The primary goal of using the congruence class registry is to deal with equality predicates in sequent VCs effectively. Equalities are pervasive in math specifications and theories, and for effectiveness in verifying programs, they must be handled differently from other predicates.

7.1.1 Why Equality Cannot Be Treated Like Any Other Predicate

The automated prover may use rules of logic to prove sequent VCs and, when necessary, can use mathematical theorems from the library to prove each sequent VC in the collection. While using theorems from the library is reasonable for other predicates, the general approach is inefficient for equalities because of their pervasiveness—an excessive number of steps that would be taken when using equality theorems from the library.

Consider two equality predicates `a = b` and `b = c`. If equality is not handled specially, and recovered from a theory in the library, deducing that `a = c` would require drawing the transitivity theorem from the library and instantiating it properly before we can conclude `a = c`. It is contrary to

how humans would deal with such simple equality. The knowledge of equality accumulated over time allows humans to deduce simple equalities from a given set of equations quickly. For our example, it is easy for humans to look at `a = b` and `b = c`, and quickly deduce `a = c`. Using a general, mechanized approach that goes through many steps to establish even simple equalities undesirable.

The main problem with building the theory of equality into the library and treating it like any other predicate is the excessive number of steps taken, even for straightforward conclusions. Besides, selecting theorems from the library tend to bring all other related theorems into the process. These additional theorems lead to an extensive collection of equality facts added to the sequent being verified. Most of these facts would not contribute to the proof of the sequent, and including them introduces mostly unnecessary steps.

Additionally, equalities are characterized by equivalence and congruence properties, which, when employed, affect all terms in the sequent VC introducing many irrelevant equality facts. Those irrelevant equality facts clutter the sequent VC, potentially causing extra steps that do not contribute to the final result. The following example demonstrates the problems.

Consider $\{a = b, b = c, d = c, e = d + f\} \Rrightarrow \{F(a + f) = F(e)\}$, a sequent VC with four equalities in the antecedent. One crucial equality fact needed for verification of this sequent VC is $F(a + f) = F(a + f)$, which results from applying the reflexivity property of equality on every term in the sequent VC. As a result, in addition to $F(a + f) = F(a + f)$, other irrelevant equality facts from all other terms are simultaneously added as shown in Figure (7.1). These extra equality facts do not contribute to the proof, and they cause even more facts to be added when other equality rules are applied.

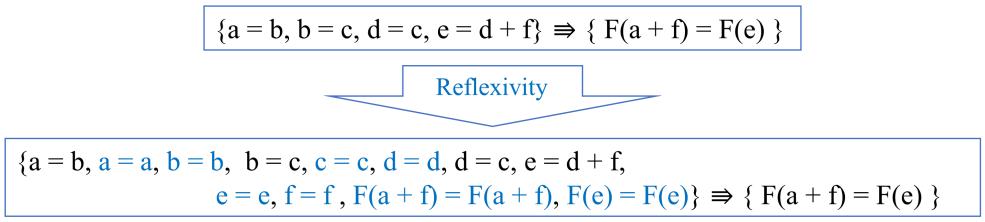


Figure (7.1) Applying reflexivity property from the library to a sequent VC

Additional equalities come from congruence relations of functions. In the presence of func-

tions in sequents, the equality facts about their arguments can be congruently related through function application. For example, for a single-argument function `f(a)`, knowing `a = b`, we can deduce `f(a) = f(b)` through function application. So both equivalence and congruence properties introduce more equalities in the sequent.

7.1.2 Efficient Handling of Equality Using Congruence Classes

Because of the equivalence and congruence properties of equality, terms can be partitioned into congruence classes. The congruence classes efficiently establish all equalities and their consequences, without having to instantiate and use equality results from the theory library. Additionally, congruence class components can be efficiently implemented. Therefore, we can leverage the properties of congruence classes and deal with equality predicates in sequent VCs by partitioning their terms into congruence classes. The following is a simple example of how effectively congruence classes deal with equality and its consequences.

Consider $\{\mathtt{a} = \mathtt{b}, \mathtt{b} = \mathtt{c}\} \Rrightarrow \{\mathtt{F(a)} \geq \mathtt{F(c)}\}$, a sequent VC with two equality predicates in the antecedent. The congruence class partition in Figure (7.2) is produced by applying a congruence closure on the sequent. Each term in the sequent is initially in its congruence class, and the classes collapse as the equalities are introduced. For example, when `a = b` is introduced into the partition, $\{\mathtt{a}\}$ and $\{\mathtt{b}\}$ merges to $\{\mathtt{a}, \mathtt{b}\}$. Any change in the partition can cause other classes to be updated accordingly. Thus, $\{\mathtt{F(a)}\}$ and $\{\mathtt{F(b)}\}$ that used to be in separate classes, collapse to a single class $\{\mathtt{F(a)}, \mathtt{F(b)}\}$ as a consequence of $\{\mathtt{a}, \mathtt{b}\}$. With congruence classes, the knowledge of `a` and `b` being equal is kept by the fact that `a` and `b` are in the same class.

{ {a}, {b}, {c}, {F(a)}, {F(c)}, {F(a) ≥ F(c)} }
{ {a, b}, {c}, {F(a), F(b)}, {F(c)}, {F(a) ≥ F(c), F(b) ≥ F(c), } }
⋮
{ {a, b, c}, {F(a), F(b), F(c)}, {F(a) ≥ F(c), F(b) ≥ F(c), F(c) ≥ F(c), F(a) ≥ F(a),
F(a) ≥ F(b), F(b) ≥ F(a), F(b) ≥ F(b), F(b) ≥ F(c), F(c) ≥ F(a)} }

Figure (7.2) Dealing with equalities using congruence classes

After `b = c` is introduced into the partition, and all classes are updated, the final partition in Figure (7.2) has the necessary capability to establish other related equalities quickly. Compared to using equality theory in the library, all equalities are established using the partition *in a single step*. In the example above, we can immediately establish `a`, `b`, and `c` as equal terms because they

are in the same congruence class.

Congruence classes also deal with the unnecessary clutter in a sequent VC on applying the equality properties. For example, equalities like `a = b` and `b = a` must appear in the sequent VC when applying the symmetric property. They are now represented by a single class $\{\texttt{a}, \texttt{b}\}$ in the congruence class partition.

We have developed the congruence class registry that leverages the ability of congruence classes to deal with all equalities efficiently. The challenges are in integrating congruence classes with the sequent calculus and in developing elaboration rules to support all the manipulations necessary to prove a sequent VC. The following sections describe the congruence class registry and its central role in Uni-Prover.

7.2 Multi-Level Organization of Congruence Classes

Congruence classes in the registry are collections of trees, and searching is not straightforward as it would be with individual trees. So using a congruence class registry in turn introduces a non-trivial searching challenge. The search process must be as effective as possible to minimize the number of steps taken to prove sequent VCs.

To facilitate effective searching, we have organized the registry into four levels. The organization avoids the investigation of every tree entity in the registry as it would be in regular tree searching. The following sections discuss these levels, and explain how they make searching effective.

7.2.1 Congruence Classes and Varieties

Figure (7.3) illustrates the congruence class registry with classes and subclasses. The collections of trees registered from expressions and subexpressions in the sequent VC and elaboration rules are on the right of the diagram. As explained in Section 7.1.2, initially, singleton congruence classes with single trees are created in the registry. When equalities are registered, congruence classes with subexpressions known to be equal will collapse into a single congruence class containing trees from both classes.

Congruence classes presented in red circles represent one of the levels used in the registry organization. Each class in the registry is designated with a unique number called a congruence class designator (`C_Cls_Dsntr`).

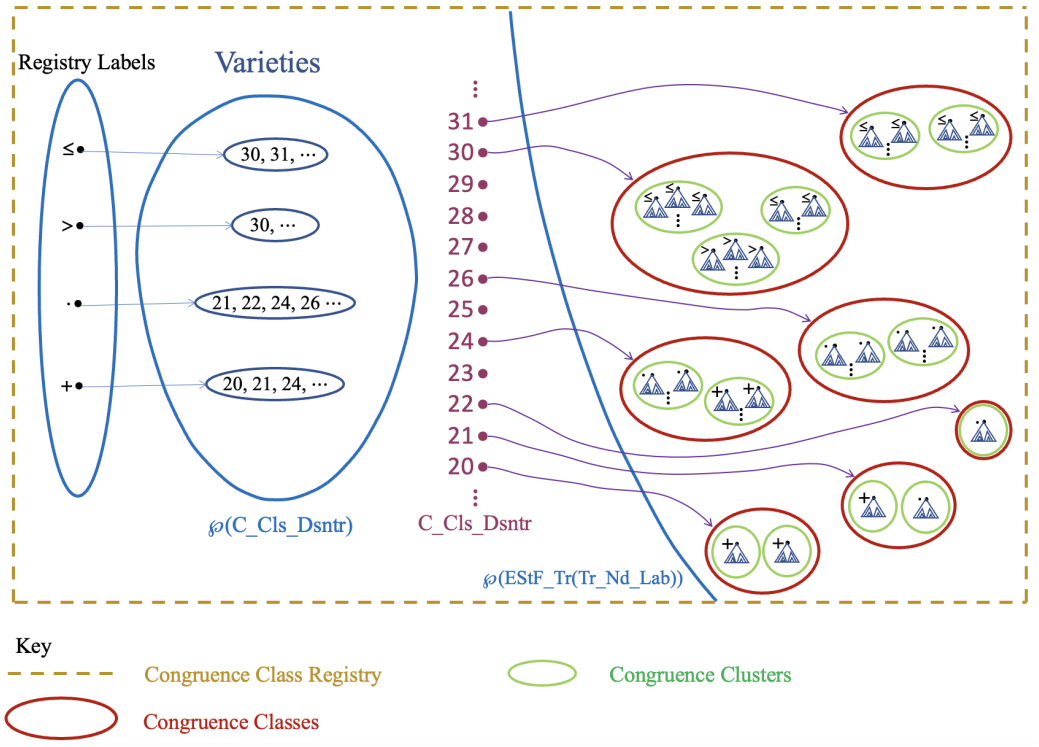


Figure (7.3) Congruence class varieties, classes, and clusters

To minimize the potential classes to be searched, classes are organized into *varieties*. This organization keeps an ordered list of all congruence classes containing at least one tree with the root node label designating a variety. Varieties are shown on the left of Figure (7.3). Each registry label determines a single variety. For ≤, the variety contains congruence class `30,31` and any other class that has a tree with ≤ at the root. When the class searched falls under a variety, searching is narrowed to only classes within the variety, and all other varieties are eliminated.

7.2.2 Congruence Clusters and Stands

For even more effective searching and to support counter-matching of clauses, we introduce another level in the registry organization called congruence clusters to further refine the classes.

If we denote the congruence class for a tree representing a term `T` as `C_Cls(T)`, and its congruence cluster by `C_Clstr(T)`, then the key property of congruence clusters is that for any

function f and g at the root of two terms with subtree terms R, S, T and U is the following:

If C_Clstr(f(R, S)) = C_Clstr(g(T, U)),

then f = g, **and** C_Cls(R) = C_Cls(T) **and** C_Cls(S) = C_Cls(U)

Congruence clusters being more refined than congruence classes means the following:

If C_Clsrt(S) = C_Clstr(T), **then** C_Cls(S) = C_Cls(T)

A critical observation on the cluster organization: Clusters are the most primitive refinement. They are partitions and respect each other's boundaries. They do not overlap and have a downward completeness property where everything is in a cluster. If a tree is in a cluster, its subtrees must be in some clusters too. These properties are essential in searching. It means that by searching the lower level like clusters, we are still searching for the same things in the higher level, just in smaller chunks and without crossing the boundaries of the higher level. This property explains the exhaustiveness in these levels, where going through the lower level one by one will eventually get through the higher level, making the searching efficient by not losing anything. Congruence clusters are designated in the registry with unique numbers called congruence clusters designators (Clstr_Dsntr), as shown in green in Figure (7.4).

The organization in the registry provides yet another opportunity to further confine our search to clusters with similar "root node labels" (Rt_Lab) that we call congruence class *stands*. Stands are shown in Figure (7.4). They group clusters that contain trees with the same root node label in a congruence class.

If we denote congruence class stand as C_Stand, then for terms S and T, stands have the property that C_Stand(S) = C_Stand(T) if and only if C_Cls(S) = C_Cls(T), and Rt_Lab(S) = Rt_Lab(T). Therefore, if C_Clstr(S) = C_Clstr(T), then C_Stand(S) = C_Stand(T).

Figure (7.4) shows several stands designated by the classes and registry labels. When searching a congruence class, a stand with clusters having trees with the root node label being searched is retrieved first. Only clusters within the identified stand are checked, and other clusters in the class are eliminated from the search. For example, searching for a tree with a root node $\leq$ limits our search to only clusters 55 and 53 in the first stand shown in the diagram.

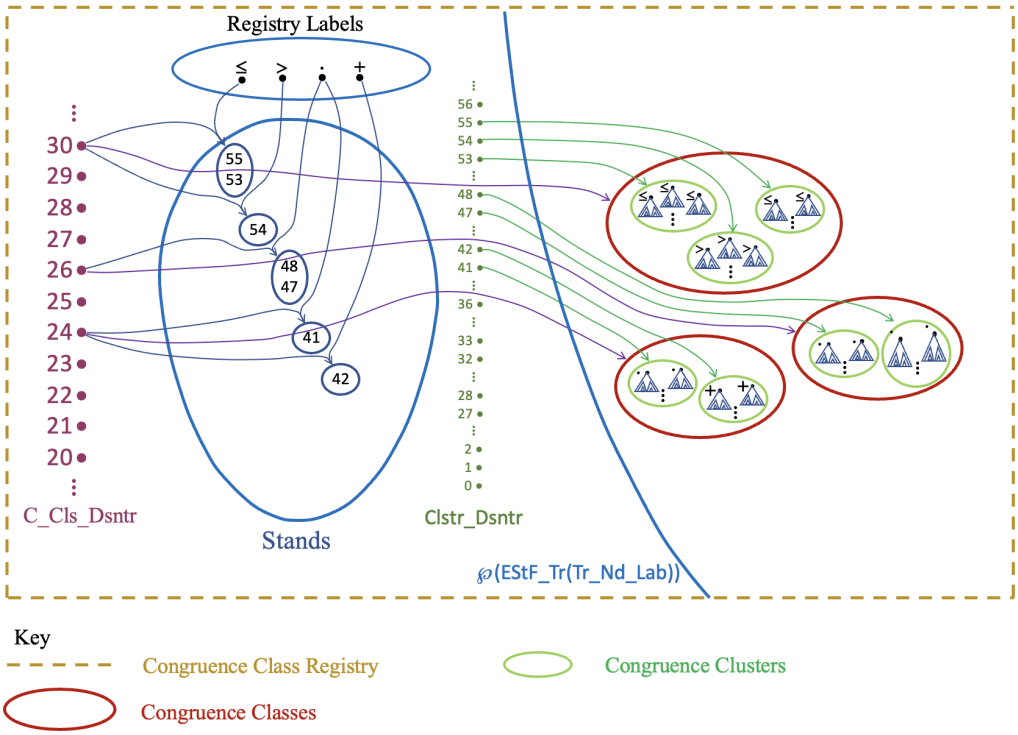


Figure (7.4) Congruence class registry stands

Each level in the registry serves a purpose, and collectively they make searching in the registry more effective. Taken together, the registry concept is capable of supporting operations to prove a sequent VC.

7.3 Summary of the Congruence Class Registry Concept

We have detailed the congruence class registry template in Chapter 9. This section briefly discusses the congruence class registry template. It provides an introduction of key details at an informal level.

The congruence class registry concept is parameterized, and a *client* should supply types and values to instantiate it. The client in the context of this section is the component that uses the congruence class registry. Listing (13) shows a few of these parameters starting with tree node label (`Tr_Nd_Lab`) and tree category tag(`Tr_Cgry_Tag`), which are types. For example, if Integer is used as the tree node label type, then variables, constants, and operators in sequent VCs, which are represented as strings, will be converted into integers. The other two parameters set

the maximum number of congruence classes (`C_Cls_D_Cap`) and clusters (`Clstr_D_Cap`) for the registry. Additional parameters and details are discussed later in Chapter 9.

The specifications written in the concept use definitions and notations defined in the library of RESOLVE theories, including the general tree theory. All used theories are brought into the scope through the concept **`uses`** statement.

The uses statement is followed by a concept level requires clause, which sets some boundaries to the values provided through the parameters. For example, the two assertions in the requires clauses provided in Listing (13) set a lower bound for `C_Cls_D_Cap` and `Clstr_D_Cap` provided by the client.

The following definitions specify two values used in the registry to designate congruence class and clusters. Mathematically, both `C_Cls_Dsntr` and `Clstr_Dsntr` are defined as a finite set of natural numbers in an interval from `0` to the respective maximum designator capacity.

7.3.1 Registry Types and Abstraction

The central type exported by the concept is the congruence class registry (`CC_Reg`) type. But to add entries to the registry and search, handles are needed for classes and clusters. So first we discuss the handle types that we term designators.

Designators for classes and clusters are only usable inside the registry. Outside, clients are provided accessors, allowing them access to registered clusters and classes. Without the introduction and use of these types, direct pointers or numbers must be exported to client for manipulation and that would violate abstraction.

Accessors and designators must be described separately without complicating reasoning as accessors will not be changing, but designators change as congruence classes and clusters get merged.

We have achieved a clean separation between designators and accessors by having the concept exporting two types, `C_Cls_Accessor` (Congruence Class Accessor) and `Clstr_Accessor` (Cluster Accessor). As shown in Listing 13, `C_Cls_Accessor` and `Clstr_Accessor` are specified as a subset of `C_Cls_Dsntr` and `Clstr_Dsntr`, respectively. Even though accessors are specified as numbers, the only operations allowed are `Replica` for creating copies and `Are_Equal` for equality check. Clients cannot perform any other arithmetic operations on the accessors.

Another aspect of this separation is to allow more than one congruence class registry to be created, and in a parallel processing environment, the exported accessors will be available to clients

for each created registry.

The congruence class registry (`CC_Reg`) type is modeled mathematically as a Cartesian Product (`Cart_Prod`) of different elements. Listing (13) only mention a few of them. The first is a function `CC_Designated`, mapping congruence class designators to created congruence classes. The second, `Top_CC_Dsntr`, keeps track of the most recently used class designator. The third is a function `Cr_Designated` used to map cluster designators to clusters. The recently used cluster designator is also kept in `Top_Cr_Dsntr`. The rest of the elements of this mathematical tuple can be found in Chapter 9.

```
Concept Congruence_Registry_Temp(type Tr_Nd_Lab, Tr_Cgry_Tag;
                                 eval C_Cls_D_Cap, Clstr_D_Cap, .... );
        uses Relativization_Ext for General_Tree_Theory;
        requires 0 < C_Cls_D_Cap and 0 < Clstr_D_Cap ... ;
    Def. C_Cls_Dsntr: ℘_fin(ℕ) =_N [0...C_Cls_D_Cap];
    Def. Clstr_Dsntr: ℘_fin(ℕ) =_N [0...Clstr_D_Cap];
    Type_Family C_Cls_Accessor ⊆ C_Cls_Dsntr;
    Oper Replica ( restores c: C_Cls_Accessor ): C_Cls_Accessor;
    Oper Are_Equal( restores c, d: C_Cls_Accessor ): Boolean;

    Type_Family Clstr_Accessor ⊆ Clstr_Dsntr;
    Oper Replica ( restores p: Clstr_Accessor ): Clstr_Accessor;
    Oper Are_Equal( restores c, d: Clstr_Accessor ): Boolean;

    Type_Family CC_Reg ⊆
        Cart_Prod
            CC_Designated: C_Cls_Dsntr⟶℘(EStF_Tr(Tr_Nd_Lab)),
            Top_CC_Dsntr: C_Cls_Dsntr,
            Cr_Designated: Clstr_Dsntr⟶℘(EStF_Tr(Tr_Nd_Lab))
            Top_Cr_Dsntr: Clstr_Dsntr,
                        ⋮
        end;
    Abbnl Def. C_Class (Rg: CC_Reg): ℘_fin(℘(EStF_Tr(Tr_Nd_Lab))) = (...);
                —Registry operations are shown in upcoming listings—
end Congruence_Registry_Temp;
```

Listing (13) Congruence class registry concept types

7.3.2 Concept Conciseness and Comprehensibility

Conciseness and comprehensibility of the concept are necessary to make it usable. The following have been considered in the specifications to reduce the mental effort in understanding the non-trivial registry concept specifications for both clients and implementers of the congruence class registry concept.

First, reducing the number of pieces of the mathematical `CC_Reg` model is essential in making the concept usable. The model of the state of the registry contains only what is essential, independent, and sufficient to directly or indirectly (through definitions in the theories and those local to the concept) specify everything needed in the registry. For example, while the registry has four levels, they are a refinement (derivative) of each other, and only congruence classes and clusters are necessary to describe the registry's state fully. The varieties and stands can be described entirely using classes and clusters. This property has allowed us to export only two accessors to the client, and they are sufficient to provide full access to the registry.

Second, we have introduced abbreviational (`Abbnl`) definitions that are defined locally in the concept to replace parts of the specifications that appear several times throughout the concept. With abbreviational definitions, only their names appear on other parts of the concept. For example, abbreviational definition `C_Class` described in Listing 13 allows us to use only the name `C_Class` instead of entirely writing out what it defines in every occurrence in the concept.

7.4 Operations to Register a Cluster

The concept describes primary operations to modify the state of the registry, navigate the registry, observe its status, and get values from the registry. An exhaustive list of operations is provided in Appendix C. This section presents the first set of primary operations in Listing (14) used for registration.

```
Oper Register_Cluster_Lbld(preserves Lab:Tr_Nd_Lab, replaces
    c:C_Cls_Accessor, alters atb: Tr_Cgry_Attbt, updates Rg:CC_Reg);
```

```
Oper Is_Already_Reg_Clstr( preserves Lab: Tr_Nd_Lab,
                                    restores Rg: CC_Reg): Boolean
Oper Get_Accr_for( preserves Lab: Tr_Nd_Lab,
                   replaces c: C_Cls_Accessor, updates Rg: CC_Reg );
Oper Append_to_Arg_Lst( restores c: C_Cls_Accessor,
                   updates Rg: CC_Reg affecting_only Clstr_Arg_Lst);
```

Listing (14) Congruence class registering operations

The key operation in this set is `Register_Cluster_Lbld`, which creates a new congruence class and cluster given a tree node label and arguments appended in the argument list. It returns an accessor to the new addition through the replaces mode parameter. The other three operations are used together with `Register_Cluster_Lbld` in the registration process. The following is an example registration for illustration.

Consider a sequent VC $\{a + b \leq 9 \cdot a, a + b = 8\} \Rightarrow \{8 \leq 9 \cdot a\}$ in Figure (7.5) placed in a nested list form. We can process the sequent from left to right to register it. The clause $a + b \leq 9 \cdot a$ will be registered first, starting with `a`, followed by `b`, and building the expression bottom-up. Let us assume `a` and `b` are already registered, and $a + b$ is the term being registered, as shown in Figure (7.5). The client must check the clusters existence in the registry using the operation `Is_Already_Reg_Clstr` before registration. If already registered, its class accessor can be retrieved using operation `Get_Accr_for`. In most cases, accessors are retrieved to be used in registering an operator as we build up an expression bottom up. Figure (7.6) shows a snippet of the RESOLVE code in a component using the operations in Listing (14) called before the plus operator is registered.

The code uses an **`if`** condition in lines `8` and `14` to check if `a` and `b` are registered. In the case they are registered, only their accessors are retrieved and stored in `c` and `d`, respectively. Otherwise, they are registered, and their accessors are returned by replacing the values in `c` and `d` accordingly. The argument list is left empty when `a` and `b` are registered, as variables have no arguments. However, to register a plus operator, `c` and `d` must be appended to the argument list before the register operation is called. On line `23`, a check is made to see if the cluster for a plus operator with arguments `c` and `d` is already registered. In the example, the cluster is new and is registered on line `26`.

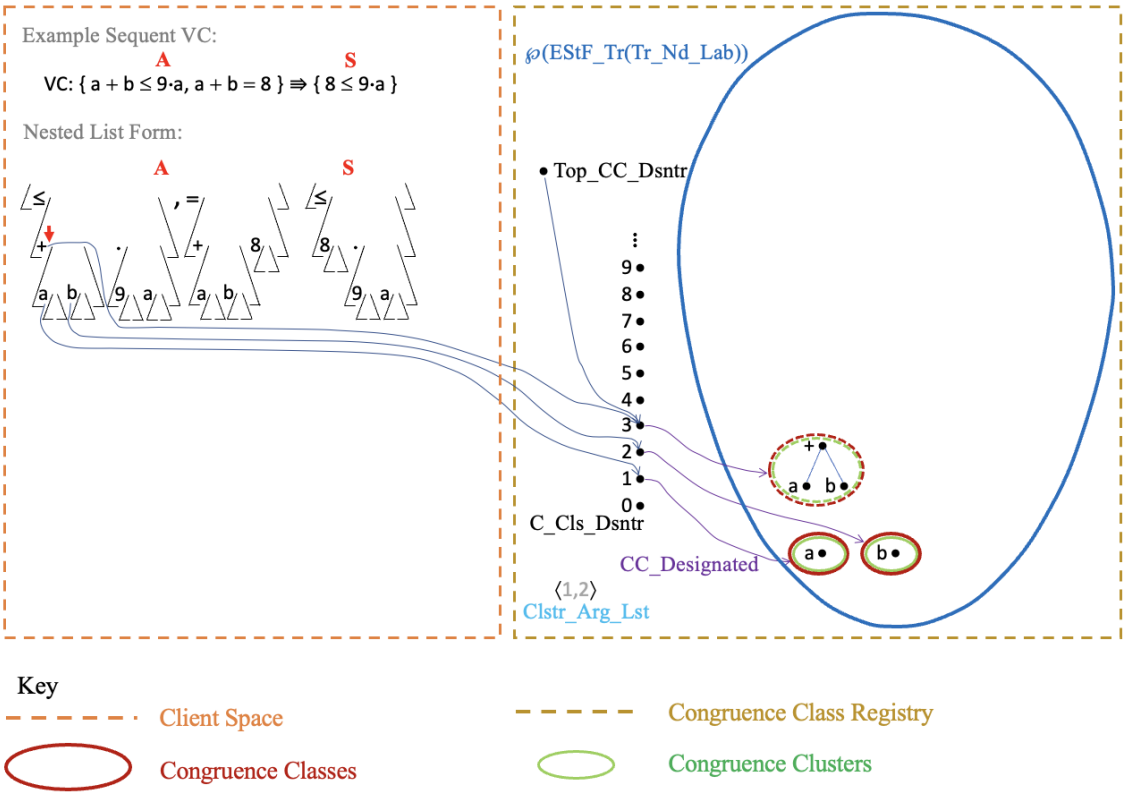


Figure (7.5) Registering a + operator with a and b as arguments

7.5 Congruence Operations

Listing (15) shows two congruence operations specified in the concept to deal with equality predicates in the antecedent of the sequent VC. The first operation `Make_Congruent` takes two congruence class accessors for two classes that are known to be equal and merge them into a single class. The second operation `Are_Congruent` checks if two provided congruence accessors are already congruent.

```
Oper Make_Congruent (restores c, d: C_Cls_Accessor, updates Rg: CC_Reg );
Oper Are_Congruent ( restores c, d: C_Cls_Accessor,
                                    restores Rg: CC_Reg ): Boolean;
```

Listing (15) Congruence operations in the registry

The two operations can be used together to ensure that operation `Make_Congruent` is only called with non-congruent class accessors. To illustrate this use, consider the same sequent VC $\{a + b \leq 9 \cdot a, a + b = 8\} \Rightarrow \{8 \leq 9 \cdot a\}$ used above, this time we proceed with the registration of

```
7     --Code omitted for brevity--
8     If (Is_Already_Reg_Clstr('a', Rg)) then
9         Get_Accr_for('a', c, Rg);
10    else
11        Register_Cluster_Lbld('a', c, A, Rg);
12    end;
13
14    If (Is_Already_Reg_Clstr('b', Rg)) then
15        Get_Accr_for('b', d, Rg);
16    else
17        Register_Cluster_Lbld('b', d, A, Rg);
18    end;
19
20    Append_to_Arg_Lst(c, Rg);
21    Append_to_Arg_Lst(d, Rg);
22
23    If (Is_Already_Reg_Clstr('+', Rg)) then
24        Get_Accr_for('+', e, Rg);
25    else
26        Register_Cluster_Lbld('+', e, A, Rg);
27    end;
28    --code omitted for brevity--
```

Figure (7.6) RESOLVE code to register a term $a + b$

`a + b = 8`. Figure (7.7) shows the effect of calling the `Make_Congruent` operation on two classes, one for + operator and another for a constant `8`.

The accessor for `a + b` and `8` must be retrieved from the registry by the client. Figure (7.8) shows a RESOLVE code using the `Make_Congruent` operation. The code retrieves the needed accessors using the `Get_Accr_for` operation first and call `Make_Congruent` if and only if `Are_Congruent` returns false.

7.6 Searching Operations

Searching the registry involves mainly moving from one class to another and from one cluster to another. The operations in Listing (16) are specified to traverse and search the registry.

```
Oper Advance_CC_Accr_for( restores x: Tr_Nd_Lab,
                    updates c:C_Cls_Accessor, restores Rg:CC_Reg);
Oper Is_Vrty_Maximal_for(restores x:Tr_Nd_Lab, restores c: C_Cls_Accessor,
                                          restores Rg:CC_Reg):Boolean;
Oper Advance_Clstr_Accr_for( restores x: Tr_Nd_Lab,
                    restores c:C_Cls_Accessor, updates p: Clstr_Accessor,
                                                restores Rg:CC_Reg);
Oper Is_Stand_Maximal_for( restores x: Tr_Nd_Lab,
                restores c: C_Cls_Accessor, restores p: Clstr_Accessor,
                                        restores Rg: CC_Reg ): Boolean;
```

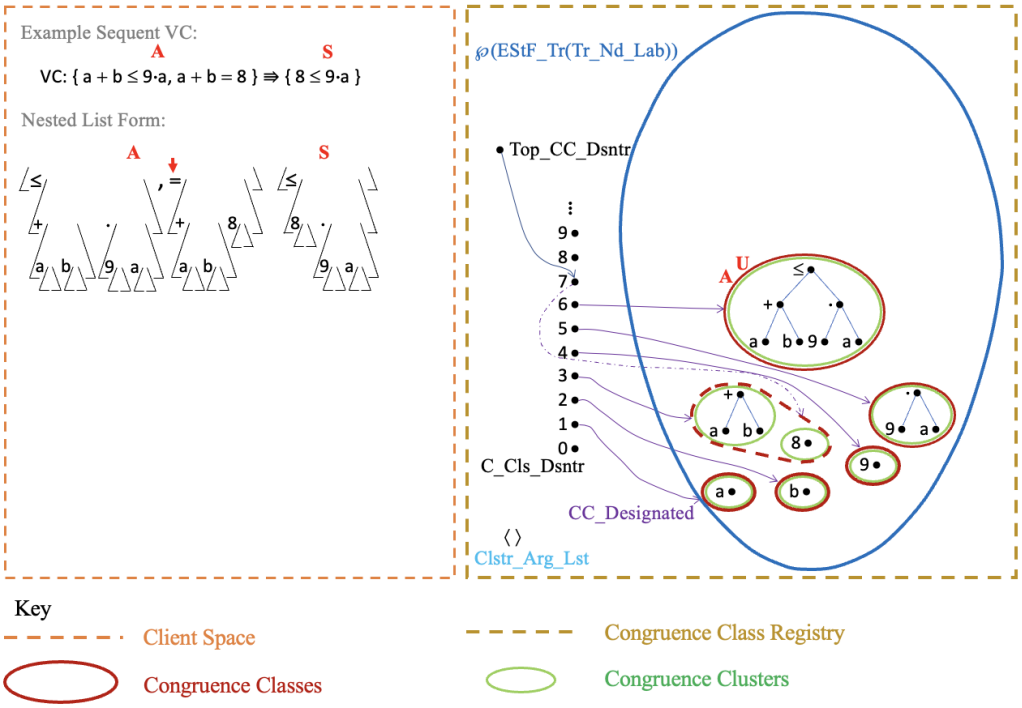


Figure (7.7) Registering an equality predicate $a + b = 8$

```
31    --Code omitted for brevity--
32     Get_Accr_for('+', c, Rg);
33     Get_Accr_for('8', d, Rg);
34
35     If (not Are_Congruent(c, d, Rg)) then
36         Make_Congruent(c, d, Rg);
37     end;
38     --code omitted for brevity--
```

Figure (7.8) RESOLVE code using `Make_Congruent` operation

Listing (16) Registry searching operations

The search for a precursor tree in the registry starts at the root, where the right class and cluster are located first before moving to the next level. The precursor tree in Figure (7.9) has a ≤ at the root. Therefore, a variety designated by ≤ is retrieved, and one class after another is searched. To advance between classes within a variety, `Advance_CC_Accr_for` operation is normally used within a loop. The loop continues until we find the right cluster or the classes are exhausted in a variety. A boolean operation `Is_Vrty_Maximal_for` returns true when classes in the variety are exhausted and can be used as a loop condition.

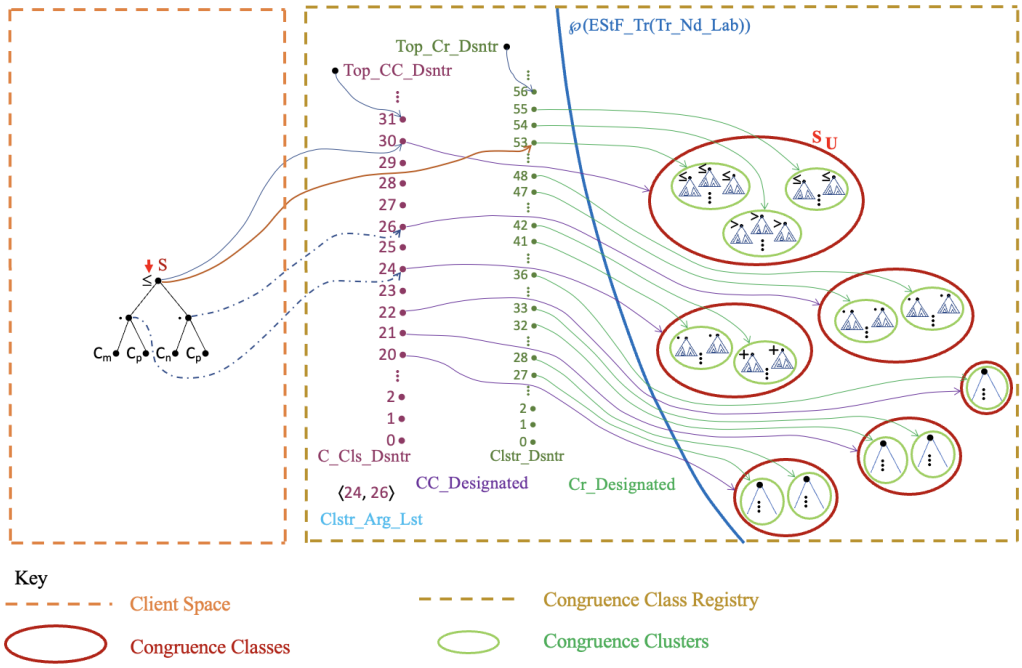


Figure (7.9) Searching a precursor tree in the registry

Clusters inside a class are searched to match the precursor tree root and root branches. The root on the precursor tree in Figure (7.9) needs to match a cluster with a root label ≤, and its branches should have classes with trees having a product operator as a root. Therefore, a stand for ≤ in the first class of ≤ variety is searched starting with the first cluster. If the cluster does not match what we are looking for, the next one is checked. Moving from one cluster to another within a stand is facilitated by operation `Advance_Clstr_Accr_for`. To check if the clusters in the stand are exhausted, a boolean operation `Is_Stand_Maximal_for` is used.

7.7 Summary

This chapter has motivated the need for a congruence class registry and summarized the concept that specifies its behavior informally. The formal specification of the concept in detail is provided in Chapter 9. The next chapter describes mathematical developments used in the formal specification of the registry.

Chapter 8

Reusable Registry Mathematics

This chapter contains the mathematical definitions that underlie the formal specification of the congruence class registry in the next Chapter 9. The mathematics we introduce are intended to be abstract, simple, and general to achieve the comprehensibility and reusability in the specifications where they are used. Specific properties for each definition are stated in Appendix B.

This chapter is meant to be a reusable mathematical unit that can be used in any specification. The motivation and purpose of many definitions will likely be obvious only after reading the next chapter where they are used. So a co-reading of these two chapters is unavoidable.

8.1 Partition

The design of our congruence class registry is based on the fact that the congruence classes are partitions. The classes are independent and do not overlap. So in this section, we define partitions, congruence partitions, complete congruence partitions, and sub-partitions.

8.1.1 Refines Relationship

The following definition is used in defining sub-partition in Section 8.1.5. The refines relationship holds between two sets of sets in which all elements in one set are found in the other other set.

A set `K` refines `L` when a union of elements `A` in `K` is equal to a union of elements `C` in `L`, and

for any A and C, either A and C are independent, or A is contained in C.

Def.(K: $\wp(\wp(\mathtt{S}\colon \mathbb{Set}))$ Refines (L: $\wp(\wp(\mathtt{S}))$): $\mathbb{B}$ = ($\bigcup_{\mathtt{A}:\mathtt{K}} \mathtt{A} = \bigcup_{\mathtt{C}:\mathtt{L}} \mathtt{C}$

and $\forall \mathtt{A}\colon \mathtt{K}, \forall \mathtt{C}\colon \mathtt{L}, \mathtt{A} \cap \mathtt{C} = \varnothing$ **or** $\mathtt{A} \subseteq \mathtt{C}$);

8.1.2 Set Partition

A partition is a set of sets where each set in the partition is independent and does not overlap others. The predicate Is_Partition_of is a Boolean operator that specifies P as a partition of S if a union of all cells C in P equals S, and for any arbitrary set A and B in P, they are either disjoint or equal. This predicate is useful in defining a congruence partition in the following section.

Def : Is_Partition_of($\mathtt{S} : \mathbb{Set}, \mathtt{P} : \wp(\wp(\mathtt{S})) \sim \{\{\varnothing\}\}$) : $\mathbb{B}$ = ($\bigcup_{\mathtt{C}:\mathtt{P}} \mathtt{C} = \mathtt{S}$ **and**

$\forall \mathtt{A}, \mathtt{B} : \mathtt{P},\ \mathtt{A} \cap \mathtt{B} = \varnothing$ **or** $\mathtt{A} = \mathtt{B}$)

8.1.3 Congruence Partition

A congruence partition contains independent congruence classes. The registry in this work contains congruence classes, which are collections of trees—trees that denote clauses of atomic sequents. So a congruence partition of a collection of trees with more specific properties is defined next.

The predicate Is_Congruence_Partition is a Boolean operator defined for the partition P defined as a finite set of trees. In this case, P is a congruence partition if, first of all, it is a partition, and secondly, if any tree exists in a partition, then there is a class in the partition that contains it. This definition becomes helpful in defining a complete congruence partition in the following section.

Def : Is_Congruence_Partition($\mathtt{P} : \wp_{\mathtt{fin}}(\wp(\mathtt{EStF\_Tr}(\Gamma : \mathbb{Set})))$) : $\mathbb{B}$ = (

Is_Partition_of($\bigcup_{\mathtt{C}:\mathtt{P}} \mathtt{C}, \mathtt{P}$) **and** $\forall \alpha : \mathtt{Str}(\mathtt{P}), \forall \mathtt{F} : \Gamma$, **if** $\mathtt{Jn}[\mathtt{Ag}_{(\mathbb{L}^{\circ},\{\Lambda\})}(\langle [[[\alpha]]] \rangle), \{\mathtt{F}\}] \cap \bigcup_{\mathtt{C}:\mathtt{P}} \mathtt{C} \neq \varnothing$,

then $\exists \mathtt{D} : \mathtt{P} \ni \mathtt{Jn}[\mathtt{Ag}_{(\mathbb{L}^{\circ},\{\Lambda\})}(\langle [[[\alpha]]] \rangle), \{\mathtt{F}\}] \subseteq \mathtt{D}$);

8.1.4 Complete Congruence Partition

A complete congruence partition defines a property for trees in the registry where if a class for the root is in a congruence partition, so are classes for its root branches. Classes for trees in the registry are built bottom-up, and the completeness property ensures the lower-level classes in the tree are congruence partitions before the level next up.

The predicate `Is_Cplt_Congr_Prttn` specifies a partition `P` as a complete congruence partition if, first of all, `P` is a congruence partition, and secondly, for any tree created with a node and branches among them `T`, if the resulting tree is in the partition, then all branches are in the partition, including `T`.

This predicate is used in the registry constraints specification to ensure that the congruence classes defined locally as an abbreviation in Section 9.4.1 have the completeness property.

Def : `Is_Cplt_Cngr_Prttn`(P: $\wp(\wp($`EStrF_Tr`(Γ: $\mathbb{Set}))))$: $\mathbb{B}$ = (

`Is_Congruence_Partition`(P) **and** $\forall \beta, \gamma :$ `Str`(`EStrF_Tr`(Γ)), $\forall$ G: Γ, T: `EStrF_Tr`,

if `Jn`$(\beta \circ \langle$`T`$\rangle \circ \gamma,$ G$) \in \bigcup_{\text{C: P}}$ C, **then** T $\in \bigcup_{\text{C:P}}$ C);

8.1.5 Sub-partition

The registry contains congruence classes kept in levels defined in Section 8.2. The levels have the property that they refine each other. Refinement is defined in Section 8.6 and used here to define sub-partition.

Generally, a sub-partition of a partition defines a partition for a refinement in sets of sets. The predicate `Is_Subpartition_of` is an infix operator, which specifies a set `K` as a sub-partition of set `L` if `K` and `L` are both partitions, and `K` refines `L`.

This predicate is used in the specification of constraints presented in Section 9.5, and specify that the lower levels in the registry are always a sub-partition of higher levels.

Def : (K: $\wp(\wp($S : $\mathbb{Set})))$`Is_Subpartition_of`(L: $\wp(\wp($S$)))$: $\mathbb{B}$ = (

`Is_Partition_of`$(\bigcup_{\text{A:K}}$A, K$)$ **and** `Is_Partition_of`$(\bigcup_{\text{A:K}}$A, L$)$ **and** K `Refines` L);

8.2 Congruence Clusters, Stands, and Varieties

The definitions here formally describe the formation of clusters, stands and varieties, which are useful in specifying operations involving those notions in the registry.

8.2.1 Cluster Formation

Clusters in the registry are at the lowest level in the organization and refine stands. A cluster is created from a root node and trees on the root branches. Therefore, given a set of trees, we can formulate a collection of all possible clusters from root nodes and root branches. The operator `Cluster_from` returns this collection of clusters from a collection of trees. This operator is useful in specifying registry operations `Register_Cluster_Lbld`, `Is_Already_Reg_Clstr`, and `Get_Accr_for` presented in Section 9.6. The `Cluster_from` operator is also used in defining `Prpr_Cluster` in the next section.

Def : $\texttt{Cluster\_from}(\texttt{R}\colon \wp(\wp(\texttt{EStF\_Tr}(\Gamma\colon \mathbb{Set}))), \alpha\colon \texttt{Str}(\texttt{R}), \texttt{F}\colon \Gamma)\colon \wp(\texttt{EStF\_Tr}(\Gamma)) = ($

$$\texttt{Jn}[\texttt{Ag}_{(\mathbb{L}^{\circ},\{\texttt{A}\})}(\langle[[[\alpha]]]\rangle), \{\texttt{F}\}]);$$

8.2.2 Proper Cluster

The `Cluster_from` operator returns a collection of all possible clusters from the trees in the registry. This collection may contain clusters that do not belong in the registry. On the contrary, a proper cluster must be in the registry.

The definition of a `Prpr_Cluster` is more restrictive compared to `Cluster_from`. It includes a condition that all clusters `P` created using `Cluster_from` must be in the registry `R`. This operator is used in the specification of registry concept constraints presented in Section 9.5.

Def : $\texttt{Prpr\_Cluster}(\texttt{R}\colon \wp(\wp(\texttt{EStrF\_Tr}(\Gamma\colon \mathbb{Set}))))\colon \wp(\wp(\texttt{EStrF\_Tr}(\Gamma))) = \{$

$$\texttt{P}\colon \wp(\texttt{EStrF\_Tr}(\Gamma)) \mid \exists\ \alpha\colon \texttt{Str}(\texttt{R}), \exists\ \texttt{F}\colon \texttt{Rt\_Lab}[\textstyle\bigcup_{\texttt{C:R}} \texttt{C}] \ni \texttt{Cluster\_from}(\texttt{R}, \alpha, \texttt{F}) = \texttt{P}$$

$$\textbf{and } \texttt{P} \cap \textstyle\bigcup_{\texttt{C:R}} \texttt{C} \neq \varnothing\};$$

8.2.3 Stand Formation

Stands refine congruence classes, and they are formed by a collection of clusters of trees with the same root label. A `Stand_from` operator is defined to return a set of all clusters having the same root node label `F` from a provided set of trees. The operator `Stand_from` is used in the next definition.

Def. $\texttt{Stand\_from}(\texttt{C}\colon\ \wp(\texttt{EStrF\_Tr}(\Gamma\colon\ \mathbb{Set})), \texttt{F} : \Gamma)\colon\ \wp(\texttt{C})\ =\ \{\texttt{T}\colon\ \texttt{C}\ \mid\ \texttt{Rt\_Lab}(\texttt{T}) = \texttt{F}\};$

The operator `Stand_for` returns a nonempty set of stands. Each stand is described by the `Stand_from` operator applied at a set level on `R` and a set of root node labels on each `C` in `R`. This operator is also used in the specification of registry concept constraints presented in Section 9.5.

Def. $\texttt{Stand\_for}(\texttt{R}\colon \wp(\wp(\texttt{EStrF\_Tr}(\Gamma : \mathbb{Set})))) : \wp(\wp(\texttt{EStrF\_Tr}))) = ($

$$\texttt{Stand\_from}[\texttt{R}, \bigcup_{\texttt{C:R}} \texttt{Rt\_Lab}[\texttt{C}]] \sim \{\{\varnothing\}\}]);$$

8.2.4 Variety Formation

At the top most level in the registry are the verieties refined by congruence classes. A variety is formed by a collection of classes with trees having the same root label. The `Variety_from` operator is defined to form a collection of classes with a tree `T` having a root node label `F`. The operator `Variety_from` is used in defining `Variety_for` next.

Def. $\texttt{Variety\_from}(\texttt{D}\colon \wp(\texttt{EStF\_Tr}(\Gamma\colon \mathbb{Set})), \texttt{F}\colon \Gamma) : \wp(\texttt{D}) = \{\texttt{T}\colon \texttt{D} \mid \texttt{Rt\_Lab}(\texttt{T}) = \texttt{F}\};$

The operator `Variety_for` returns a set of all varieties from a collection of trees `D` for each root label available in `D`. The collection is defined using `Variety_from` operator applied at a set level on a set of trees and root node labels in `D`. This operation is also used in the specification of registry concept constraints presented in Section 9.5.

Def. $\texttt{Variety\_for}(\texttt{D}\colon \wp(\texttt{EStF\_Tr}(\Gamma\colon \mathbb{Set})))\colon \wp(\texttt{D}) = (\texttt{Variety\_from}[\{\texttt{D}\}, \texttt{Rt\_Lab}(\texttt{D})]);$

8.3 Index Sets

The classes and clusters in the congruence class registry will be mapped from specific designators. While classes and clusters are designated differently, they are refinements of each other and their designators map to the same set of trees. At an abstract level, these designators are indices in a mapping function. The next two definitions `Idx_Rfln` and `Mnml_Idx_Rfln` describe the relation between the two index sets.

Stands in the registry are designated with a pair containing a root label and a class. Each pair maps to stand. Similarly, clusters are designated by a cluster designator, and each designator maps to a cluster. Because every stand contains a list of clusters, we can create a new mapping between each stand designator and a set of cluster designators it contains. This new mapping is defined by the operator `Idx_Rfln`, which is used in the next definition `Mnml_Idx_Rfln`.

Def. $\texttt{Idx\_Rfln}(\texttt{ER}\colon (\texttt{I}\colon \mathbb{Set}) \to \wp(\texttt{S}\colon \mathbb{Set}), \texttt{E}\colon (\texttt{J}\colon \mathbb{Set}) \to \wp(\bigcup_{\texttt{i}\colon \texttt{I}} \texttt{ER}(\texttt{i}))) \colon \texttt{J} \to \wp(\texttt{I}) =$

$$\lambda \texttt{j}\colon \texttt{J}.\{\texttt{i}\colon \texttt{I} \mid \texttt{ER}(\texttt{i}) \cap \texttt{E}(\texttt{j}) \neq \varnothing\};$$

The operator `Mnml_Idx_Rfln` is used in specifying abbreviational definitions in the registry to describe the mapping from designators in cases where merging happens and more than one designator points to the same thing in the registry. This definition introduces ordering in those designators.

Compared to the `Idx_Rfln` operator defined above, `Mnml_Idx_Rfln` introduces the notion of well-ordering on the index sets the operator `Idx_Rfln` is used to define. This definition is used in specifying abbreviational operator `Mnml_VCC_Dsntr` presented in Section 9.4.6.

Def. $\texttt{Mnml\_Idx\_Rfln}(\texttt{ER}\colon (\texttt{I}\colon \mathbb{Set}) \to \wp(\texttt{S}\colon \mathbb{Set}), \texttt{E} : (\texttt{J} : \mathbb{Set}) \to \wp(\bigcup_{\texttt{i:I}} \texttt{ER}(\texttt{i})),$

$$\preceq : \texttt{Wl\_Ordng})\colon \texttt{J} \to \wp(\texttt{I}) = \lambda \texttt{j} : \texttt{J}.\{\texttt{i}\colon \texttt{Idx\_Rfln}(\texttt{ER}, \texttt{E})(\texttt{j}) \mid \forall \texttt{h}\colon \texttt{Idx\_Rfln}(\texttt{ER}, \texttt{E})(\texttt{j}),$$

$$\textbf{if } \texttt{ER}(\texttt{i}) = \texttt{ER}(\texttt{h}), \textbf{ then } \texttt{i} \preceq \texttt{h}\};$$

8.4 Merging in the Registry

The definitions in this section are provided to specify an operation `Make_Congruent` in Section 9.6.2. The operation makes two classes known to be equal in the registry congruent, affecting mapping functions. The definitions here describe how the classes relate to each other after merging and how functions are updated.

The operation `Mrg_Val_at` explains the behavior of mapping functions after two sets mapped from different domains are merged to form a new set. This operation finds its use in specifying `Make_Congruent` in Section 9.6.2, which merges two classes to a single class.

The operator takes an old mapping function that maps the two indices to different sets and returns a new mapping function with both indices mapping to the union of the two sets.

Def. $\texttt{Mrg\_Val\_at}(\texttt{c},\texttt{d}\colon (\texttt{D}\colon \mathbb{Set}), \texttt{F}\colon \texttt{D} \to \wp(\texttt{S}\colon \mathbb{Set})) : \texttt{D} \to \wp(\texttt{S}) =$

$$\lambda \texttt{a}\colon \texttt{D}.\left(\left\{ \begin{array}{ll} \texttt{F(c)} \cup \texttt{F(d)} & \textbf{if } \texttt{F(a)} \cap (\texttt{F(c)} \cup \texttt{F(d)}) \neq \varnothing \\ \texttt{F(a)} & \textbf{otherwise} \end{array} \right\}\right)$$

The operation `Mrg_Val_at` identifies two mapping functions, an old mapping before two sets merge and a new mapping after merge. The Subset function ($\sqsubseteq_{\texttt{SF}}$) is true when the set created by the old function is the subset of the set created by the new function. The subset function is also used in the specification of `Make_Congruent` in Section 9.6.2.

Def.$(\texttt{F}\colon (\texttt{D}\colon \mathbb{Set}) \to \wp(\texttt{S}\colon \mathbb{Set})) \sqsubseteq_{\texttt{SF}} (\texttt{G}\colon \texttt{D} \to \wp(\texttt{S})) : \mathbb{B} = (\forall \texttt{d}\colon \texttt{D}, \texttt{F(d)} \subseteq \texttt{G(d)})$;

In the registry, merging two classes may cause a cascade of other classes collapsing as other classes become congruent. The behavior of the mapping function as the registry goes through this cascade of class merges is defined by the operator `CCD_Rpr_Fnal`.

This operator uses `Ptrn_Indcd_by` and `Clr_Ext` to describe repaired function `F` as the classes are merged. Definitions `Prtn_Indcd_by` and `Clr_Ext` are provided in Section 8.6.

Def. $\texttt{CCD\_Rpr\_Fnal}(\texttt{F}\colon (\texttt{D}\colon \mathbb{Set}) \to \wp(\wp(\texttt{EStrF\_Tr}(\Gamma\colon \mathbb{Set}))) : \texttt{D} \to \wp(\wp(\texttt{EStF\_Tr}(\Gamma))) =$

$$\lambda \texttt{d}\colon \texttt{D}.(\texttt{Prtn\_Indcd\_by}(\texttt{Clr\_Ext}(\texttt{F}));$$

We can define more properties for the two functions before and after merging classes in the registry using the predicate `SubDsgnts`. In this case, the first mapping function (`E`) sub-designates the second mapping function (`F`) if the non-empty set created by the first function is a subset of the non-empty set created by the second function. This predicate is also helpful in specifying `Make_Congruent` operation in Section 9.6.2.

Def. (E: (I: $\mathbb{Set}$) → $\wp(\wp(\mathtt{S}: \mathbb{Set} \sim \varnothing))$ `SubDsgnts` (F: I → $\wp(\wp(\mathtt{S}: \mathbb{Set} \sim \varnothing))$: $\mathbb{B}$ =

$(\forall \mathtt{i}: \mathtt{I}, \mathtt{E}(\mathtt{i}) \subseteq \mathtt{F}(\mathtt{i}))$;

8.5 Factorization

This section contains definitions that are used in specifying consistency in tagging. Every class and cluster will be tagged with extra information to aid in searching the registry. The client relates clusters and classes to tags. While in the registry, designators are either mapped to tags with one function, and mapped to to classes and clusters in other mapping. We provide two definitions that describe a cross-mapping needed state consistency in tagging from the two parallel mapping functions with the same domain. The resulting function creates a mapping between their ranges.

The predicate `Is_L_Fctbl_for` is a Boolean operator that defines two mapping functions, `G` and `H`, with the same domain `D` as left factorable for any domain `c` and `d` only if the function `G` maps `c` and `d` to the same range value, then `H` also maps `c` and `d` to the same range value.

Def. `Is_L_Fctbl_for`(G: (D: $\mathbb{Set}$) → (R: $\mathbb{Set}$), H: D → (S: $\mathbb{Set}$): $\mathbb{B}$ = (

$\forall \mathtt{c}, \mathtt{d}: \mathtt{D}$, **if** $\mathtt{G}(\mathtt{c}) = \mathtt{G}(\mathtt{d})$, **then** $\mathtt{H}(\mathtt{c}) = \mathtt{H}(\mathtt{d})$);

The domain $\mathbb{LLF}\mathtt{n}$ is defined as a set containing a pair of left factorable functions `G` and `H` and used as a domain in an implicit definition of Left Factor (`LFctr`) provided below.

Def. $\mathbb{LFF}\mathtt{n}$: $\wp(\mathbb{F}\mathtt{n} \times \mathbb{F}\mathtt{n}) = \{\prec \mathtt{G}, \mathtt{H} \succ : \mathbb{F}\mathtt{n} \times \mathbb{F}\mathtt{n} \mid \mathtt{Is\_L\_Fctbl\_for}(\mathtt{G}, \mathtt{H})\}$

Given that `G` and `H` are left fuctorable functions, we can provide an implicit definition of a left factor (`LFctr`) as a function that maps an image of `G` to an image of `H` where for any `d` in

the domain of G, the value a function LFctr maps to, given G(d) is equal to the value a function H maps d. This function and the predicate Is_L_Fctbl_for are used in the specification of registry concept constraints presented in Section 9.5.

Implicit Def. LFctr((≺G, H≻): $\mathbb{LFF}$m): Im(G) → Im(H) **is**

$$\forall \mathtt{d}\colon \mathtt{Dom(G)}, \mathtt{LFctr}(\prec \mathtt{G}, \mathtt{H} \succ)(\mathtt{G(d)}) = \mathtt{H(d)};$$

8.6 Auxiliary Definitions

The definitions in this section do not appear directly in the congruence class registry specifications but are used in defining congruence class repair functional in Section 8.4.

8.6.0.1 Connected Via

The operator Cnctd_via defines a set of all D's such that there exists a string α such that any two Q's say E and F, which happens to be consecutive and forms a string ⟨E⟩ ∘ ⟨F⟩. If ⟨E⟩ ∘ ⟨F⟩ is a substring of ⟨C⟩ ∘ α ∘ ⟨D⟩, then their intersection should be nonempty if C and D are connected. This definition is used in Prtn_Indcd_by defined next.

Def. Cnctd_via(Q: ℘(℘(S: $\mathbb{Set}$)), C: Q): ℘(Q) = {

$$\mathtt{D}\colon \mathtt{Q} \mid \exists \alpha\colon \mathtt{Str(Q)} \ni \forall \mathtt{E}, \mathtt{F}\colon \mathtt{Q},\ \textbf{if} \langle \mathtt{E} \rangle \circ \langle \mathtt{F} \rangle\ \mathtt{Is\_Substring}\ \langle \mathtt{C} \rangle \circ \alpha \circ \langle \mathtt{D} \rangle,\ \textbf{then}\ \mathtt{EIF} \neq \varnothing\};$$

8.6.0.2 Partition Induced By

From a function F that maps D to S, Prtn_Indcd_by returns an updated F, where for any domain d in set D, a map is created to everything connected to F(d) via a set defined by applying F to every element in D (F[D]).

Def. Prtn_Indcd_by(F: (D: $\mathbb{Set}$) → ℘(S: $\mathbb{Set}$)): D → ℘(S) =

$$\lambda \mathtt{d}\colon \mathtt{D}.\left(\bigcup_{\mathtt{A:Cnctd\_via(F[D],F(d))}} \mathtt{A}\right);$$

8.6.0.3 Induced Expansion

The operator `Indcd_Exp` of a mapping function `F` given `R` returns an updated function `F`, where for any domain `d`, it will be mapped to a union of a set defined by `F(d)` and a collection of all `C`'s in `R` such that each `C` overlaps `F(d)`. This definition is used in `Clr_Ext` defined next.

Def. $\texttt{Indcd\_Exp}(\texttt{F}\colon (\texttt{D}\colon \mathbb{Set} \to \wp(\texttt{S} : \mathbb{Set}), \texttt{R}\colon \wp(\texttt{S}))\colon \texttt{D} \to \wp(\texttt{S}) =$

$$\lambda\ \texttt{d}\colon \texttt{D}.\left(\texttt{F(d)} \cup \bigcup_{\texttt{C:R}\ni\texttt{C}\cap\texttt{F(d)}\neq\emptyset} \texttt{C}\right);$$

8.6.0.4 Cluster Extension

This definition explicitly describes a mapping function `F` used in the `Indcd_Exp` operator as a mapping from a set `D` to a set of set of trees. The operator `Clr_Ext` returns an updated `F` described using the `Indcd_Exp` operator, where `F` and proper clusters in `F` are used. In this definition, we are looking for proper clusters that overlap with what `F` maps to given the domain set `D`.

Def. $\texttt{Clr\_Ext}(\texttt{F}\colon (\texttt{D}\colon \mathbb{Set}) \to \wp(\wp(\texttt{EStrF\_Tr}(\Gamma : \mathbb{Set})))\colon \texttt{D} \to \wp(\wp(\texttt{EStrF\_Tr}(\Gamma))) =$

$$\texttt{Indcd\_Exp}(\texttt{F}, \texttt{Prpr\_Cluster}(\texttt{F})));$$

8.7 Summary

The mathematical development presented in this chapter should be considered strictly a first presentation of the ideas involved. The focus has been on formalizing the Registry specification, and there is room for improvements to the development. An Appendix B contains the definitions and essential results in the form of theorems and corollaries. Those results underpin the adequacy and consistency of the specifications in the next chapter.

Chapter 9

Registry Specification

This chapter presents a formal specification of the congruence class registry. The entire concept `Congruence_Registry_Temp` is given in Appendix C. This chapter presents the concept in parts along with a detailed description of each part.

It is important to keep in mind that the formal specification (concept) is the view that a client sees, and it reveals nothing about any realization, opening the possibility for an implementer to choose details that will lead to efficiency.

9.1 Concept Parameters

The `Congruence_Registry_Temp` includes several concept parameters shown in Listing 17. The tree node label (`Tr_Nd_Lab`), tree category tag (`Tr_Cgry_Tag`), and tree category attribute (`Tr_Cgry_Attbt`) are types specified by the client during the instantiation of the concept using a facility. For example, if an Integer is used as the tree node label type, variables, constants, and operators represented as strings in the sequent VCs must be converted into integers. The client also provides a default attribute (`Dflt_Attbt`) used in the registration of clusters. The following parameters are three integer upper bounds for the congruence class designator (`C_Cls_D_Cap`), cluster designator (`Clstr_D_Cap`), and an argument list (`Arg_Lst_Cap`). The client is also expected to provide a mapping between the classes and the tags to aid searching in the registry. This mapping is specified by a category tag function (`Cat_Tag_Fn`) supplied as a parameter. All parameters are supplied before the registry is created. Therefore, they should be doable in external terms without

the client knowing how things are represented inside the registry.

```
Concept Congruence_Registry_Temp(
            type Tr_Nd_Lab, Tr_Cgry_Tag, Tr_Cgry_Attbt;
            eval C_Cls_D_Cap, Clstr_D_Cap, Arg_Lst_Cap: Integer;
            eval Dflt_Attbt: Tr_Cgry_Attbt;
            Def Is_Consistent_Tagging(
                Cat_Tag_Fn: ℘fin(℘(EStF_Tr(Tr_Nd_Lab)))→Tr_Cgry_Tag):𝔹);
        uses Relativization_Ext for General_Tree_Theory;
        requires 0 < C_Cls_D_Cap and 0 < Clstr_D_Cap and 0 < Arg_Lst_Cap
                 and 0 < Rt_Lab_Cap which_entails C_Cls_D_Cap,
                           Clstr_D_Cap, Arg_Lst_Cap, Rt_Lab_Cap:ℕ;
    Def. C_Cls_Dsntr: ℘fin(ℕ) =ℕ [0...C_Cls_D_Cap];
    Def. Clstr_Dsntr: ℘fin(ℕ) =ℕ [0...Clstr_D_Cap];
                           ⋮
end Congruence_Registry_Temp;
```

Listing (17) Congruence class registry concept parameters

The mathematics used in the specification of this registry are included in the **uses** clause, and the bounds for the integer values supplied by the client must be non-negative.

As explained in Chapter 7, each congruence class and cluster in the registry is represented by a designator. The two designators are defined in the concept as a finite set of natural numbers ($\mathbb{N}$) in an interval starting from 0 to `C_Cls_D_Cap` for `C_Cls_Dsntr`, and from 0 to `Clstr_D_Cap` for `Clstr_Dsntr`, as shown in Listing 17. The square brackets [...] is a set operator that applies in this case $\mathbb{N}$ to every element in the set within the bracket.

```
Concept Congruence_Registry_Temp(type Tr_Nd_Lab, Tr_Cgry_Tag;...);
                           ⋮
    Type_Family C_Cls_Accessor ⊆ C_Cls_Dsntr;
        exemplar c;
        initialization
            ensures c = 0;
    Oper Replica(restores c: C_Cls_Accessor): C_Cls_Accessor;
        ensures Replica = c;
    Oper Are_Equal(restores c, d: C_Cls_Accessor): Boolean;
```

```
        ensures Are_Equal = (c = d);
    Type_Family Clstr_Accessor ⊆ Clstr_Dsntr;
        exemplar p;
        initialization
            ensures p = 0;
    Oper Clstr_Reset(replaces p: Clstr_Accessor);
        ensures p = 0;
                        :
end Congruence_Registry_Temp;
```

Listing (18) Congruence class registry types

9.2 Congruence Class and Cluster Accessors

The congruence class accessor (`C_Cls_Accessor`) and cluster accessor (`Clstr_Accessor`) are two accessor types exported by the registry. Each accessor is specified in Listing 18 as a type family and mathematically modeled as a subset of designators.

The accessors are specified as natural numbers. However, the client cannot perform normal arithmetic operations on them. Only three operations specified in the concept are allowed to operate on these accessors. This design is part of the information hiding limiting the client's knowledge of how accessors are represented inside. Only the type is exported, and to operate on accessors, only three operations specified are useful. The first operation `Replica` returns a copy of the supplied accessor. The second operation `Are_Equal`, returns **`true`** when two accessors are equal. Finally, operation `Clstr_Reset` restart a cluster accessor to its initial value.

9.3 Congruence Class Registry Model

The central type exported by the concept is the Congruence Class Registry (`CC_Reg`), which is mathematically modeled as a Cartesian product of nine fields shown in Listing (19). The first member of the tuple associates congruence class designator with a class it designates. The next one (`Top_CC_Dsntr`) keeps track of the top designator assigned so far, so that a new designator can be given to the next new class.

```
Concept Congruence_Registry_Temp(type Tr_Nd_Lab, Tr_Cgry_Tag;...);
                          ⋮
    Type_Family CC_Reg ⊆
        Cart_Prod
            CC_Designated: C_Cls_Dsntr⟶℘(EStF_Tr(Tr_Nd_Lab)),
            Top_CC_Dsntr: C_Cls_Dsntr,
            Cr_Designated: Clstr_Dsntr⟶℘(EStF_Tr(Tr_Nd_Lab))
            Top_Cr_Dsntr: Clstr_Dsntr,
            Clstr_Arg_Lst: Str(C_Cls_Dsntr),
            Ctr_Tag: Clstr_Dsntr⟶Tr_Cat_Tag,
            S_Tag: Tr_Nd_Lab×C_Cls_Dsntr⟶Tr_Cat_Tag,
            V_Tag: Tr_Nd_Lab⟶Tr_Cat_Tag,
            CC_Attbt: C_Cls_Dsntr⟶Tr_Cgry_Attbt
        end;
    exemplar Rg;
                          ⋮
end Congruence_Registry_Temp;
```

Listing (19) Congruence class registry mathematical model

Cluster designators are associated with clusters in the next mapping `Cr_Designated` (Cluster Designated). The top cluster designator used so far is tracked in (`Top_Cr_Dsntr`) for the same reason above as for (`Top_CC_Dsntr`).

The cluster argument list (`Clstr_Arg_Lst`) is specified as a string of congruence class designators. The cluster argument list is mostly used in building up clusters. It keeps designators representing the classes for cluster arguments. The formation of clusters is a bottom-up process where once classes for subtrees are known, they are added to the argument list and used in creating a cluster on an upper level. The cluster argument list is also used in a top-down searching process where searching the next level down from a node will use a list of congruence classes it contains.

The cluster tagging function (`Ctr_Tag`) maps a cluster designator to a tree category tag. Similarly, the stand tagging function (`S_Tag`) and variety tag function (`V_Tag`) map their respective designators to tree category tags. The last member is a function `CC_Attbt` (congruence class attribute), which maps the congruence class designators to tree category attributes defined by the

client.

9.4 Abbreviational Definitions

Abbreviational definitions are local to the concept and simplify the specification of constraints and operations in the concept. In this section, each abbreviational definition is described in the order they are presented in Listing (20).

9.4.1 Congruence Class

An operator `C_Class` returns a set of congruence classes in the congruence class registry `Rg`. It is defined in Listing (20) as a finite set of set of trees, equal to a set obtained by applying the congruence class designated function (`CC_Designated`) on each active congruence class designator from `1` to `Top_CC_Dsntr`.

9.4.2 Registry Label

An operator `Rgry_Lab` returns a set of root node labels from a collection of all trees in the registry `Rg`. It is defined in Listing (20) as a finite set of tree node labels, equal to a set obtained by applying a root label function (`Rt_Lab`) on each `C` in a union of all congruence classes in the registry (`Rg`).

9.4.3 Variety Class

An operator `Variety_Cls` returns a set of congruence classes in the registry with trees having a root node label equal to `x`. It is defined in Listing (20) as a set of trees (`EStF_Tr(Tr_Nd_Lab)`), equal to a set of `T`'s such that a root node label returned by a function `Rt_Lab` on `T` equals `x`.

9.4.4 Stand Designator

An operator `Stand_Dsntr` returns a set of stand designators in the congruence class registry `Rg`. It is defined in Listing (20) as a finite set of pairs formed by a cross between a set of root labels in the registry (`Rgry_Lab(Rg)`) and congruence class designators in the active set $_{\mathbb{N}}[1..$`Rg.Top_CC_Dsntr`$]$.

9.4.5 Stand Class

An operator `Stand_Cls` returns a congruence class in the registry with a stand designated by a root node label `x` and a class accessor `c`. It is defined in Listing (20) as a set of trees in the intersection of a variety class containing trees with root node label `x` and those in a class designated by `c`.

9.4.6 Minimal Varietal Congruence Class Designator

An operator `Mnml_VCC_Dsntr` returns a set of minimal congruence class designators for a root node label `x`. It is defined in Listing (20) as a finite set of class designators obtained by a minimal index reflection operator (`Mnml_Idx_Rfln`) given a mapping function `CC_Designated`, congruence classes for each root label `v`, and an ordering function $\leq$. In this case, `Mnml_Idx_Rfln` returns a set of congruence class designators for the provided root label `x`.

9.4.7 Minimal Congruence Class Designator

An operator `Mnml_CC_Dsntr` return a set of all minimal congruence class designators in the registry for every root node label available in the registry. It is defined in Listing (20) as a finite set of class designators obtained by `Mnml_VCC_Dsntr` operator for the registry `Rg`, and every root node label `x` in the set of root labels `Rgry_Lab(Rg)`.

9.4.8 Minimal Stand Cluster Designator

An operator `Mnml_SClstr_Dsntr` returns a set of minimal stand cluster designators for a root node label `x`. It is defined in Listing (20) as a finite set of cluster designators obtained by a minimal index reflection operator given a mapping function `Cr_Designated`, stand classes for each stand designator $\prec$`v`, `a`$\succ$, and an ordering function $\leq$. In this case, `Mnml_Idx_Rfln` returns a set of cluster designators for the provided stand designator $\prec$`x`, `c`$\succ$.

```
Concept Congruence_Registry_Temp(type Tr_Nd_Lab, Tr_Cgry_Tag;...);
                          ⋮
    Abbnl Def. C_Class(Rg: CC_Reg): ℘fin(℘(EstF_Tr(Tr_Nd_Lab))) = (
                              Rg.CC_Designated[ℕ[1..Rg.Top_CC_Dsntr]]);
    Abbnl Def. Rgry_Lab(Rg: CC_Reg): ℘fin(Tr_Nd_Lab) = (
```

```
                                                    Rt_Lab[⋃C:C_Class(Rg) C]);
    Abbnl Def. Variety_Cls(Rg:CC_Reg, x:Rgry_Lab(Rg)):
            ℘(EStF_Tr(Tr_Nd_Lab)) = {T: ⋃C:C_Class(Rg) C | Rt_Lab(T) = x}
    Abbnl Def. Stand_Dsntr(Rg: CC_Reg): ℘fin(Tr_Nd_Lab × C_Cls_Dsntr) = (
                              Rgry_Lab(Rg) × ℕ[1..Rg.Top_CC_Dsntr]);
    Abbnl Def. Stand_Cls(Rg:CC_Reg, ≺x,c≻: Stand_Dsntr(Rg)):
                ℘(EStF_Tr(Tr_Nd_Lab)) =
                              (Variety_Cls(Rg, x) ∩ Rg.CC_Designated(c));
    Abbnl Def. Mnml_VCC_Dsntr(Rg: CC_Reg, x: Rgry_Lab(Rg)):
     ℘fin(C_Cls_Dsntr) = Mnml_Idx_Rfln(
        Rg.CC_Designated, λ v: Rgry_Lab(Rg).Variety_Cls(Rg, v), ≤)(x);
    Abbnl Def. Mnml_CC_Dsntr(Rg: CC_Reg): ℘fin(C_Cls_Dsntr) =
                                    ⋃x:Rgry_Lab(Rg) Mnml_VCC_Dsntr(Rg, x);
    Abbnl Def. Mnml_SClstr_Dsntr(Rg: CC_Reg, x: Rgry_Lab(Rg),
        c: Mnml_VCC_Dsntr(Rg, x)): ℘fin(Clstr_Dsntr) =
          Mnml_Idx_Rfln(Rg.Cr_Designated,
            λ≺v,a≻: Stand_Dsntr(Rg).Stand_Cls(Rg, λ≺v,a≻), ≤)(λ≺x,c≻);
                          ⋮
end Congruence_Registry_Temp;
```

Listing (20) Local abbreviational definitions

9.5 Concept Constraints

Assertions in the **constraints** clause must be valid for every exported congruence class registry. Constraints are specified to set boundaries on the fields used in the CC_Reg model. The first set of these assertions is provided in Listing (21). A complete concept containing all assertions together is presented in Appendix C. The set in Listing (21) constrain the bounds for designators.

```
    Type_Family CC_Reg ⊆
        Cart_Prod
            CC_Designated: C_Cls_Dsntr⟶℘(EStF_Tr(Tr_Nd_Lab)),
            Top_CC_Dsntr: C_Cls_Dsntr,
            Cr_Designated: Clstr_Dsntr⟶℘(EStF_Tr(Tr_Nd_Lab))
```

```
            Top_Cr_Dsntr: Clstr_Dsntr,
            Clstr_Arg_Lst: Str(C_Cls_Dsntr),
                          ⋮
        end;
    exemplar Rg;
                          ⋮
    constraints 0 ≤ Rg.Top_CC_Dsntr and Rg.Top_CC_Dsntr ≤ C_Cls_D_Cap
    and 0 ≤ Rg.Top_Cr_Dsntr and Rg.Top_Cr_Dsntr ≤ Clstr_D_Cap and
    |Rg.Clstr_Arg_Lst| ≤ Arg_Lst_Cap and  (||Rgry_Lab(Rg)|| ≤ Rt_Lab_Cap
    which_entails ||Rgry_Lab(Rg)||: ℕ) and
```

Listing (21) Congruence class registry concept constraints part 1

The second set of constraints is provided in Listing (22). The four assertions in this group are specified to guarantee consistency between functions and operators defined locally in the concept to those defined in theories. For example, the first assertion in this group uses a defined boolean operator `Is_Cplt_Cngr_Prttn` in the theory to guarantee that the congruence class returned by a locally defined operator `C_Class` is a complete congruence partition. The predicate `Is_Cplt_Cngr_Prttn` is described in Section 8.1.4.

Similarly, the second assertion guarantees applying cluster designated function on each active cluster designator in the interval $_{\mathbb{N}}$[1..`Rg.Top_Cr_Dsntr`] forms a set of clusters that must be equal to a set of proper clusters obtained through an operator `Prpr_Cluster`, which is explained in Section 8.2.2.

The **`which_entails`** clause follows additional properties of the operator `Prpr_Cluster` defined in Appendix B to specify what the preceding assertions entail when they are true. The statements state that the collection of congruence clusters is a sub-partition of stand classes, and the collection of stand classes is a sub-partition of congruence classes and variety classes in the registry. The predicate `Is_Subpartition_of` is used in specifying these assertions and is described in Section 8.1.5.

```
Type_Family CC_Reg ⊆
        Cart_Prod
            ...
            Cr_Designated: Clstr_Dsntr⟶℘(EStF_Tr(Tr_Nd_Lab))
```

```
            Top_Cr_Dsntr: Clstr_Dsntr,
            ...
        end;
    exemplar Rg;
                          ⋮
    ( Is_Cplt_Cngr_Prttn(C_Class(Rg)) and
    Rg.Cr_Designated[ℕ[1..Rg.Top_Cr_Dsntr]] = Prpr_Cluster(C_Class(Rg))
    and Stand_Cls[{Rg}, Stand_Dsntr(Rg)] = Stand_for(C_Class(Rg))
    and Variety_Cls[{Rg}, Rgry_Lab(Rg)] = Variety_for(⋃C:C_Class(Rg) C)
    which_entails Rg.Cr_Designated[ℕ[1..Rg.Top_Cr_Dsntr]]
    Is_Subpartition_of Stand_Cls[{Rg}, Stand_Dsntr(Rg)] and
    Stand_Cls[{Rg}, Stand_Dsntr(Rg)] Is_Subpartition_of C_Class(Rg) and
    Stand_Cls[{Rg}, Stand_Dsntr(Rg)] Is_Subpartition_of
        Vrty_Cls_for[⋃C:C_Class(Rg) C, Rgry_Lab(Rg)]) and
```

Listing (22) Congruence class registry concept constraints part 2

The final set of constraints are in Listing (23). The first assertion in this set describes a cluster argument list (`Clstr_Arg_Lst`), a string of minimal congruence class designators. The second assertion guarantees the two registry functions `Cr_Designated`, and `Ctr_Tag` are left factorable functions. Which entails, first, we can create a left factorable function ($\mathbb{LFF}\mathbb{n}$) as a pair of the two functions, `Rg.Cr_Designated` and `Rg.Ctr_Tag`, and second, a cross-map function between clusters and cluster tags returned by `LFctr`$(\prec$`Rg.Cr_Designated`$,$`Rg.Ctr_Tag`$\succ)$ has a consistent tagging. That is, the tagging information for the cluster is consistent with the actual content of the cluster.

Similar to how the cluster designated function and cluster tag function are constrained above, the two sets of constraints that follows achieve the same guarantees for varieties and stands.

```
Type_Family CC_Reg ⊆
        Cart_Prod
            ...
            Cr_Designated: Clstr_Dsntr⟶℘(EStF_Tr(Tr_Nd_Lab))
            ...
            Clstr_Arg_Lst: Str(C_Cls_Dsntr),
            Ctr_Tag: Clstr_Dsntr⟶Tr_Cat_Tag,
```

```
        S_Tag: Tr_Nd_Lab×C_Cls_Dsntr⟶Tr_Cat_Tag,
        V_Tag: Tr_Nd_Lab⟶Tr_Cat_Tag,
        CC_Attbt: C_Cls_Dsntr⟶Tr_Cgry_Attbt
    end;
exemplar Rg;
                        ⋮
Rg.Clstr_Arg_Lst: Str(Mnml_CC_Dsntr(Rg)) and
(Is_L_Fctbl_for(Rg.Cr_Designated, Rg.Ctr_Tag)
which_entails
    ≺Rg.Cr_Designated, Rg.Ctr_Tag≻: 𝕃𝔽𝔽𝔫 and
Is_Consistent_Tagging(LFctr(≺Rg.Cr_Designated, Rg.Ctr_Tag≻)) and
(Is_L_Fctbl_for(λ x: Rgry_Lab(Rg).(Variety_Cls(Rg,x)), Rg.V_Tag)
which_entails
    ≺λ x: Rgry_Lab(Rg).(Variety_Cls(Rg, x)), Rg.V_Tag≻: 𝕃𝔽𝔽𝔫 and
Is_Consistent_Tagging(
        LFctr(≺λ x: Rgry_Lab(Rg).(Variety_Cls(Rg, x)), Rg.V_Tag≻
(Is_L_Fctbl_for(λ N: Stand_Dsntr(Rg).(Stand_Cls(Rg, N)), Rg.S_Tag)
which_entails
    ≺λ N: Stand_Dsntr(Rg).(Stand_Cls(Rg, N)), Rg.S_Tag≻: 𝕃𝔽𝔽𝔫 and
Is_Consistent_Tagging(
        LFctr(≺λ N: Stand_Dsntr(Rg).(Stand_Cls(Rg, N)), Rg.S_Tag≻))
                        ⋮
```

Listing (23) Congruence class registry concept constraints part 3

9.6 Concept Primary Operations

The following operations are primary to the concept and specified to be orthogonal and efficiently realizable.

9.6.1 Register Cluster Operation

A cluster is registered using an operation `Register_Cluster_Lbld` specified in Listing (24). The operation has four parameters, a tree node label (`Lab`), a congruence class accessor (`c`),

tree category attribute (`Tr_Cgry_Attbt`) and a registry (`Rg`).

The operation `Register_Cluster_Lbld` has four preconditions specified in the **`requires`** clause. The first one restricts its use to the registration of a new cluster only, and the next three guarantee a room in the registry for a new cluster and class.

The **`ensures`** clause has several assertions that can be grouped into sets of relating assertions. The first set is for congruence classes, and the first assertion guarantees the `Top_CC_Dsntr` is updated. The next assertion ensures the incoming value in `c` is now replaced with a new `Top_CC_Dsntr`. Because the created cluster would be the first one in the congruence class accessed by `c`, the following assertion guarantees the created congruence class (`CC_Designated(c)`) is equal to the newly created cluster. It is important that the added congruence class not to affect any existing classes in the registry. This restriction is explicitly enforced by the next assertion using a *restricted to* symbol (↾) to specify which part of the registry is updated by the operation. The user also provides an attribute (`Tr_Cgry_Attbt`) for top-level classes identifying them as either coming from the antecedent or succedent. Other registered classes are supplied with a default attribute.

A similar set of assertions above are used to specify changes in the registry for congruence clusters, and once a new cluster is created, the last assertion ensures the congruence cluster argument is left empty (`Rg.Clstr_Arg_Lst` = Λ) for the next cluster.

```
Oper Register_Cluster_Lbld( preserves Lab: Tr_Nd_Lab,
        replaces c: C_Cls_Accessor, alters atb: Tr_Cgry_Attbt, updates
            Rg: CC_Reg );
    requires Cluster_from(
        C_Class(Rg), Rg.CC_Designated[[Rg.Clstr_Arg_Lst]], Lab) ∩
                                                ⋃c:Cclass(Rg) C = ∅ and
        Rg.Top_CC_Dsntr + 1 ≤ C_Cls_D_Cap and
                        Rg.Top_Cr_Dsntr + 1 ≤ C_Clstr_D_Cap and
                                ||Rgry_Lab(Rg)|| + 1 ≤ Rt_Lab_Cap;
    ensures Rg.Top_CC_Dsntr = #Rg.Top_CC_Dsntr + 1 and
        c = Rg.Top_CC_Dsntr and Rg.CC_Designated(c) = Cluster_from(
            C_Class(#Rg), #Rg.CC_Designated[[#Rg.Clstr_Arg_Lst]], Lab) and
        Rg.CC_Designated ↾ (C_Cls_Dsntr ~ {c}) =
                    #Rg.CC_Designated ↾ (C_Cls_Dsntr ~ {c}) and
        Rg.CC_Attbt(c) = atb and
```

```
Rg.CC_Attbt ↾ (C_Cls_Dsntr ~ {c}) =
        #Rg.CC_Attbt ↾ (C_Cls_Dsntr ~ {c})
Rg.Top_Cr_Dsntr = #Rg.Top_Cr_Dsntr + 1 and
Rg.Cr_Designated(Rg.Top_Cr_Dsntr) = Rg.CC_Designated(c) and
Rg.Cr_Designated ↾ ℕ[1..#Rg.Top_Cr_Dsntr] =
            #Rg.Cr_Designated ↾ ℕ[1..#Rg.Top_Cr_Dsntr] and
Rg.Clstr_Arg_Lst = Λ
```

Listing (24) Specifications for Register Cluster Labeled operation

9.6.2 Making Classes Congruent and Checking

Two classes in the registry with trees known to be equal are merged into a single class, and the new class contains trees from both classes. The operation `Make_Congruent` is called for this purpose. This operation also works with `Are_Congruent` operation, which checks to see if two classes in the registry are already congruent.

The operation `Make_Congruent` receives two accessors, `c` and `d`, and updates the registry by merging the two designated classes to a single class if and only if the congruence classes contain trees known to be equal.

The operation is specified in Listing (30) with a requires clause that needs `c` and `d` to be minimal congruence class designators and designate different congruence classes. The operation `Are_Congruent` also specified in Listing (30), is used to determine if the supplied accessors `c` and `d` are congruent, a case where the `Make_Congruent` operation cannot be called.

The **`ensures`** clause in the `Make_Congruent` operation has six post-conditions to describe the behavior of the `Make_Congruent` operation after it is called. The first two conditions are to guarantee that by calling the operation, no changes happen to the registry's top congruence class designator (`Top_CC_Dsntr`), and top cluster designator (`Top_Cr_Dsntr`).

The next two assertions use an infix operator `SubDsgnts` described in Section 8.4 to specify the state of a newly created class in relation to the two individual classes designated by `c` and `d`.

The Minimum Fixed Point (`MFP`) is applied after merging `c` and `d` using the function `Mrg_Val_at`, and followed by successive approximation to the function `CC_Designated` as congruence classes collapse following a merge of classes designated by `c` and `d`. The `Make_Congruent`

operation does not change the cluster argument list, and this is stated in the last assertion.

```
Oper Make_Congruent( restores c, d: C_Cls_Accessor, updates Rg: CC_Reg);
    requires c, d: Mnml_CC_Dsntr(Rg) and
                    Rg.CC_Designated(c) ≠ Rg.CC_Designated(d);
    ensures Rg.Top_CC_Dsntr = #Rg.Top_CC_Dsntr and
            Rg.Top_Cr_Dsntr = #Rg.Top_Cr_Dsntr and
            #Rg.CC_Dsignated SubDsgnts Rg.CC_Designated and
            #Rg.Cr_Dsignated SubDsgnts Rg.Cr_Designated and
            Rg.CC_Designated = MFPwrt(
                    ⊑SF, CCD_Rpr_Fnal(Mrg_Val_at(c, d, #Rg.CC_Designated)))
            and Rg.Clstr_Arg_Lst = #Rg.Clstr_Arg_Lst;
Oper Are_Congruent( restores c, d: C_Cls_Accessor,
                                    restores Rg: CC_Reg): Boolean;
    ensures Are_Congruent = ( c, d: Mnml_CC_Dsntr(Rg) and
                        Rg.CC_Designated(c) = Rg.CC_Designated(d));
```

Listing (25) Specifications for Make Congruent operation

9.6.3 Multi-Level Traversal and Search Operations

Searching the registry for a tree starts at the root, where a class and a matching cluster must be found before progressing to the next level. The operations described in this section achieves movement from one class to another within a variety, and from one cluster to another in a stand.

The first operation `Advance_CC_Accr_for` is used to get the next congruence class accessor in the variety. The current accessor `c` is supplied by the user as a parameter, and after the operation is called, the new accessor for the next congruence class is returned by replacing the value in `c`.

The `Advance_CC_Accr_for` operation is specified in Listing (26). The preconditions in the **`requires`** clause needs the supplied tree node label `x` to be in the set of registry labels, and at least one active congruence class designator after supplied accessor `c` for the operation to be called.

The tree node label `x` can be checked if it is in the set of registry label using an operation `Is_Rgry_Lab`. To check whether the active congruence class designators in the variety list are exhausted, the operation `Is_Vrty_Maximal_for` is called first. This operation returns **`true`** when the supplied congruence class accessor `c` is the last one in the variety list. The operations

`Is_Rgry_Lab` and `Is_Vrty_Maximal_for` are also specified in Listing (26).

Once the operation `Advance_CC_Accr_for` is called, the minimum designator for the class considered next in the variety is assigned to `c`, as stated in the the **ensures** clause. The minimum designator is returned as many class designators will point to the same class when classes are merged in the registry.

```
Oper Advance_CC_Accr_for( restores x: Tr_Nd_Lab,
                updates c: C_Cls_Accessor, restores Rg: CC_Reg );
    requires x: Rgry_Lab(Rg) and Mnml_VCC_Dsntr(Rg, x) ⋂
                    ℕ[c + 1..Rg.Top_CC_Dsntr] ≠ ∅ which_entails
       (Mnml_VCC_Dsntr(Rg, x) ⋂
                        ℕ[c + 1..Rg.Top_CC_Dsntr]): (℘(ℕ) ~ {∅});
    ensures c = GLB(Mnml_VCC_Dsntr(Rg, x) ⋂
                                        ℕ[#c + 1..Rg.Top_CC_Dsntr]);
Oper Is_Vrty_Maximal_for( restores x: Tr_Nd_Lab,
        restores c: C_Cls_Accessor, restores Rg: CC_Reg ): Boolean;
    requires x: Rgry_Lab(Rg);
    ensures Is_Vrty_Maximal_for = (Mnml_VCC_Dsntr(Rg, x) ⋂
                                    ℕ[#c + 1..Rg.Top_CC_Dsntr] = ∅);
Oper Is_Rgry_Lab( restores x: Tr_Nd_Lab,
                            restores Rg: CC_Reg ): Boolean;
     ensures Is_Rgry_Lab = (x: Rgry_Lab(Rg));
Oper Advance_Clstr_Accr_for( restores x: Tr_Nd_Lab,
        restores c: C_Cls_Accessor, updates p: Clstr_Accessor,
                                            restores Rg: CC_Reg );
    requires x: Rgry_Lab(Rg) and c:Mnml_VCC_Dsntr(Rg, x) and
        p: Mnml_SClstr_Dsntr(Rg, x, c) and
        Mnml_SClstr_Dsntr(Rg, x, c) ⋂ ℕ[p + 1..Rg.Top_Cr_Dsntr] ≠ ∅
        which_entails (Mnml_SClstr_Dsntr(Rg, x, c) ⋂
                            ℕ[p + 1..Rg.Top_Cr_Dsntr]): (℘(ℕ) ~ {∅});
    ensures p = GLB(Mnml_SClstr_Dsntr(Rg, x, c) ⋂
                                        ℕ[#p + 1..Rg.Top_Cr_Dsntr]);
Oper Is_Stand_Maximal_for( restores x: Tr_Nd_Lab,
        restores c: C_Cls_Accessor, restores p: Clstr_Accessor,
```

```
                                    restores Rg: CC_Reg ): Boolean;
    ensures Is_Stand_Maximal_for = (x: Rgry_Lab(Rg) and
        c:Mnml_VCC_Dsntr(Rg,x) and Mnml_SClstr_Dsntr(Rg, x, c) ⋂
                                    N[p + 1..Rg.Top_Cr_Dsntr] = ∅);
```

Listing (26) Specifications for advance accessor operations

The second operation `Advance_Clstr_Accr_for` gets the next congruence cluster accessor in a stand after a user supplied accessor `p`. The next cluster accessor is returned by replacing incoming value in `p` at the end of the operation.

The operations `Is_Rgry_Lab` and `Is_Stand_Maximal` are useful in checking the preconditions for this `Advance_Clstr_Accr_for` before it is called. Once the operation is called, the minimum designator for the cluster considered next in the stand is assigned to `p`, as stated in the the **ensures** clause.

9.6.4 Remaining Capacity Operations

The three operations in Listing (27) are possible for getting the remaining capacities in the registry. The first operation `Rmng_CC_Dsntr_Cap` returns a count of unused congruence class designators in the registry. The second operation `Rmng_Clstr_Dsntr_Cap` returns a count on unused congruence cluster designators in the registry. Finally, the operation `Rmng_Lab_Cap` returns the remaining node label capacity in the registry.

```
Oper Rmng_CC_Dsntr_Cap(restores Rg: CC_Reg ): Integer;
    ensures Rmng_CC_Dsntr_Cap = (C_Cls_D_Cap − Rg.Top_CC_Dsntr);
Oper Rmng_Clstr_Dsntr_Cap(restores Rg: CC_Reg ): Integer;
    ensures Rmng_Clstr_Dsntr_Cap = (C_Clstr_D_Cap − Rg.Top_Cr_Dsntr);
Oper Rmng_Lab_Cap(restores Rg: CC_Reg ): Integer;
    ensures Rmng_Lab_Cap = (Rt_Lab_Cap − ||Rgry_Lab(Rg)||);
```

Listing (27) Specifications for remaining capacity operations

9.6.5 Minimal Stand Cluster Designator

Clusters in a particular stand will be merged when classes are merged in the registry, and when that happens, multiple designators end up pointing to the same cluster. The operation `Is_Mnml_SClstr_Dsntr` returns **`true`** when a supplied cluster accessor `p` is a minimum stand cluster designator among those pointing the same cluster. Listing (28) formally specify the operation `Is_Mnml_SClstr_Dsntr` .

```
Oper Is_Mnml_SClstr_Dsntr( restores x: Tr_Nd_Lab,
        restores c: C_Cls_Accessor, restores p: Clstr_Accessor,
                                    restores Rg: CC_Reg ): Boolean;
    ensures Is_Mnml_SClstr_Dsntr = (x: Rgry_Lab(Rg) and
        c:Mnml_VCC_Dsntr(Rg,x) and p:Mnml_SClstr_Dsntr(Rg, x, c));
```

Listing (28) Specifications for Is Minimal Stand Cluster Designator operation

9.6.6 Registry Tag Operations

The client supplies extra information called tags to clusters, stands, and varieties, which help aid in searching the registry. One example of tags a client can provide are fragments discussed in Section 10.3. Once the tags are attached, the client can get them using the three operations specified in Listing (29).

```
Oper Get_Ctr_Tag_for( restores x: Tr_Nd_Lab,
        restores c: C_Cls_Accessor, restores p: Clstr_Accessor,
                    restores Rg: CC_Reg, replaces Tg: Tr_Cgry_Tag);
    requires x: Rgry_Lab(Rg) and c: Mnml_VCC_Dsntr(Rg, x) and
                                p: Mnml_SClstr_Dsntr (Rg, x, c);
    ensures Tg = (Rg.Ctr_Tag(p));
Oper Get_Stand_Tag_for( restores x: Tr_Nd_Lab,
        restores c: C_Cls_Accessor, restores p: Clstr_Accessor,
                        restores Rg: CC_Reg, replaces Tg: Tr_Cgry_Tag);
    requires x: Rgry_Lab(Rg) and c: Mnml_VCC_Dsntr(Rg, x);
    ensures Tg = (Rg.S_Tag(x,c));
Oper Get_Vrty_Tag_for( restores x: Tr_Nd_Lab,
                    restores Rg: CC_Reg, replaces Tg: Tr_Cgry_Tag);
```

```
    requires x: Rgry_Lab(Rg);
    ensures Tg = (Rg.V_Tag(x));
```

Listing (29) Specifications for registry tag operations

9.6.7 Argument List Operations

Registration of a cluster needs the respective arguments in the registry before the register cluster operation is called. The operation `Append_to_Arg_Lst` in Listing (30) is used to append congruence class accessors for arguments to the cluster argument list in the registry. During the registration process, arguments in the registry are removed one by one using an operation `Rmv_First_Arg_Dsntr_to`. The client can keep track of the length of the argument list as arguments are removed using an operation `Arg_Lst_Length`, which tells how many arguments are still in the list. At any point, the client can get the list of all arguments for a cluster by calling the operation `List_Args_from`. This operation is used in the top-down searching of clusters.

```
Oper Append_to_Arg_Lst( restores c: C_Cls_Accessor,
                updates Rg: CC_Reg affecting_only Clstr_Arg_Lst);
    requires |Rg.Clstr_Arg_Lst| + 1 ≤ Arg_Lst_Cap;
    ensures Rg.Clstr_Arg_Lst = #Rg.Clstr_Arg_Lst ∘ ⟨c⟩;
Oper Rmv_First_Arg_Dsntr_to( replaces c: C_Cls_Accessor,
                updates Rg: CC_Reg affecting_only Clstr_Arg_Lst);
    requires 1 ≤ |Rg.Clstr_Arg_Lst|;
    ensures c = ⫷(Prt_btwn(0, 1, #Rg.Clstr_Arg_Lst))
            and Rg.Clstr_Arg_Lst =
                Prt_btwn(1, |#Rg.Clstr_Arg_Lst|, #Rg.Clstr_Arg_Lst);
Oper Arg_Lst_Length( restores Rg: CC_Reg): Integer;
    ensures Arg_Lst_Length = |Rg.Clstr_Arg_Lst|;
Oper List_Args_from( restores x: Tr_Nd_Lab,
        restores c: C_Cls_Accessor, restores p: Clstr_Accessor,
                            updates Rg: CC_Reg, affecting_only
                                Clstr_Arg_Lst);
    requires x: Rgry_Lab(Rg) and c: Mnml_VCC_Dsntr(Rg, x) and
```

```
                                                p: Mnml_SClstr_Dsntr (Rg, x,
                                                    c);
    ensures Rg.CC_Designated[[Rg.Clstr_Arg_Lst]] =
                    Arg_Str_for(C_Class(Rg), Rg.Cr_Designated(p));
```

Listing (30) Specifications for argument list operations

9.7 Summary

The complete specification of the concept presented in this chapter is presented in Appendix C, and its mathematics is presented in Appendix B. The abstract and formal specification makes it possible to create more than one implementation depending on different performance trade-offs and ultimately check the correctness of a prover implementation itself.

Chapter 10

Prover Design and Registry Implementation

This chapter presents a design of Uni-Prover in Section 10.1, and a description of a prototype implementation of the congruence class registry component in Section 10.2. The implementation limits the number of steps taken to prove a sequent VC, making it possible to prove a wider swath of VCs. It has been integrated into the RESOLVE verifying compiler.

10.1 Uni-Prover Design

Figure (10.1) shows an overall design of Uni-Prover with all core components necessary to perform the functions described in Chapter 4. We have harnessed the idea of designing for change where components are created to accommodate upgrades or changes that may happen in the future. This design allows us to incorporate, for example, the grayed-out components intended to further optimize the registry when developed. Nevertheless, the other components are sufficient to provide the functionality without them. All components are designed to be modular, flexible, and reusable.

The congruence class registry in orange is a central component doing most of the heavy lifting for the Uni-Prover. Its specification and implementation form the core of this dissertation, and they are presented in Chapter 9 and Section 10.2, respectively. The following is a discussion of other components that will work with the registry to perform the functions necessary for the

Uni-Prover to prove a sequent VC effectively.

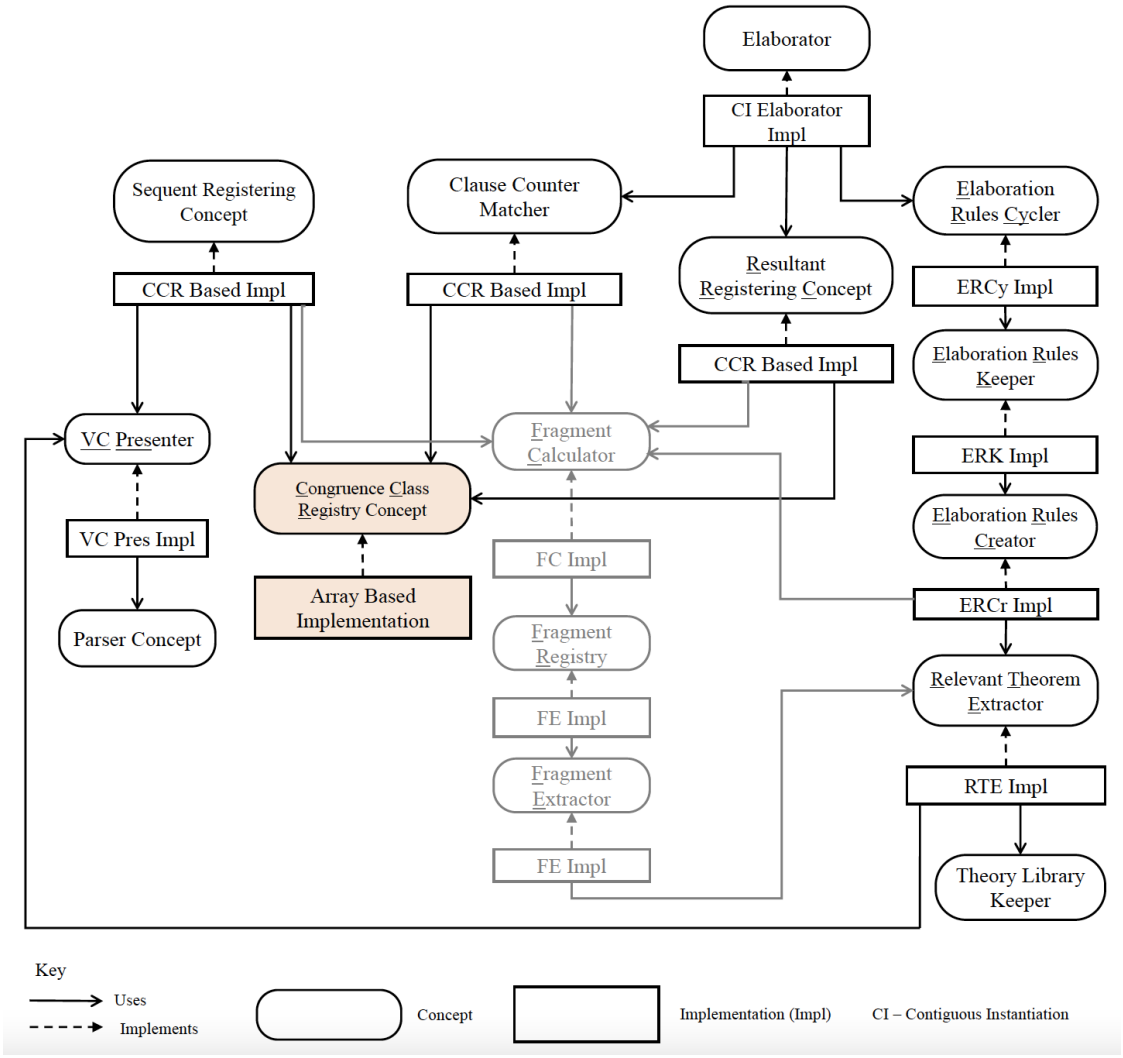


Figure (10.1) Overall Uni-Prover design

On the left is `Sequent Registering Concept (SRC)`, whose implementation uses three other concepts, `VC Presenter (VCP)`, `Congruence Class Registry (CCR)`, and `Fragment Calculator (FC)` to register sequent VCs into the registry. The `VCP` hands over one sequent VC at a time to the `SRC` implementation, which uses operations from the `CCR` to register the sequent VC. The `SRC` implementation attach attributes along with tags supplied by the client and `FC` to classes and clusters it creates in the registry. Attributes are used in verification of the sequent VC, and tags aids searching in the registry. The `FC` works with Fragment Registry, which collects all fragments created from relevant theorems through `Fragment Extractor`. Fragments and their use in the registry are presented in

Section 10.3.1.

Theorems in Uni-Prover are presented through elaboration rules, which are then used to verify sequent VCs. The process starts with the `Relevant Theorem Extractor` (`RTE`) component, which uses the `Theory Library Keeper` and `VC Presenter` to retrieve theorems from the library relevant to the target sequent VC to be proved. `RTE` is intended to keep the instantiation process effective by selecting only those theorems with a potential to prove the target sequent VC. The `Elaboration Rules Creator` uses `RTE` and `Fragment Calculator` to create elaboration rules from relevant theorems and tag them with information representing their fragments in the `Fragment Registry`.

Once elaboration rules are created, the `Elaboration Rules Cycler` component cycles them and presents one rule after another for instantiation. The cycler can be optimized to prioritize some rules over others. Once a rule is selected, its precursor clauses are counter-matched with what we have in the registry. Matching is the function of a `Clause Counter Matcher`, which can use the `Fragment Calculator` to optimize its matching process by using the fragments tagged in the clusters and classes. Once all precursor clauses are counter-matched, the implementation for the `Resultant Registering Concept` registers the resultant clause to the registry. At the same time, it uses the `Fragment Calculator` to add fragment information to the registered clause. The `Elaborator` component coordinates the functions of the three components, the `Clause Counter Matcher`, `Resultant Registering Concept`, and `Elaboration Rule Cycler`.

10.2 A Protoype Implementation of the Registry

We present a prototype implementation of the congruence class registry based on the software development principles discussed in Section 1.1. This prototype is a baseline for future implementations and future optimizations discussed in Section 10.3.

The congruence class registry performs all manipulations necessary to prove atomic sequents effectively. Classes in the registry contain a collection of trees, making a description of the registry and its implementation a challenge. The main limitation facing the Uni-Prover is resource usage, so the design of every structure, process, and strategy used in Uni-Prover needs to be as effective as possible. The following are the design decisions we have made for the array-based prototype implementation in this work.

Figure (10.2) shows four main record data structures we have created for the four levels in the congruence class registry. Each record has fields pointing directly to the necessary information required by the level or indirectly to another record. For this implementation, integers are used as pointers on each field. Integers are easy to manipulate and can be used with array structures for random access in constant time.

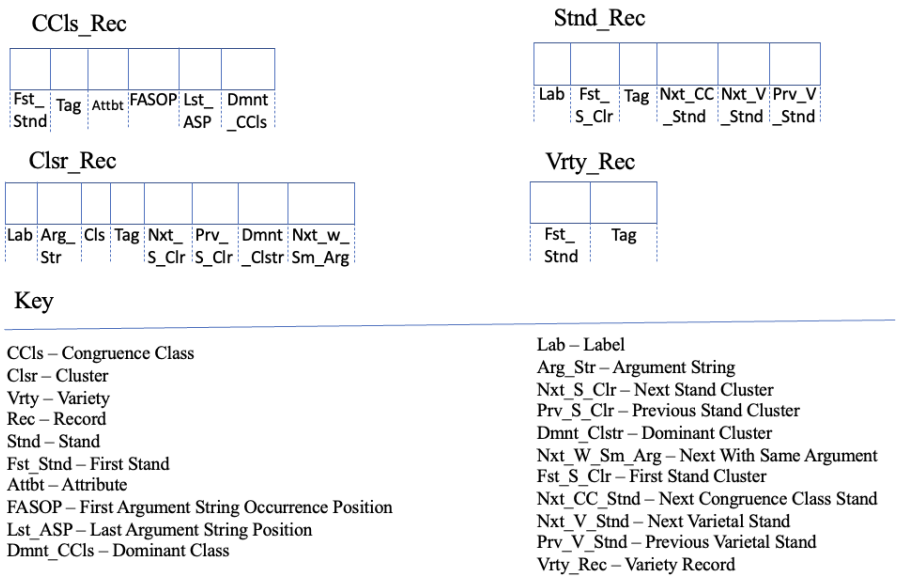


Figure (10.2) Main structures for the congruence class registry levels

A record is created for every constructed congruence class, cluster, stand, and variety. At each level, any new record created is stored in a data structure capable of supporting needed updates and access. The array data structure is selected for this implementation as it provides constant-time access to its elements, which is necessary for some of the computation-intensive operations in the registry. The following is a discussion on each array structure used in our implementation.

10.2.1 Congruence Class Array

Figure (10.3) shows a congruence class array (`CC_Arr`) structure that keeps all created congruence class records (`CCls_Rec`) in the registry. Each class in the registry is assigned a designator (`CC_Dsntr`), which is used as an index in `CC_Arr` for the created `CCls_Rec`. The size `CCC` of the congruence class array is determined by the congruence class designator capacity (`C_Cls_D_Cap`) set by the client during an instantiation of the registry. The following is a discussion on each field in the `CCls_Rec`.

Congruence classes contain stands for each root node label they hold. We keep an ordered list of stands in a class, and the `CCls_Rec` keeps a pointer to the first stand in the field `Fst_Stnd`. Stands are stored in the stand array structure explained in section 10.2.3. A `Tag` field in `CCls_Rec` points to a secondary structure with extra information to make the searching effective. Proposed secondary structures are discussed in section 10.3 as a future direction of this work. Other information currently incorporated in `CCls_Rec` are the attributes (`Attbt`). `Attbt` that identify the side of the sequent VC for the final class record created for the clause in the sequent VC. Two attributes are useful, the antecedent and succedent.

The `FASOP` field in `CCls_Rec` points to a structure `FASOP`, which keeps the first position in the argument string that a congruence class occurred. All arguments are handled by a secondary structure Cluster Argument Array (`Clstr_Arg_Arr`) described in section 10.2.5. For any class' first occurrence in `Clstr_Arg_Arr`, its position is kept in the `FASOP` for effective searching and updating. Additionally, `Lst_ASP` holds a position in `FASOP` for the respective class that last occurred in `Clstr_Arg_Arr`. The final field is essential when two classes are merged, and one class becomes dominant. The `Dmnt_Cls` field for the non-dominant class will point to the dominant class, and the dominant class will point to itself.

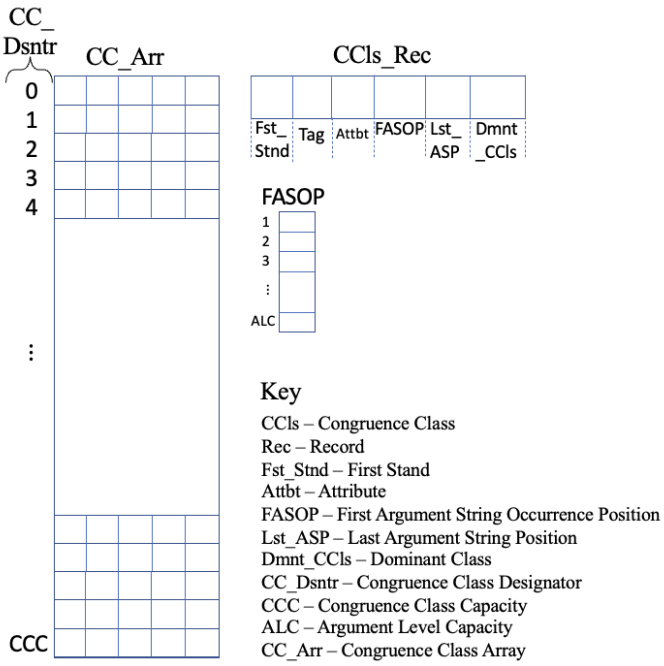


Figure (10.3) Congruence class array structure

10.2.2 Cluster Array

Figure (10.4) shows a cluster array (`Clstr_Arr`) structure that keeps all created cluster records (`Cltr_Rec`) in the registry. Cluster designators (`Clstr_Dsntr`) are used as indices to `Clstr_Arr` for each `Cltr_Rec` created. The size `CLC` of the cluster array is determined by the cluster designator capacity (`Clstr_Dsntr_Cap`) set by the client during an instantiation of the registry. The following is a discussion on each field in a `Cltr_Rec`.

The first two fields represent a label and a string of arguments defining a cluster. The arguments are kept in a secondary structure `Clstr_Arg_Arr` explained in section 10.3. The `Arg_Str` field in `Cltr_Rec` holds a pointer to the position in `Clstr_Arg_Arr` representing the arguments associated with the cluster.

Because every cluster belongs to a class, a `Cls` field points to the `CC_Arr` holding the `CCls_Rec` the cluster belongs. Clusters can also be tagged with extra information to assist with searching. Such information is managed by a different structure and is decoupled from the registry. Only a `Tag` pointer to the extra information is kept in the `Cltr_Rec`.

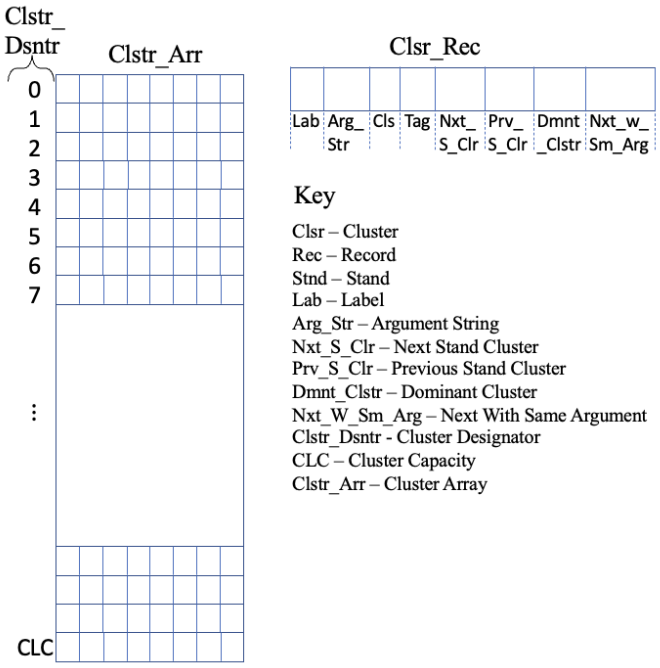


Figure (10.4) Congruence cluster array structure

10.2.3 Stand Array

Stands contain a list of clusters with the same root node label. Figure (10.5) shows a stand Array (`Stnd_Arr`) that keeps all stand records (`Stnd_Rec`) created in the registry. Each `Stnd_Rec` created has a pointer (`Fst_S_Clr`) to the `Clstr_Arr` holding the first stand cluster in the ordered list of clusters the respective stand has. All stands within a congruence class are kept in an ordered list. We can move to the next stand in the list by following the next congruence class stand pointer (`Nxt_CC_Stnd`).

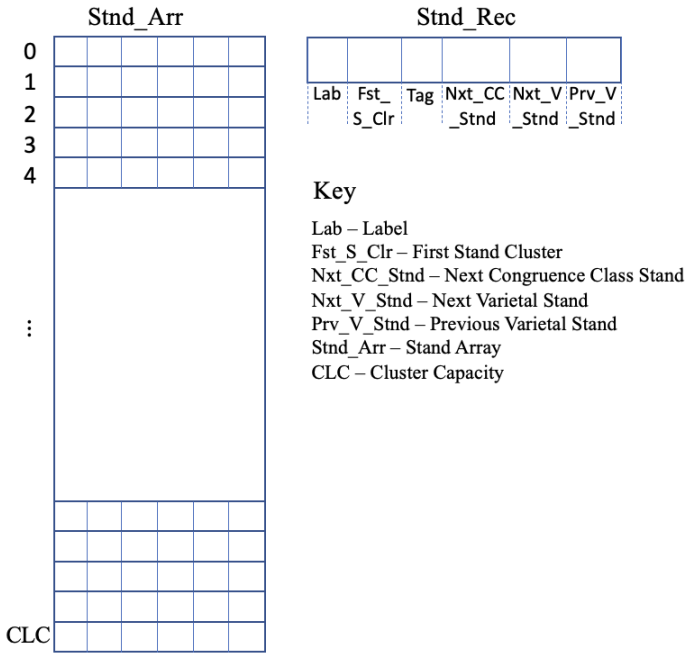


Figure (10.5) Congruence cluster array structure

Stands in different classes that are designated by the same label (`Lab`) are joined together in a doubly-linked list to form a variety. The two pointers `Nxt_V_Stnd` and `Prv_V_Stnd` are used to navigate this variety list.

10.2.4 Variety Array

Variety keeps a list of congruence classes containing at least one tree with the root node label designating the variety. Figure (10.6) shows a variety array (`Vrty_Arr`) that keeps all variety records (`Vrty_Rec`) created in the registry. Each `Vrty_Rec` has a pointer to the first stand in the list of stands belonging to a variety. The size of `Vrty_Arr` (`RLC`) is bounded by the root label capacity

(`Rt_Lab_Cap`) set by the client upon instantiation of the registry.

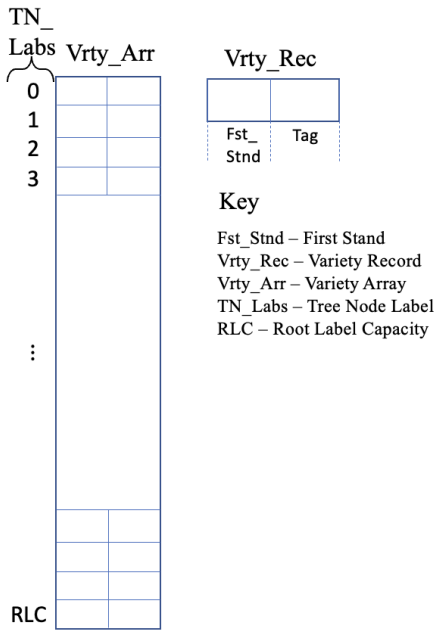


Figure (10.6) Congruence cluster array structure

10.2.5 Cluster Argument Array

Before the `Register_Cluster_Lbld` operation is called, a cluster to be registered must be checked if it already exists in the registry. This check involves looking at the node label and the arguments if they exist in the registry. Because the check is frequent, it causes the operation to be expensive, requiring effective structures that are practical and efficient.

Another expensive operation is `Make_Congruent`. The operation is called to merge two classes, which might be arguments to some clusters, which in turn may cause cascade of clusters and classes to merge following a collapse of two classes. Whenever `Make_Congruent` operation is called, arguments must be updated accordingly to keep the search effective.

A structure handling cluster arguments must effectively support frequent searching and updates for the two operations above to be practical. We have developed a cluster argument array (`Clstr_Arg_Arr`) structure in Figure (10.7) to handle all cluster arguments created. Each argument is a cluster argument record (`Clstr_Arg_Rec`) kept in the `Clstr_Arg_Arr`. The fields in `Clstr_Arg_Rec` are designed to allow effective searching and updates, and their relation can be visualized as a tree shown in Figure (10.8).

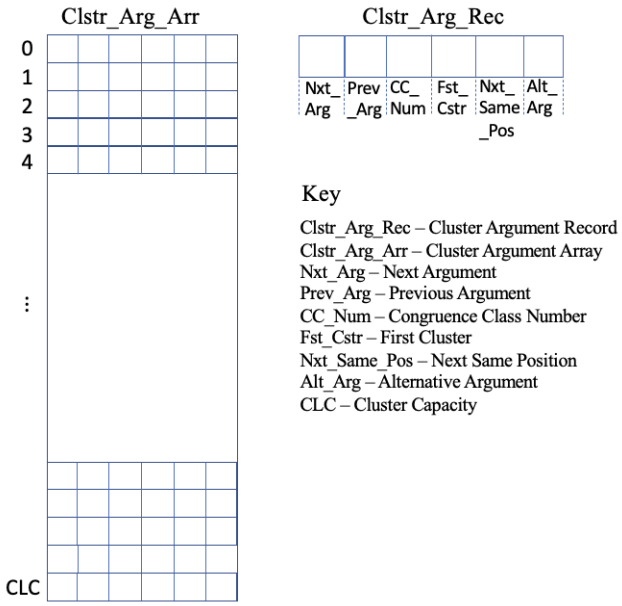


Figure (10.7) Congruence cluster argument array structure

Each node in the tree is a `Clstr_Arg_Rec` and an entry in the `Clstr_Arg_Arr`. The first field `Nxt_Arg` in `Clstr_Arg_Rec` points to the next node in the tree, and the previous node is reached through `Prev_Arg`. Each argument in a cluster is a class, and `CC_Num` points to the class in `CC_Arr` for which `Clstr_Arg_Rec` is created.

The argument string tree is built in levels. The first level is for the first argument, the second level is for the second argument, and progressing similarly for later levels. The number of levels is bounded by `Max_Arg_Lst_Len` set by the client. The field `Fst_Clstr` in `Clstr_Arg_Rec` points to the first cluster in `Clstr_Arr` with the argument created. This field is updated on the last recorded argument for the cluster. For example, a cluster with class `9` and `7` as arguments, the `Fst_Clstr` field is not changed when creating `Clstr_Arg_Rec` for `7` but updated when creating `Clstr_Arg_Rec` for the second argument `9`. An argument string $\langle 9, 7 \rangle$ is shown in see Figure (10.8) as an example.

For a class that appears on different arguments, the positions within a level are connected in a list. Its first position in the argument array is kept in `FASOP` for each level. The `Nxt_Same_Pos` field points to the next occurrence of the same class in `Clstr_Arg_Rec` within a level. The last field, `Alt_Arg`, is an alternative argument and connects all `Clstr_Arg_Rec` in the same level under the same parent.

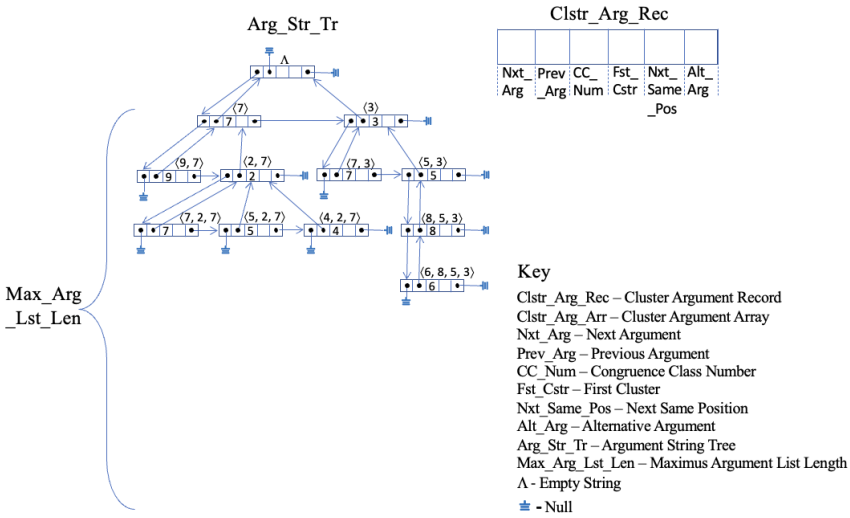


Figure (10.8) Congruence cluster argument string tree structure

10.2.6 Sequent Verification Status Check Using Bit Arrays

An expensive process in the implementation is to check whether a sequent VC is proven. This check must happen in every new cluster registration and every update in the registry. Because of how frequently we have to check if a sequent VC is proven, the operation must be effective and involve fast and practical structures.

A sequent VC is proved when two similar clauses appear on both sides of the sequent VC, a case in the registry where a top-level class will have both attributes. For effectiveness, before the two classes are merged, the attributes must be updated and merged.

We have employed bit arrays, an array data structure that compactly stores bits. Bit arrays are efficient for storing and and they maximally use data cache. They are effective at exploiting bit-level parallelism in hardware to perform constant-time bit-wise operations. In many cases, individual bits cannot be accessed. However, bit-wise operations like OR, AND, XOR, and NOT can be used to determine the position state in a vector.

Here, we can efficiently merge bit arrays representing two classes to one by using an `OR` operation. On every merge, attributes of a newly formed class are checked using a cardinality operation to see if it includes all attributes.

10.3 Future Optimizations

The congruence class registry component developed in this work is flexible and reusable. The flexibility in the design is achieved by decoupling components that are intended to support the registry. This component-based design increases flexibility by allowing necessary changes on these secondary components without changing the core component. We have followed the principle of separation of concerns by limiting information known to the registry about the secondary structures. This way, secondary components can be developed independently.

The rest of this section proposes additional structures and optimizations that can be incorporated into the congruence class registry for effective searching and updates.

10.3.1 Precursor Patterns, Skeleton Patterns, and Skeleton Fragments

Precursor patterns $\mathtt{P_{i,n}}$ contain the same operators, variables, and constants as in precursor trees. Skeleton patterns also have a structure similar to precursor trees, except that all constants and variables replaced with empty trees (Ω). From each precursor clause in the set $\{\mathtt{C_{i,1}},\cdots,\mathtt{C_{i,k}}\} \sim \{\mathtt{C_{i,m}}\}$, a skeleton pattern $\mathtt{S_{i,n}}$ is created. Because the skeleton patterns are only a structure of node labels with empty tree leaves, many of the precursor patterns will likely map to the same skeleton pattern. Each skeleton $\mathtt{S_{i,n}}$ is decomposed into skeletal fragments $\mathtt{F_{i,n,p}}$—one for each subtree of $\mathtt{S_{i,n}}$, and $\mathtt{S_{i,n}}$ itself. The skeleton fragments are proposed to aid in the counter-matching process and improve the searching process.

For example, consider a theorem $\mathtt{UDF_1} : \forall \mathtt{x}, \mathtt{y}, \mathtt{z} : \mathbb{Z},\ \mathtt{if\ x} \leq \mathtt{y\ and\ 0} \leq \mathtt{z\ then\ x} \cdot \mathtt{z} \leq \mathtt{y} \cdot \mathtt{z}$ with three clauses. From $\mathtt{UDF_1}$, three elaboration rules can be created. We have selected one of the rules shown in Figure (10.9) to illustrate the role of skeleton patterns and fragments.

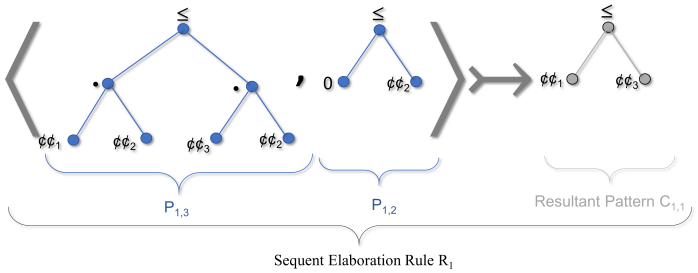


Figure (10.9) Elaboration rule represented by expression trees

Elaboration rule $\mathtt{R_1}$ has two precursor patterns $\mathtt{P_{1,3}}$ and $\mathtt{P_{1,2}}$, and a resultant pattern $\mathtt{C_{1,1}}$.

The two precursor patterns $P_{1,3}$ and $P_{1,2}$, form two skeletons patterns $S_{1,3}$ and $S_{1,2}$, which are further broken down into a set of skeleton fragments F_i as shown in Figure (10.10).

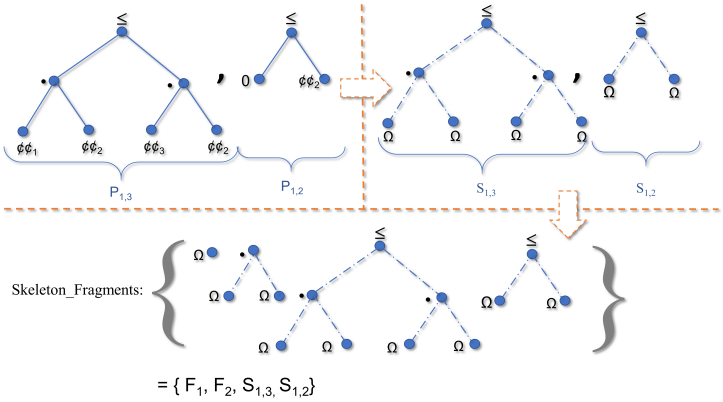


Figure (10.10) Transformation of expression trees to skeleton fragments

10.3.2 Using Skeleton Fragments for Effective Searching

A standard precursor tree matching starts from the root node and searches for a class in the registry with a cluster having a similar root node. A cluster provides classes in its argument to match the root branches of the precursor tree in the next level. As explained in section 4.5, this process is repeated until the variables and constants are replaced with congruence classes. A precursor tree is matched if the variables and constants are translated to congruence classes and match their occurrences in the tree. If the cluster does not match the precursor tree at any point in the search, the search backtracks and other classes and clusters are selected and tried.

Standard precursor tree matching searches the registry blindly. As a result, too much effort may be wasted on a search when a mismatch is discovered. The skeleton fragments and patterns reduce this search cost by considering whether a suitable skeleton fragment exists in the cluster before searching it. If looking for fragments is fast, it will save the time spent following clusters that will end ultimately in a mismatch.

It is possible to perform a calculation that can quickly determine if a skeleton fragment exists in a cluster. The thesis is that the preliminary computational check will be cheaper than an exhaustive search into the clusters only to find it is not a match. Finding a fragment in a cluster does not guarantee a complete precursor clause match that includes variables and constants. However, it eliminates all clusters without target fragments from the search.

10.3.3 Fragments Registry

The plan is to develop a fragment registry to hold all fragments that exist in the congruence class registry. From a tree representing a clause, the registry will store created fragments (while rejecting duplicates) by contiguously enumerating the collection of fragments created. This enumeration is intended to allow fast searching of fragments.

The promised performance gain in skeleton fragment matching is based on the thesis that computation for the fragment's existence could be more efficient than a blind search. To achieve this computational efficiency, we propose an implementation of fragments registry using bit arrays. The expectation is that the bit vector will not be long since the number of theorems that relate to the sequent being proved will be small. The search is expected to take advantage of the fast computation that can be done using bit array operations to check the status of bits in arrays or to merge bit arrays.

10.4 Integration with the RESOLVE Verifying Compiler

We have developed a Java implementation of the array-based congruence class registry following the implementation details described in Section 10.2. The end goal is for the registry to be integrated into the RESOLVE verifying compiler to work in conjunction with the sequent VC generator [61] and mathematical units to establish correctness of VC sequents generated from RESOLVE software.

While much work remains, at this time, the congruence class registry, the core piece of the Uni-Prover, has been successfully integrated within the existing RESOLVE verifying compiler. The reigstry works with the sequent VC generator, and sequent VCs are successfully registered. Because other necessary components to integrate the mathmematical library are not yet built, the registry can only verify a subset of sequent VCs that do not need any theorems for verification. For the Uni-Prover, and hence, the RESOLVE verifying compiler to be fully functional, all other important components described in our design have to be built.

Chapter 11

Summary and Future Work

The work presented here addresses the challenge of proving software in several ways. First, it introduces the idea of "specificationally rich language," necessary for describing object-based software adequately and laying the foundation for the design of a prover capable of working with an extendable mathematical library to verify implementations. Secondly, because of the need for an extensible mathematical library, this work notes that, existing automated verification systems are limited in what they can achieve. Currently, automated verification systems are limited by specialized decision procedures working with a fixed collection of theories. This research has taken a fundamental step in overcoming this limitation by conceiving a Universal Automated Prover for atomic sequents (Uni-Prover), that can work with an extendable mathematical library.

The central contribution is a universal prover that will become a necessary piece of any adequate automated verification system of the future. This dissertation presents an overall design of the Uni-Prover, and a prototype implementation of a formally specified central piece in the prover—the congruence class registry. The prover is designed for change, allowing currently proposed components to function fully with or without future upgrades, changes, or optimizations. The registry is designed to be effective and includes strategies geared to minimize the number of verification steps. One of the strategies developed is the multi-level searching for congruence classes in the registry.

At the core of the prover is the contiguous instantiation strategy, a trigger-less and effective strategy that does not need user-supplied heuristics, to instantiate universal quantified theorems.

The dissertation has developed a formal specification and a prototype implementation of the congruence class registry. The specification is written using new mathematical definitions purposely

developed in this work to make the specifications comprehensible and concise. Its mathematical model is designed with essential functions which are independent and adequate to describe the behavior of the registry concisely. The specification also includes operations to register new classes and clusters, search the registry, and make equal classes congruent.

With a prototype implementation of a congruence class registry, this research has shown that a critical part of the Uni-Prover can be realized. Currently, the prototype is integrated into the RESOLVE verifying compiler and is able to prove sequent VCs that do not require theorems to be verified. Sample sequet VC proofs are shown in Appendix F. The full-functioning Uni-Prover requires all components proposed in the design to be implemented.

The rest of this chapter discusses the remaining work to realize Uni-Prover, additional components for optimizing the core component, and future research directions to extend this work.

11.1 Completing a Fully Functional Uni-Prover

As illustrated in Figure 10.1, the following are the main components to be completed for a working Uni-Prover.

- `Sequent Registering Component`, which registers sequent VCs using the operations in the congruence class registry.
- `Clause Counter Matcher`, which uses the operations in the congruence class registry to match the precursor clauses to the clusters inside the registry.
- `Resultant Registering Component`, which uses the operations in the congruence class registry to register a resultant clause.
- `Relevant Theorem Extractor`, which extracts a subset of theorems relevant to the target sequent VC being proved.
- `Elaboration Rules Creator`, which creates elaboration rules from a set of relevant theorems.
- `Elaboration Rules Keeper`, which keeps all elaboration rules created from the relevant theorems.
- `Elaboration Rules Cycler`, which optimally cycles through the elaboration rules to ensure most promising rules are instantiated first and used to elaborate the sequent VC in the registry.

- `Elaborator`, which coordinates precursor clause counter-matching process, registration of resultant clause, and use of the rule suggested by the Elaboration Rule Cycler compoent.

11.2 Future Optimizations

The following components, noted in Chapter 10, are intended to optimize the core component by providing the supporting structures necessary to make the searching process in the registry much more effective.

- `Fragment Extractor`, which creates fragments out of the relevant theorems selected for the target sequent VC.
- `Fragment Registry`, which stores all fragments created from relevant theorems.
- `Fragment Calculator`, which perform all calculations needed to assign, update fragments numbers, and check the existence of a fragment.

In general, the above optimizations can be done without major changes to the existing system. This is the goal of designing for change.

11.3 Future Verification with Uni-Prover

Uni-Prover is designed to verify all obvious sequent VCs generated from programs. If the theories are well developed initially, relatively infrequently mathematicians will need to be involved in elaborating the mathematical library with more sophisticated theories to make the VCs obvious for verification. For testing this goal, a verification of an implementation of a component using exploration trees presented in Chapter 6 is suggested. For example, a map implementation using exploration trees presented in [43] is an ideal candidate to establish the progress of a Uni-Prover-based verification system.

This work has used the RESOLVE verification system as the framework, with the currently developed congruence class registry integrated into the verifying compiler. However, the design of Uni-Prover can be integrated to other automated verification systems. For example, it can be used as one of the backend prover in Why3 [20].

11.4 Prover Verification

This research has only developed an array-based implementation for the congruence class registry, and future research may consider implementing the registry using other structures. One advantage of having the congruence class component specified is the ability to develop many implementations for different performance trade-offs.

Another research direction is to build on the principles of modular software design, with separate specifications and implementations of components such as the Registry. This design provides an opportunity to verify the correctness of each component and eventually establish the correctness of Uni-Prover itself.

11.5 Opportunities for Education and Training

One main contribution of the RESOLVE verification system is in education. For decades it has been used as a tool in multiple universities to teach foundations in software development and reasoning. The goal is to teach students to write verifiable software that also has other desirable characteristics, such as understandability and maintainability. Using the RESOLVE verifying compiler, students learn to write specifications and implementations, and verify their code for correctness.

A fully developed Uni-Prover integrated into the RESOLVE verifying compiler can provide more opportunities for graduate students and researchers to experiment with and reason about more sophisticated concepts. Currently, the scope and usage of the RESOLVE verification system as well as other systems are limited, in part because of the limitations of the underlying prover. The Uni-Prover is an important step in helping students and researchers explore the full compass of automated formal verification.

Appendices

Appendix A Congruence Class Registry Specification

```
Concept Congruence_Registry_Temp( type Tr_Nd_Lab, Tr_Cgry_Tag, Tr_Cgry_Attbt;
            eval C_Cls_D_Cap, Clstr_D_Cap, Max_Arg_Lst_Len, Rt_Lab_Cap: Integer;
            eval Dflt_Attbt: Tr_Cgry_Attbt;
            def Is_Consistent_Tagging(
                        Cat_Tag_Fn: ℘fin( ℘(EStF_Tr(Tr_Nd_Lab)))→Tr_Cgry_Tag ): 𝔹 );
   (* Tree Node Label, Tree Category Tag, Tree Category Attribute, [Congruence Class,
     Cluster] Designator Capacity, Maximum Argument List Length, Root Label Capacity,
                                                    Category Tag Function *)
      uses Relativization_Ext for General_Tree_Theory;
   requires 0 < C_Cls_D_Cap and 0 < Clstr_D_Cap and 0 < Max_Arg_Lst_Len and
         0 < Rt_Lab_Cap which_entails
                  C_Cls_D_Cap, Clstr_D_Cap, Max_Arg_Lst_Len, Rt_Lab_Cap: ℕ;
   Def. C_Cls_Dsntr: ℘fin(ℕ) = ℕ[0 .. C_Cls_D_Cap];      (* Congruence Class Designator *)
   Def. Clstr_Dsntr: ℘fin(ℕ) = ℕ[0 .. Clstr_D_Cap];      (* Cluster Designator *)

   Type_Family C_Cls_Accessor ⊆ C_Cls_Dsntr;             (* Congruence Class Accessor *)
      exemplar c;
      initialization
         ensures c = 0;

   Oper Replica( restores c: C_Cls_Accessor ): C_Cls_Accessor;
      ensures Replica = c;

   Oper Are_Equal( restores c, d: C_Cls_Accessor ): Boolean;
      ensures Are_Equal = ( c = d );

   Type_Family Clstr_Accessor ⊆ Clstr_Dsntr;             (* Cluster Accessor *)
      exemplar p;
      initialization
         ensures p = 0;

   Oper Clstr_Reset( replaces p: Clstr_Accessor );       (* Cluster Reset *)
      ensures p = 0;
```

Type_Family CC_Reg ⊆ (* Congruence Class Registry *)
Cart_Prod
CC_Designated: C_Cls_Dsntr→ ℘(EStF_Tr(Tr_Nd_Lab)),
(* Congruence Class Designated *)
Top_CC_Dsntr: C_Cls_Dsntr, (* Top Congruence Class Designator *)
Cr_Designated: Clstr_Dsntr→ ℘(EStF_Tr(Tr_Nd_Lab)), (* Cluster Designated *)
Top_Cr_Dsntr: Clstr_Dsntr, (* Top Cluster Designator *)
Clstr_Arg_Lst: Str(C_Cls_Dsntr) (* Cluster Argument List *)
Ctr_Tag: Clstr_Dsntr→Tr_Cat_Tag, (* Cluster Tag *)
S_Tag: Tr_Nd_Lab×C_Cls_Dsntr→Tr_Cat_Tag, (* Stand Tag *)
V_Tag: Tr_Nd_Lab→Tr_Cat_Tag, (* Variety Tag *)
CC_Attbt: C_Cls_Dsntr→Tr_Cgry_Attbt (* Congruence Class Attribute *)
end;
exemplar Rg;

Abbnl Def. C_Class(Rg: CC_Reg): $\wp_{fin}$(℘(EStF_Tr(Tr_Nd_Lab))) = (
(* Congruence Class *) Rg.CC_Designated[ℕ[1 .. Rg.Top_CC_Dsntr]]);

Abbnl Def. Rgry_Lab(Rg: CC_Reg): $\wp_{fin}$(Tr_Nd_Lab) = (Rt_Lab[$\bigcup_{C:\ C\_Class(Rg)}$ C]);
(* Registry Label *)

Abbnl Def. Variety_Cls(Rg: CC_Reg, x: Rgry_Lab(Rg)): ℘(EStF_Tr(Tr_Nd_Lab)) =
(* Variety Class *) { T: $\bigcup_{C:\ C\_Class(Rg)}$ C | Rt_Lab(T) = x };

Abbnl Def. Stand_Dsntr(Rg: CC_Reg): $\wp_{fin}$(Tr_Nd_Lab×C_Cls_Dsntr) = (
(* Stand Designator *) Rgry_Lab(Rg) × ℕ[1..Rg.Top_CC_Dsntr]);

Abbnl Def. Stand_Cls(Rg: CC_Reg, ⟨x, c⟩: Stand_Dsntr(Rg)): ℘(EStF_Tr(Tr_Nd_Lab))
(* Stand Class *) = (Variety_Cls(Rg, x) ⋂ Rg.CC_Designated(c));

Abbnl Def. Mnml_VCC_Dsntr(Rg: CC_Reg, x: Rgry_Lab(Rg)): $\wp_{fin}$(C_Cls_Dsntr) =
Mnml_Idx_Rfln(Rg.CC_Designated, λ v: Rgry_Lab(Rg)**.**Variety_Cls(Rg, v), ≤)(x);
(* Minimal Varietal Congruence Class Designator *)

Abbnl Def. Mnml_CC_Dsntr(Rg: CC_Reg): $\wp_{fin}$(C_Cls_Dsntr) =
$\bigcup_{x:\ Rgry\_Lab(Rg)}$ Mnml_VCC_Dsntr(Rg, x);

Abbnl Def. Mnml_PClstr_Dsntr(Rg: CC_Reg, x: Rgry_Lab(Rg),
c: Mnml_VCC_Dsntr(Rg, x)): $\wp_{fin}$(Clstr_Dsntr) = Mnml_Idx_Rfln(Rg.Cr_Designated,
λ ⟨v, a⟩: Stand_Dsntr(Rg)**.**Stand_Cls(Rg, ⟨v, a⟩), ≤)(⟨x, c⟩);
(* Minimal Cluster Designator *)

constraints 0 ≤ Rg.Top_CC_Dsntr **and** Rg.Top_CC_Dsntr ≤ C_Cls_D_Cap **and**
0 ≤ Rg.Top_Cr_Dsntr **and** Rg.Top_Cr_Dsntr ≤ Clstr_D_Cap **and**
|Rg.Clstr_Arg_Lst| ≤ Max_Arg_Lst_Len **and** (||Rgry_Lab(Rg)|| ≤ Rt_Lab_Cap
which_entails ||Rgry_Lab(Rg)||: ℕ) **and** (Is_Cplt_Cngr_Prttn(C_Class(Rg)) **and**
Rg.Cr_Designated[ℕ[1 .. Rg.Top_Cr_Dsntr]] = Prpr_Cluster(C_Class(Rg)) **and**
Stand_Cls[{Rg}, Stand_Dsntr(Rg)] = Stand_for(C_Class(Rg)) **and**
Variety_Cls[{Rg}, Rgry_Lab(Rg)] = Variety_for($\bigcup_{C:\ C\_Class(Rg)}$ C)
which_entails Rg.Cr_Designated[ℕ[1 .. Rg.Top_Cr_Dsntr]] Is_Subpartition_of
Stand_Cls[{Rg}, Stand_Dsntr(Rg)] **and**
Stand_Cls[{Rg}, Stand_Dsntr(Rg)] Is_Subpartition_of C_Class(Rg) **and**

Stand_Cls[{Rg}, Stand_Dsntr(Rg)] Is_Subpartition_of
Vrty_Cls_for[$\bigcup_{C:\ C\_Class(Rg)}$ C, Rgry_Lab(Rg)]) **and**
Rg.Clstr_Arg_Lst: Str(Mnml_CC_Dsntr(Rg)) **and**
(Is_L_Fctbl_for(Rg.Cr_Designated, Rg.Ctr_Tag) **which_entails**
⟨Rg.Cr_Designated, Rg.Ctr_Tag⟩: 𝕃𝔽𝔽𝕞) **and**
Is_Consistent_Tagging(LFctr(⟨Rg.Cr_Designated, Rg.Ctr_Tag⟩)) **and**
(Is_L_Fctbl_for(λ x: Rgry_Lab(Rg).(Variety_Cls(Rg, x)), Rg.V_Tag)
which_entails ⟨λ x: Rgry_Lab(Rg).(Variety_Cls(Rg, x)), Rg.V_Tag⟩: 𝕃𝔽𝔽𝕞) **and**
Is_Consistent_Tagging(LFctr(⟨ λ x: Rgry_Lab(Rg).(Variety_Cls(Rg, x)),
Rg.V_Tag ⟩)) **and**
(Is_L_Fctbl_for(λ N: Stand_Dsntr(Rg).(Stand_Cls(Rg, N)), Rg.P_Tag)
which_entails ⟨ λ N: Stand_Dsntr(Rg).(Stand_Cls(Rg, N)), Rg.P_Tag ⟩: 𝕃𝔽𝔽𝕞) **and**

Is_Consistent_Tagging(LFctr(⟨ λ N: Stand_Dsntr(Rg).(Stand_Cls(Rg, N)),
Rg.V_Tag ⟩)) **and**

initialization
ensures Rg.Top_CC_Dsntr = 0 **and** Rg.Top_Cr_Dsntr = 0 **and**
Rg.Clstr_Arg_Lst = Λ;

Oper Register_Cluster_Lbld(**preserves** Lab: Tr_Nd_Lab, **replaces** c: C_Cls_Accessor,
(* Register Cluster Labled *) **alters** atb: Tr_Cgry_Attbt, **updates** Rg: CC_Reg);
requires Cluster_from(C_Class(Rg), Rg.CC_Designated[[Rg.Clstr_Arg_Lst]], Lab) ⋂ $\bigcup_{C:\ C\_Class(Rg)} C = \varnothing$ **and**
Rg.Top_CC_Dsntr + 1 ≤ C_Cls_D_Cap **and** Rg.Top_Cr_Dsntr + 1 ≤ C_Clstr_D_Cap
and ||Rgry_Lab(Rg)|| + 1 ≤ Rt_Lab_Cap;
ensures Rg.Top_CC_Dsntr = #Rg.Top_CC_Dsntr + 1 **and** c = Rg.Top_CC_Dsntr **and**
Rg.CC_Designated(c) = Cluster_from(C_Class(#Rg),
#Rg.CC_Designated[[#Rg.Clstr_Arg_Lst]], Lab) **and**
Rg.CC_Designated ⤆ (C_Cls_Dsntr~{c}) = #Rg.CC_Designated ⤆ (C_Cls_Dsntr~{c})
and Rg.CC_Attbt(c) = atb **and**
Rg.CC_Attbt ⤆ (C_Cls_Dsntr~{c}) = #Rg.CC_Attbt ⤆ (C_Cls_Dsntr~{c}) **and**
Rg.Top_Cr_Dsntr = #Rg.Top_Cr_Dsntr + 1 **and**
Rg.Cr_Designated(Rg.Top_Cr_Dsntr) = Rg.CC_Designated(c) **and**
Rg.Cr_Designated ⤆ ℕ[1..#Rg.Top_Cr_Dsntr] =
#Rg.Cr_Designated ⤆ ℕ[1..#Rg.Top_Cr_Dsntr] **and**
Rg.Clstr_Arg_Lst = Λ;

Oper Make_Congruent(**restores** c, d: C_Cls_Accessor, **updates** Rg: CC_Reg);
requires c, d: Mnml_CC_Dsntr(Rg) **and** Rg.CC_Designated(c) ≠ Rg.CC_Designated(d);
ensures Rg.Top_CC_Dsntr = #Rg.Top_CC_Dsntr **and**
Rg.Top_Cr_Dsntr = #Rg.Top_Cr_Dsntr **and**
#Rg.CC_Designated SubDsgnts Rg.CC_Designated **and**
#Rg.Cr_Designated SubDsgnts Rg.Cr_Designated **and**
Rg.CC_Dsntd = MFPwrt($\sqsubseteq_{SF}$, CCD_Rpr_Fnal(Mrg_Val_at(c, d, #Rg.CC_Dsntd)))
and Rg.Clstr_Arg_Lst = #Rg.Clstr_Arg_Lst;

Oper Are_Congruent(**restores** c, d: C_Cls_Accessor, **restores** Rg: CC_Reg): Boolean;
  ensures Are_Congruent = (c, d: Mnml_CC_Dsntr(Rg) **and**
    Rg.CC_Designated(c) = Rg.CC_Designated(d));

Oper Is_Already_Reg_Clstr(**preserves** Lab: Tr_Nd_Lab, **restores** Rg: CC_Reg): Boolean;
  ensures Is_Already_Reg_Clstr = (Cluster_from(C_Class(Rg),
    Rg.CC_Designated[[Rg.Clstr_Arg_Lst]], Lab) ⋂ $\bigcup_{C:\ C\_Class(Rg)}$ C = ∅);

  (* Is Already Registered Cluster *)

Oper Advance_Clstr_Accr_for(**restores** x: Tr_Nd_Lab, **restores** c: C_Cls_Accessor,
  (* Advance Cluster Accessor for *) **updates** p: Clstr_Accessor, **restores** Rg: CC_Reg);
  requires x: Rgry_Lab(Rg) **and** c: Mnml_VCC_Dsntr(Rg, x) **and**
    p: Mnml_PClstr_Dsntr(Rg, x, c) **and**
    Mnml_PClstr_Dsntr(Rg, x, c) ⋂ ℕ[p + 1 .. Rg.Top_Cr_Dsntr] ≠ ∅ **which_entails**
      (Mnml_PClstr_Dsntr(Rg, x, c) ⋂ ℕ[p + 1 .. Rg.Top_Cr_Dsntr]): (℘(ℕ)~{∅});
  ensures p = GLB(Mnml_PClstr_Dsntr(Rg, x, c) ⋂ ℕ[#p + 1 .. Rg.Top_Cr_Dsntr]);

Oper Is_Stand_Maximal_for(**restores** x: Tr_Nd_Lab, **restores** c: C_Cls_Accessor,
    restores p: Clstr_Accessor, **restores** Rg: CC_Reg): Boolean;
  ensures Is_Stand_Maximal_for = (x: Rgry_Lab(Rg) **and** c: Mnml_VCC_Dsntr(Rg, x)
    and Mnml_PClstr_Dsntr(Rg, x, c) ⋂ ℕ[p + 1 .. R.Top_Cr_Dsntr] = ∅);

  (* Is Stand Maximal for *)

Oper Advance_CC_Accr_for(**restores** x: Tr_Nd_Lab, **updates** c: C_Cls_Accessor,
  (* Advance Congruence Classs Accessor for *) **restores** Rg: CC_Reg);
  requires x: Rgry_Lab(Rg) **and** Mnml_VCC_Dsntr(Rg, x) ⋂
    ℕ[c + 1 .. Rg.Top_CC_Dsntr] ≠ ∅ **which_entails**
      (Mnml_VCC_Dsntr(Rg, x) ⋂ ℕ[c + 1 .. Rg.Top_CC_Dsntr]): (℘(ℕ)~{∅});
  ensures c = GLB(Mnml_VCC_Dsntr(Rg, x) ⋂ ℕ[#c + 1 .. Rg.Top_CC_Dsntr]);

Oper Is_Vrty_Maximal_for(**restores** x: Tr_Nd_Lab, **restores** c: C_Cls_Accessor,
(* Is Variety Maximal for *) **restores** Rg: CC_Reg): Boolean;
requires x: Rgry_Lab(Rg);
ensures Is_Vrty_Maximal_for = (Mnml_VCC_Dsntr(Rg, x) ⋂
$\mathbb{N}$[c + 1 .. R.Top_CC_Dsntr] = ∅);

Oper Is_Rgry_Lab(**restores** x: Tr_Nd_Lab, **restores** Rg: CC_Reg): Boolean;
ensures Is_Rgry_Lab = (x: Rgry_Lab(Rg)); (* Is Registry Label *)

Oper Get_Accr_for(**preserves** Lab: Tr_Nd_Lab, **replaces** c: C_Cls_Accessor,
(* Get Accessor for *) **updates** Rg: CC_Reg **affecting_only** Clstr_Arg_Lst);
requires Cluster_from(C_Class(Rg), Rg.CC_Designated[[Rg.Clstr_Arg_Lst]], Lab) ⋂
$\bigcup_{C:\ C\_Class(Rg)} C \neq \emptyset$;
ensures Cluster_from(C_Class(Rg), Rg.CC_Designated[[#Rg.Clstr_Arg_Lst]], Lab) ⊆
Rg.CC_Designated(c) **and** Rg.Clstr_Arg_Lst = Λ;

Oper Rmng_CC_Dsntr_Cap(**restores** Rg: CC_Reg): Integer;
(* Remaining Congruence Class Designator Capacity *)
ensures Rmng_CC_Dsntr_Cap = (C_Cls_D_Cap ∸ Rg.Top_CC_Dsntr);

Oper Rmng_Clstr_Dsntr_Cap(**restores** Rg: CC_Reg): Integer;
(* Remaining Cluster Designator Capacity *)
ensures Rmng_Clstr_Dsntr_Cap = (C_Clstr_D_Cap ∸ Rg.Top_Cr_Dsntr);

Oper Rmng_Lab_Cap(**restores** Rg: CC_Reg): Integer;
(* Remaining Label Capacity *)
ensures Rmng_Lab_Cap = (Rt_Lab_Cap ∸ ||Rgry_Lab(Rg)||);

```
Oper Is_Mnml_VCC_Dsntr( restores x: Tr_Nd_Lab, restores c: C_Cls_Accessor,
                                                    restores Rg: CC_Reg,): Boolean;
    ensures Is_Mnml_VCC_Dsntr = ( x: Rgry_Lab(Rg) and c: Mnml_VCC_Dsntr(Rg, x) );
        (* Is Minimal Varietal Congruence Class Designator *)

Oper Is_Mnml_PClstr_Dsntr( restores x: Tr_Nd_Lab, restores c: C_Cls_Accessor,
                                restores p: Clstr_Accessor, restores Rg: CC_Reg ): Boolean;
    ensures Is_Mnml_PClstr_Dsntr = ( x: Rgry_Lab(Rg) and c: Mnml_VCC_Dsntr(Rg, x)
                                                    and p: Mnml_PClstr_Dsntr(Rg, x, c) );
        (* Is Minimal Stand Cluster Designator *)

Oper Get_Ctr_Tag_for( restores x: Tr_Nd_Lab, restores c: C_Cls_Accessor,
              restores p: Clstr_Accessor, restores Rg: CC_Reg, replaces Tg: Tr_Cgry_Tag );
    requires x: Rgry_Lab(Rg) and c: Mnml_VCC_Dsntr(Rg, x) and
        (* Get Cluster Tag for *)                        p: Mnml_PClstr_Dsntr (Rg, x, c);
    ensures Tg = ( Rg.Ctr_Tag(p) );

Oper List_Args_from( restores x: Tr_Nd_Lab, restores c: C_Cls_Accessor,
            restores p: Clstr_Accessor, updates Rg: CC_Reg affecting_only Clstr_Arg_Lst );
    requires x: Rgry_Lab(Rg) and c: Mnml_VCC_Dsntr(Rg, x) and
        (* List Arguments from *)                        p: Mnml_PClstr_Dsntr(Rg, x, c);

    ensures Rg.CC_Designated[[Rg.Clstr_Arg_Lst]] = Arg_Str_for( C_Class(Rg),
                                                    Rg.Cr_Designated(p) );

Oper Get_Stand_Tag_for( restores x: Tr_Nd_Lab, restores c: C_Cls_Accessor,
      (* Get Stand Tag for *)          restores Rg: CC_Reg, replaces Tg: Tr_Cgry_Tag );
    requires x: Rgry_Lab(Rg) and c: Mnml_VCC_Dsntr(Rg, x);
    ensures Tg = ( Rg.P_Tag(x, c) );
```

```
    Oper Get_Vrty_Tag_for( restores x: Tr_Nd_Lab, restores Rg: CC_Reg,
          (* Get Variety Tag for *)                          replaces Tg: Tr_Cgry_Tag );
        requires x: Rgry_Lab(Rg);
        ensures Tg = ( Rg.V_Tag(x) );

    Oper Rmv_First_Arg_Dsntr_to( replaces c: C_Cls_Accessor, updates Rg: CC_Reg
          (* Remove First Argument Desinator to *)          affecting_only Clstr_Arg_Lst );
        requires 1 ≤ |Rg.Clstr_Arg_Lst|;
        ensures c = ⋦( Prt_btwn(0, 1, #Rg.Clstr_Arg_Lst) ) and Rg.Clstr_Arg_Lst =
                                        Prt_btwn(1, |#Rg.Clstr_Arg_Lst|, #Rg.Clstr_Arg_Lst);

    Oper Arg_Lst_Length( restores Rg: CC_Reg ): Integer;      (* Argument List Length *)
        ensures Arg_Lst_Length = |Rg.Clstr_Arg_Lst|;

    Oper Append_to_Arg_Lst( restores c: C_Cls_Accessor, updates Rg: CC_Reg
          (* Append to Argument List *)                     affecting_only Clstr_Arg_Lst );
        requires |Rg.Clstr_Arg_Lst| + 1 ≤ Max_Arg_Lst_Len;
        ensures Rg.Clstr_Arg_Lst = #Rg.Clstr_Arg_Lst∘⟨c⟩;

end Congruence_Registry_Temp;
```

Appendix B Mathematical Developments for the Registry

This appendix contains the mathematical definitions necessary to specify the congruence class registry.

Def. (K: $\wp(\wp(S: \mathbb{Set}))$) Refines (L: $\wp(\wp(S))$): $\mathbb{B}$ = ($\bigcup_{A: K} A = \bigcup_{C: L} C$ **and** $\forall$ A: K, $\forall$ C: L,

$A \cap C = \varnothing$ **or** $A \subseteq C$);

Corollary 1: $\forall$ S: $\mathbb{Set}$, $\forall$ K, L: $\wp(\wp(S))$, **if** K Refines L, **then** $\forall$ C: L, $\bigcup_{\substack{A: K \ni \\ A \subseteq C}} A = C$;

Corollary 2: $\forall$ S: $\mathbb{Set}$, $\forall$ L: $\wp(\wp(S))$, **if** L Refines L, **then** Is_Partition_of($\bigcup_{C: L} C$,

$L \sim \{\{\varnothing\}\}$);

Corollary 3: $\forall$ S: $\mathbb{Set}$, $\forall$ K, L: $\wp(\wp(S))$, **if** Is_Partition_of($\bigcup_{A: K} A$, K) **and** K Refines L,

then Is_Partition_of($\bigcup_{C: L} C$, $L \sim \{\{\varnothing\}\}$) **iff**

($\forall$ A: K, $\forall$ C, D: L, **if** $A \subseteq C$ **and** $A \subseteq D$, **then** C = D);

Corollary 4: Is_Transitive(Refines) **meaning** $\forall$ S: $\mathbb{Set}$, $\forall$ K, L, M: $\wp(\wp(S))$,

if K Refines L **and** L Refines M, **then** K Refines M;

Corollary 5: $\forall$ S: $\mathbb{Set}$, $\forall$ K, L: $\wp(\wp(S))$, **if** K Refines L **and** L Refines K,

then $K \Delta L \subseteq \{\{\varnothing\}\}$;

Corollary 6: $\forall$ S: $\mathbb{Set}$, $\forall$ K: $\wp(\wp(S))$, K Refines $\{\bigcup_{A: K} A\}$;

Corollary 7: $\forall$ S: $\mathbb{Set}$, $\forall$ L: $\wp(\wp(S))$, $\bigcup_{C: L} \{[C]\}$ Refines L;

Corollary 8: $\forall$ S: $\mathbb{Set}$, $\forall$ L: $\wp(\wp(S))$, $\bigcup_{C: L} \wp(C)$ Refines L;

Def. Is_Partition_of(S: 𝕊𝕖𝕥, P: ℘(℘(S))~{{∅}}): 𝔹 = ($\bigcup_{C:\,P} C = S$ **and**

$\forall$ A, B: P, A ⋂ B = ∅ **or** A = B);

Corollary 1: $\forall$ S: 𝕊𝕖𝕥~{∅}, Is_Partition_of(S, {S});

Corollary 2: $\forall$ S: 𝕊𝕖𝕥, Is_Partition_of(S, ∅) **iff** S = ∅;

Corollary 3: $\forall$ S: 𝕊𝕖𝕥, Is_Partition_of(S, $\bigcup_{x:\,S}$ {{x}});

Corollary 4: $\forall$ S: 𝕊𝕖𝕥, $\forall$ P: ℘(℘(S)), **if** $\forall$ A, B: P, A ⋂ B = ∅ **or** A = B,

then Is_Partition_of($\bigcup_{C:\,P} C$, P~{{∅}});

Corollary 5: $\forall$ S: 𝕊𝕖𝕥, $\forall$ P: ℘(℘(S)), $\forall$ Q: ℘(P), **if** Is_Partition_of(S, P),

then Is_Partition_of($\bigcup_{C:\,Q} C$, Q);

Corollary 6: $\forall$ S: 𝕊𝕖𝕥, $\forall$ P: ℘(℘(S)), $\forall$ R: ℘(℘(P)), **if** Is_Partition_of(S, P) **and**

Is_Partition_of(P, R), **then** Is_Partition_of(S, { D: ℘(S) | ∃ Q: R ∍ D = $\bigcup_{C:\,Q} C$ });

Corollary 7: $\forall$ S: 𝕊𝕖𝕥, $\forall$ P: ℘(℘(S)), $\forall$ A, B: P, **if** Is_Partition_of(S, P),

then Is_Partition_of(S, (P~{A, B}) ⋃ {A ⋃ B});

Corollary 8: $\forall$ S, T: 𝕊𝕖𝕥, $\forall$ P: ℘(℘(S)), $\forall$ Q: ℘(℘(T)), **if** Is_Partition_of(S, P) **and**

Is_Partition_of(T, Q) **and** S⋂T ⊆ $\bigcup_{C:\,P\bigcap Q} C$, **then** Is_Partition_of(S ⋃ T, P ⋃ Q);

Corollary 9: $\forall$ S, R: 𝕊𝕖𝕥, $\forall$ f: S→R, Is_Partition_of(S, { A: ℘(S) | ∃ y: Im(f) ∍

A = {x: S | f(x) = y} });

Corollary 10: $\forall$ S: 𝕊𝕖𝕥, $\forall$ P: ℘(℘(S))~{{∅}}, **if** Is_Partition_of(S, P),

then ∃! C: S→P ∍ $\forall$ x: S, x ∈ C(x);

Def. (K: ℘(℘(S: 𝕊𝕖𝕥))) Is_Subpartition_of (L: ℘(℘(S))): 𝔹 = (Is_Partition_of($\bigcup_{A:\,K} A$, K)

and Is_Partition_of($\bigcup_{A:\,K} A$, L) **and** K Refines L);

Def. Is_Congruence_Partition(P: $\wp_{\text{fin}}(\wp(\text{EStF\_Tr}(\Gamma\colon \mathbb{Set})))$): $\mathbb{B}$ = (Is_Partition_of($\bigcup_{C\colon P} C$, P)

and $\forall\ \alpha$: Str(P), $\forall$ F: Γ, **if** Jn[$\text{Ag}_{(\text{L}^\circ,\ \{\Lambda\})}(\langle[\,[[\alpha]]\,]\rangle)$, {F}] $\cap \bigcup_{C\colon P} C \neq \varnothing$, **then** $\exists$ D: P ∋

Jn[$\text{Ag}_{(\text{L}^\circ,\ \{\Lambda\})}(\langle[\,[[\alpha]]\,]\rangle)$, {F}] $\subseteq$ D);

Corollary 1: Is_Congruence_Partition($\varnothing$);

Corollary 2: $\forall\ \Gamma$: $\mathbb{Set}$, $\forall$ P: $\wp_{\text{fin}}(\wp(\text{EStF\_Tr}(\Gamma\colon \mathbb{Set})))$, $\forall\ \alpha$: Str(P), $\forall$ F: Γ,

if Is_Congruence_Partition(P) **and** Jn[$\text{Ag}_{(\text{L}^\circ,\ \{\Lambda\})}(\langle[\,[[\alpha]]\,]\rangle)$, {F}] $\cap \bigcup_{C\colon P} C = \varnothing$,

then Is_Congruence_Partition(P $\cup$ {{ Jn[$\text{Ag}_{(\text{L}^\circ,\ \{\Lambda\})}(\langle[\,[[\alpha]]\,]\rangle)$, {F}]}});

Def. Is_Cplt_Cngr_Prttn(P: $\wp(\wp(\text{EStF\_Tr}(\Gamma\colon \mathbb{Set})))$): $\mathbb{B}$ = (Is_Congruence_Partition(P)

and $\forall\ \beta, \gamma$: Str(EStF_Tr(Γ)), $\forall$ G: Γ, T: EStF_Tr(Γ), **if** Jn($\beta\circ\langle T\rangle\circ\gamma$, G) $\in \bigcup_{C\colon P} C$,

(∗ Is Complete Congruence Partition ∗) **then** T $\in \bigcup_{C\colon P} C$);

Corollary 1: Is_Cplt_Cngr_Prttn($\varnothing$);

Corollary 2: $\forall\ \Gamma$: $\mathbb{Set}$, $\forall$ P: $\wp_{\text{fin}}(\wp(\text{EStF\_Tr}(\Gamma)))$, $\forall\ \alpha$: Str(P), $\forall$ F: Γ,

if Is_Cplt_Cngr_Prttn(P) **and** Jn[$\text{Ag}_{(\text{L}^\circ,\ \{\Lambda\})}(\langle[\,[[\alpha]]\,]\rangle)$, {F}] $\cap \bigcup_{C\colon P} C = \varnothing$,

then Is_Cplt_Cngr_Prttn(P $\cup$ {{ Jn[$\text{Ag}_{(\text{L}^\circ,\ \{\Lambda\})}(\langle[\,[[\alpha]]\,]\rangle)$, {F}]}});

Corollary 3: $\forall\ \Gamma$: $\mathbb{Set}$, $\forall$ P: $\wp_{\text{fin}}(\wp(\text{EStF\_Tr}(\Gamma)))$, **if** Is_Cplt_Cngr_Prttn(P),

then $\bigcup_{C\colon R}\bigcup_{T\colon C} \text{Occ\_Set}(T) = \text{Rt\_Lab}[\bigcup_{C\colon P} C]$;

Def. Cluster_from(R: $\wp(\wp(\text{EStF\_Tr}(\Gamma\colon \mathbb{Set}))$, α: Str(R), F: Γ): $\wp(\text{EStF\_Tr}(\Gamma))$ = (
Jn[$\text{Ag}_{(\mathbb{L}^\circ,\ \{\Lambda\})}(\langle[\,[[\alpha]]\,]\rangle)$, {F}]);

Corollary 1: $\forall\ \Gamma$: $\mathbb{Set}$, $\forall$ R: $\wp(\wp(\text{EStF\_Tr}(\Gamma)))$, $\forall\ \alpha$: Str(R), $\forall$ F: Γ,
$|[\ \text{Rt\_Brhs}[\text{Cluster\_from}(R, \alpha, F)]\ ]| = \{|\alpha|\}$;

Corollary 2: $\forall\ \Gamma$: $\mathbb{Set}$, $\forall$ R: $\wp(\wp(\text{EStF\_Tr}(\Gamma)))$, $\forall\ \alpha, \beta$: Str(R), $\forall$ F, G: Γ,
if Cluster_from(R, α, F) = Cluster_from(R, β, G), **then** $\alpha = \beta$ **and** F = G;

(* $\forall$ D, R: $\mathbb{Set}$, $\forall$ f: D$\to$R, **if** Is_Injective(f), **then** Is_Injective(λ S: $\wp$(D).(f[S]));
$\forall$ D, R: $\mathbb{Set}$, $\forall$ f: D$\to$R, **if** Is_Injective(f), **then** Is_Injective($\lambda\ \alpha$: Str(D).(f[[α]]));
$\forall$ D, I: $\mathbb{Set}$, $\forall$ b: D, $\forall$ $\divideontimes$: D$\boxtimes$I$^\circ$D, **if** Is_Injective($\divideontimes$),
then Is_Injective($\lambda\ \alpha$: Str(I)**.**($\text{Ag}_{(\divideontimes,\ b)}(\alpha)$));

Def. Prpr_Cluster(R: $\wp(\wp(\text{EStF\_Tr}(\Gamma\colon \mathbb{Set}))$): $\wp(\wp(\text{EStF\_Tr}(\Gamma)))$ = { P: $\wp(\text{EStF\_Tr}(\Gamma))$ |
$\exists\ \alpha$: Str(R), $\exists$ F: $\bigcup\limits_{C:\,R}\bigcup\limits_{T:\,C}$ Occ_Set(T) $\ni$ Cluster_from(R, α, F) = P **and** P $\cap \bigcup\limits_{C:\,R}$ C $\neq \varnothing$ };
(* Proper Cluster *)

Corollary 1: $\forall\ \Gamma$: $\mathbb{Set}$, $\forall$ R: $\wp_{\text{fin}}(\wp(\text{EStF\_Tr}(\Gamma)))$, **if** $\|\bigcup\limits_{C:\,R} C\| < \aleph_0$,
then Prpr_Cluster(R): $\wp_{\text{fin}}(\wp(\text{EStF\_Tr}(\Gamma)))$);

Corollary 2: $\forall\ \Gamma$: $\mathbb{Set}$, $\forall$ R: $\wp_{\text{fin}}(\wp(\text{EStF\_Tr}(\Gamma)))$, **if** Is_Cplt_Cngr_Prttn(R),
then Prpr_Cluster(R) Is_Subpartition_of R;

Corollary 3: $\forall\ \Gamma$: $\mathbb{Set}$, $\forall$ R: $\wp(\wp(\text{EStF\_Tr}(\Gamma)))$, $\forall$ P: Prpr_Cluster(R),
$\exists!\ \alpha$: Str(R), $\exists!$ F: Rt_Lab[$\bigcup\limits_{C:\,R}$ C] $\ni$ Cluster_from(R, α, F) = P;

Def. Stand_for(R: $\wp(\wp(\text{EStF\_Tr}(\Gamma\colon \mathbb{Set})))$): $\wp(\wp(\text{EStF\_Tr}(\Gamma)))$ = (
Vrty_Cls_for[R, Γ]~{{∅}});

Corollary 1: ∀ Γ: 𝕊𝕖𝕥, ∀ R: $\wp_{\text{fin}}(\wp(\text{EStF\_Tr}(\Gamma)))$, **if** $\|\bigcup_{C\colon R} C\| < \aleph_0$,
then Stand_for(R): $\wp_{\text{fin}}(\wp(\text{EStF\_Tr}(\Gamma)))$;

Corollary 2: ∀ Γ: 𝕊𝕖𝕥, ∀ R: $\wp_{\text{fin}}(\wp(\text{EStF\_Tr}(\Gamma)))$, **if** Is_Stand_of($\bigcup_{C\colon R} C$, R),
then Stand_for(R) Is_Subpartition_of R;

Corollary 3: ∀ Γ: 𝕊𝕖𝕥, ∀ R: $\wp_{\text{fin}}(\wp(\text{EStF\_Tr}(\Gamma)))$, **if** Is_Cplt_Cngr_Prttn(R),
then Prpr_Cluster(R) Is_Subpartition_of Stand_for(R);

Corollary 4: ∀ Γ: 𝕊𝕖𝕥, ∀ R: $\wp(\wp(\text{EStF\_Tr}(\Gamma)))$, $\bigcup_{N\colon \text{Plantation\_for}(R)} N = \bigcup_{C\colon R} C$;

Corollary 5: ∀ Γ: 𝕊𝕖𝕥, ∀ R: $\wp(\wp(\text{EStF\_Tr}(\Gamma)))$,

Corollary 5: ∀ Γ: 𝕊𝕖𝕥, ∀ R: $\wp(\wp(\text{EStF\_Tr}(\Gamma)))$, **if** Is_Protopartition_of($\bigcup_{C\colon R} C$, R),
then Is_Protopartition_of($\bigcup_{C\colon R} C$, Stand_for(R));

Corollary 6: ∀ Γ: 𝕊𝕖𝕥, ∀ R: $\wp(\wp(\text{EStF\_Tr}(\Gamma)))$, ∀ N: Stand_for(R),
∃! F: Rt_Lab[$\bigcup_{C\colon R} C$] ∍ Rt_Lab[N] = {F};

Def. Variety_from(D: $\wp(\text{EStF\_Tr}(\Gamma\colon \mathbb{Set}))$, F: Γ): $\wp$(D) = { T: D | Rt_Lab(T) = F };

Corollary 1: ∀ Γ: 𝕊𝕖𝕥, ∀ D: $\wp(\text{EStF\_Tr}(\Gamma\colon \mathbb{Set}))$, ∀ F, G: Rt_Lab[D],
if Variety_from(D, F) = Variety_from(D, G), **then** F = G;

Corollary 2: ∀ Γ: 𝕊𝕖𝕥, ∀ R: $\wp(\wp(\text{EStF\_Tr}(\Gamma)))$, ∀ N: Stand_for(R),
N ⊆ Variety_from($\bigcup_{C\colon R} C$, Cmn_Pln_Lab(R, N));

Corollary 3: ∀ Γ: 𝕊𝕖𝕥, ∀ R: $\wp(\wp(\text{EStF\_Tr}(\Gamma)))$, ∀ F: $\bigcup_{C\colon R} C$, Variety_from($\bigcup_{C\colon R} C$, F) =
$\bigcup_{N\colon \text{Plantation\_for}(R)\ \ni\ \text{Cmn\_Pln\_Lab}(N) = F} N$;

Def. Variety_for(D: $\wp(\text{EStF\_Tr}(\Gamma\colon \mathbb{Set}))$): $\wp$(D) = (Variety_from[{D}, Rt_Lab(D)]);

Corollary 1: ∀ Γ: 𝕊𝕖𝕥, ∀ D: $\wp(\text{EStF\_Tr}(\Gamma))$, Is_Partition_of(D, Variety_for(D));

Corollary 2: ∀ Γ: 𝕊𝕖𝕥, ∀ D: $\wp(\text{EStF\_Tr}(\Gamma))$, **if** ||Rt_Lab[D]|| < $\aleph_0$,
then Variety_for(D): $\wp_{\text{fin}}$(D);

Corollary 3: ∀ Γ: 𝕊𝕖𝕥, ∀ D: $\wp(\text{EStF\_Tr}(\Gamma))$, ∀ V: Variety_for(D), ∃! F: Rt_Lab[D] ∍
Variety_from(D, F) = V;

Def. Idx_Rfln(ER: (I: 𝕊𝕖𝕥)→ ℘(S: 𝕊𝕖𝕥), E: (J: 𝕊𝕖𝕥)→ ℘($\bigcup_{i: I}$ER(i))): J→ ℘(I) =

(∗ Index Reflection ∗) λ j: J•{ i: I | ER(i) ⋂ E(j) ≠ ∅ };

Corollary 1: ∀ I, J, S: 𝕊𝕖𝕥, ∀ ER: I→ ℘(S), ∀ E: J→ ℘($\bigcup_{i: I}$ER(i)), ∀ j, k: J,

if E(j) = E(k), **then** Idx_Rfln(ER, E)(j) = Idx_Rfln(ER, E)(k);

Corollary 2: ∀ I, J, S: 𝕊𝕖𝕥, ∀ ER: I→ ℘(S), ∀ E: J→ ℘($\bigcup_{i: I}$ER(i)), ∀ j: J,

$$E(j) \subseteq \bigcup_{i:\ Idx\_Rfln(ER,\ E)(j)} ER(i);$$

Corollary 3: ∀ I, J, S: 𝕊𝕖𝕥, ∀ ER: I→ ℘(S), ∀ E: J→ ℘($\bigcup_{i: I}$ER(i)),

if Is_Partition_of($\bigcup_{i: I}$ER(i), ER[I]), **then** ∀ j: J,

Is_Partition_of(E(j), [ER[Idx_Rfln(ER, E)(j)]⟮⋂⟯ {E(j)}]);

(∗ ER ⇝ CC_Designated, I ⇝ [1..TopCCD], E ⇝ Variety_Cls, J ⇝ Tr_Nd_Lab ∗)

Corollary 4: ∀ I, J, S: 𝕊𝕖𝕥, ∀ ER: I→ ℘(S), ∀ E: J→ ℘($\bigcup_{i: I}$ER(i)),

$$\textbf{if } ER[I] \text{ Refines } E[J], \textbf{ then } \forall\, j: J,\ \bigcup_{i:\ Idx\_Rfln(ER,\ E)(j)} ER(i) = E(j);$$

Corollary 5: ∀ I, J, S: 𝕊𝕖𝕥, ∀ ER: I→ ℘(S), ∀ E: J→ ℘($\bigcup_{i: I}$ER(i)), ∀ j, k: J,

if ER[I] Refines E[J] **and** Idx_Rfln(ER, E)(j) = Idx_Rfln(ER, E)(k), **then** E(j) = E(k);

Corollary 6: ∀ I, J, S: 𝕊𝕖𝕥, ∀ ER: I→ ℘(S), ∀ E: J→ ℘($\bigcup_{i: I}$ER(i)),

if ER[I] Refines E[J] **and** Is_Partition_of($\bigcup_{i: I}$ER(i), E[J]),

then Is_Partition_of(I, Idx_Rfln(ER, E)[J]);

(∗ ER ⇝ Cr_Designated, I ⇝ [1..TopCrD], E ⇝ Variety_Cls(Rg, x, •), J ⇝ [1..TopCCD] ∗)

Def. Mnml_Idx_Rfln(ER: (I: 𝕊𝕖𝕥)→ ℘(S: 𝕊𝕖𝕥), E: (J: 𝕊𝕖𝕥)→ ℘($\bigcup_{i: I}$ER(i)),

(* Minimal Index Reflection *) °: Wl_Ordng(I)): J→ ℘(I) =

λ j: J. { i: Idx_Rfln(ER, E)(j) | ∀ h: Idx_Rfln(ER, E)(j), **if** ER(i) =ER(h), **then** i ° h };

Corollary 1: ∀ I, J, S: 𝕊𝕖𝕥, ∀ ER: I→ ℘(S), ∀ E: J→ ℘($\bigcup_{i: I}$ER(i)), ∀ °: Wl_Ordng(I),

∀ j, k: J, **if** E(j) = E(k), **then** Mnml_Idx_Rfln(ER, E, °)(j) =

Mnml_Idx_Rfln(ER, E, °) (ER, E)(k);

Corollary 2: ∀ I, J, S: 𝕊𝕖𝕥, ∀ ER: I→ ℘(S), ∀ E: J→ ℘($\bigcup_{i: I}$ER(i)), ∀ °: Wl_Ordng(I),

∀ j: J, ER[Idx_Rfln(ER, E)(j)] = ER[Mnml_Idx_Rfln(ER, E, °)(j)];

Corollary 3: ∀ I, J, S: 𝕊𝕖𝕥, ∀ ER: I→ ℘(S), ∀ E: J→ ℘($\bigcup_{i: I}$ER(i)), ∀ °: Wl_Ordng(I),

∀ j: J, E(j) ⊆ $\bigcup_{i: \text{Mnml\_Idx\_Rfln(ER, E, }\infty\text{)(j)}}$ ER(i);

Corollary 4: ∀ I, J, S: 𝕊𝕖𝕥, ∀ ER: I→ ℘(S), ∀ E: J→ ℘($\bigcup_{i: I}$ER(i)), ∀ °: Wl_Ordng(I),

if Is_Partition_of($\bigcup_{i: I}$ER(i), ER[I]), **then** ∀ j: J,

Is_Partition_of(E(j), [ER[Mnml_Idx_Rfln(ER, E, °)(j)] ⌈⋂⌉ {E(j)}]);

Corollary 5: ∀ I, J, S: 𝕊𝕖𝕥, ∀ ER: I→ ℘(S), ∀ E: J→ ℘($\bigcup_{i: I}$ER(i)), ∀ °: Wl_Ordng(I),

if ER[I] Refines E[J], **then** ∀ j: J, $\bigcup_{i: \text{Mnml\_Idx\_Rfln(ER, E, }\infty\text{)(j)}}$ ER(i) = E(j);

Corollary 6: ∀ I, J, S: 𝕊𝕖𝕥, ∀ ER: I→ ℘(S), ∀ E: J→ ℘($\bigcup_{i: I}$ER(i)), ∀ °: Wl_Ordng(I),

∀ j, k: J, **if** Mnml_Idx_Rfln(ER, E, °)(j) = Mnml_Idx_Rfln(ER, E, °)(k) **and**

ER[I] Refines E[J], **then** E(j) = E(k);

Def. Cls_Cnctd_via(U, R: $\wp(\wp(S\colon \mathbb{Set})\sim\{\varnothing\})$, C: R): $\wp(R)$ = { D: R | ∃ α: Str(R) ϶
∀ E, F: R, **if** ⟨E⟩∘⟨F⟩ Is_Substring ⟨C⟩∘α∘⟨D⟩, **then** ∃ A: U ϶ A⋂E ≠ ∅ **and** A⋂F ≠ ∅ };
(* Class Connected via *) (* U ⇝ C_Class(Rg), R ⇝ Prpr_Cluster(U) *)

Corollary 1: ∀ S: $\mathbb{Set}$, ∀ U, R: $\wp(\wp(S)\sim\{\varnothing\})$, ∀ C: R, Cls_Cnctd_via(U, R, C) = ∅ **iff**
$C \cap \bigcup_{A\colon U} A = \varnothing$;

Corollary 2: ∀ S: $\mathbb{Set}$, ∀ U, R: $\wp(\wp(S)\sim\{\varnothing\})$, ∀ B, C: R,
if B ∈ Cls_Cnctd_via(U, R, C), **then** C ∈ Cls_Cnctd_via(U, R, B);

Corollary 3: ∀ S: $\mathbb{Set}$, ∀ U, R: $\wp(\wp(S)\sim\{\varnothing\})$, ∀ B: R, $B \in \bigcup_{C\colon R}$ Cls_Cnctd_via(U, R, C)
iff Cls_Cnctd_via(U, R, B) ≠ ∅;

Corollary 4: ∀ S: $\mathbb{Set}$, ∀ U, R: $\wp(\wp(S)\sim\{\varnothing\})$, ∀ B, C, D: R,
if B ∈ Cls_Cnctd_via(U, R, C) **and** C ∈ Cls_Cnctd_via(U, R, D),
then B ∈ Cls_Cnctd_via(U, R, D);

Corollary 5: ∀ S: $\mathbb{Set}$, ∀ U, R: $\wp(\wp(S)\sim\{\varnothing\})$,
Is_Partition_of({C: R | Cls_Cnctd_via(U, R, C) ≠ ∅},
Cls_Cnctd_via[{U}, {R}, R]~{∅});

Corollary 6: ∀ S: $\mathbb{Set}$, ∀ U, R: $\wp(\wp(S)\sim\{\varnothing\})$, **if** Is_Partition_of($\bigcup_{C\colon R}$ C, R),
then Is_Partition_of($\bigcup_{C\colon R \ ϶\ \varnothing \neq \text{Cls\_Cnctd\_via}(U, R, C)}$ C, { B: $\wp(\bigcup_{C\colon R} C)\sim\{\varnothing\}$ | ∃ C: R ϶ B = $\bigcup_{D\colon \text{Cls\_Cnctd\_via}(U, R, C)}$ D });

Corollary 7: ∀ S: $\mathbb{Set}$, ∀ U, R: $\wp(\wp(S)\sim\{\varnothing\})$, **if** $\bigcup_{A\colon U} A \subseteq \bigcup_{C\colon R} C$, **then** ∀ A: U, ∃ C: R ϶
$A \subseteq \bigcup_{D\colon \text{Cls\_Cnctd\_via}(U, R, C)} D$;

Def. Is_L_Fctbl_for(G: (D: $\mathbb{Set}$)→(R: $\mathbb{Set}$), H: D→(S: $\mathbb{Set}$)): $\mathbb{B}$ = (∀ c, d: D,
(* Is Left Factorable *) **if** G(c) = G(d), **then** H(c) = H(d));
Corollary 1: ∀ D, R, S: $\mathbb{Set}$, ∀ G: D→R, ∀ H: D→S, **if** Is_L_Fctbl_for(F, G),
then ∀ r: G[D], ∃! s: S, ∃ d: D ϶ G(d) = r **and** H(d) = s;
Corollary 2: ∀ D, R, S: $\mathbb{Set}$, ∀ G: D→R, ∀ H: D→S, **if** Is_L_Fctbl_for(F, G),
then ∃! F: Im(G)→Im(H) ϶ ∀ d: D, F(G(d)) = H(d);

Def. $\mathbb{LFFm}$: $\mathcal{P}(\mathbb{Fm}\times\mathbb{Fm})$ = { ⟨G, H⟩: $\mathbb{Fm}\times\mathbb{Fm}$ | Is_L_Fctbl_for(G, H) };
(* Left Factorable Function *)

Implicit Def. LFctr(⟨G, H⟩: $\mathbb{LFFm}$): Im(G)→Im(H) **is** (* Left Factor *)
∀ d: Dom(G), LFctr(⟨G, H⟩)(G(d)) = H(d);

Appendix C A Data Abstraction for Navigable Trees

```
Concept Exploration_Tree_Template( type Node_Label; eval k, Initial_Capacity: Integer );
        uses Std_Integer_Fac, Std_Boolean_Fac, General_Tree_Theory with
                                                                Relativization_Ext;
        requires 1 ≤ k and 0 < Initial_Capacity which_entails k: ℕ>0 and
                                                                Initial_Capacity: ℕ;
    Var Remaining_Cap: ℕ;
        initialization
            ensures Remaining_Cap = Initial_Capacity;

    Family Tree_Posn ⊆ U_Tr_Pos( k, Node_Label );
        exemplar P;
        initialization
            ensures P.Path = Λ and P.Rem_Tr = Ω;
        finalization
            ensures Remaining_Cap = #Remaining_Cap + N_C (P.Path Ψ P.Rem_Tr);

    Oper Advance( eval dir: Integer; upd P: Tree_Posn );
        requires P.Rem_Tr ≠ Ω and 1 ≤ dir ≤ k;
        ensures P.Rem_Tr = ⫬( Prt_btwn(dir ∸ 1, dir, Rt_Brhs(#P.Rem_Tr)) ) and
        P.Path = #P.Path∘⟨( Rt_Lab(#P.Rem_Tr), Prt_btwn(0, dir ∸ 1, Rt_Brhs(#P.Rem_Tr)),
                                                Prt_btwn(dir, k, Rt_Brhs(#P.Rem_Tr)) )⟩;

    Oper Reset( upd P: Tree_Posn );
        ensures P.Path = Λ and P.Rem_Tr = #P.Path Ψ #P.Rem_Tr;

    Oper At_an_End( rest P: Tree Posn ): Boolean;
        ensures At_an_End = ( P.Rem_Tr = Ω );

    Oper Add_Leaf( alt Labl: Node_Label; upd P: Tree_Posn );
        affects Remaining_Cap;
        requires P.Rem_Tr = Ω and Remaining_Cap > 0;
        ensures P.Path = #P.Path and P.Rem_Tr = Jn( ⟨Ω⟩k, #Labl ) and
                                                Remaining_Cap = #Remaining_Cap ∸1;
```

```
    Oper Remove_Leaf( rpl Leaf_Lab: Node_Label; upd P: Tree_Posn );
        affects Remaining_Cap;
        requires P.Rem_Tr ≠ Ω and Rt_Brhs(P.Rem_Tr) = ⟨Ω⟩^k;
        ensures P.Path = #P.Path and P.Rem_Tr = Ω and Leaf_Lab = Rt_Lab(#P.Rem_Tr)
                                                and Remaining_Cap = #Remaining_Cap + 1;

    Oper At_a_Leaf( rest P: Tree_Posn ): Boolean;
        ensures At_a_Leaf = (Rt_Brhs(#Rem_Tr)=⟨Ω⟩ ^k);

    Oper Swap_Label( upd Labl: Node_Label; upd P: Tree_Posn );
        requires P.Rem_Tr ≠ Ω;
        ensures Labl = Rt_Lab(#P.Rem_Tr) and P.Path = #P.Path and
                                                P.Rem_Tr = Jn( Rt_Brhs(#P.Rem_Tr), #Labl );

    Oper Swap_Rem_Trees( upd P, Q: Tree_Posn );
        ensures P.Path = #P.Path and Q.Path = #Q.Path and P.Rem_Tr = #Q.Rem_Tr and
                                                                Q.Rem_Tr = #P.Rem_Tr;

    Oper Swap_w_Rem( upd P, Q: Tree_Posn );
        ensures P.Path = Λ and P.Rem_Tr = #Q.Rem_Tr and Q.Path = #Q.Path∘#P.Path and
                                                                Q.Rem_Tr = #P.Rem_Tr;

    Oper Retreat( upd P: Tree_Posn );
        requires P.Path ≠ Λ;
        ensures P.Path = Prt_btwn(0, |#P.Path| ∸ 1, #P.Path) and P.Rem_Tr =(
                    Prt_Btwn (|#P.Path| ∸ 1, |#P.Path|, #P.Path) Ψ #P.Rem_Tr;

    Oper Path_Length( rest P: Tree_Posn ): Integer;
        ensures Path_Length = |P.Path|;

    Oper Rmng_Capacity(): Integer;
        ensures Rmng_Capacity = ( Remaining_Cap );
end Exploration_Tree_Template;
```

Appendix D The General Tree Theory

Precis General_Tree_Theory;
uses General_String_Theory **with** Relativization_Ext, Basic_Multiset_Theory;

Definition Is_Tree_Former(Tr: $\mathcal{Cls}$, Ω: Tr, Jn: Str(Tr)×$\mathbb{El}$→(Tr ~ {Ω})): $\mathcal{B}$ = (
Pty 1: ∀ α, β: Str(Tr), ∀ x, y: $\mathbb{El}$, **if** Jn(α, x) = Jn(β, y), **then** α = β **and** x = y;
Pty 2: ∀ C: $\mathcal{P}$(Tr),
if (i) Ω ∈ C **and**
(ii) ∀ α: Str(C), ∀ x: $\mathbb{El}$, Jn(α, x) ∈ C,
then C = Tr);
(* Tree, Empty Tree, Join, Is_Tree_Former: *)
Corollary 1: ∀ Tr: $\mathcal{Cls}$, ∀ Ω: Tr, ∀ Jn: Str(Tr)×$\mathbb{El}$→(Tr ~ {Ω}),
if Is_Tree_Former(Tr, Ω, Jn), **then** ∀ U, V: $\mathcal{Cls}$, ∀ p: U×Str(Tr)×$\mathbb{El}$→U,
∀ b: U→V, ∀ s: U×Str(V)×Str(Tr)×$\mathbb{El}$→V, ∃! f: U×Tr→V ∋ ∀ α: Str(Tr),
∀ u: U, ∀ x: $\mathbb{El}$, f(u, Ω) = b(u) **and** f(u, Jn(α, x)) = s(u, f[p(u, α, x), [α]], α, x);
(* Inductive definability, permutation, basis, successor, function*)
Corollary 2: ∀ Tr_1, Tr_2: $\mathcal{Cls}$, ∀ Ω_1: Tr_1, ∀ Ω_2: Tr_2, ∀ Jn_1: Str(Tr_1)×$\mathbb{El}$→(Tr_1 ~ {Ω_1}),
∀ Jn_2: Str(Tr_2)×$\mathbb{El}$→$_D$(Tr_2~ {Ω_2}), **if** Is_Tree_Former(Tr_1, Ω_1, Jn_1) **and**
Is_Tree_Former(Tr_2, Ω_2, Jn_2), **then** ∃! h: Tr_1→Tr_2 ∋ h(Ω_1) = Ω_2 **and**
∀ α: Str(Tr_1), ∀ x: $\mathbb{El}$, h(Jn_1(α, x)) = Jn_2(h(α), x) **and** Is_Bijective(h);
(* Isomorphism of instances *)
Corollary 3: ∃ Tr: $\mathbb{Cls}$, ∃ Ω: Tr, ∃ Jn: Str(Tr)×$\mathbb{El}$→(Tr ~ {Ω}) ∋
Is_Tree_Former(Tr, Ω, Jn);
(* Satisfiability *)
Categorical Definition for ($\mathbb{Tr}$: $\mathcal{Cls}$, Ω: $\mathbb{Tr}$, Jn: Str($\mathbb{Tr}$)×$\mathbb{El}$→($\mathbb{Tr}$ ~ {Ω})) **is**
Is_Tree_Former($\mathbb{Tr}$, Ω, Jn);
Corollary 1: Is_Surjective(Jn);

Implicit Definition Indcd_Fn(U, V: $\mathcal{Cls}$, b: U→V, s: U×Str(V)×Str(𝕋𝕣)×𝔼𝕝→V,
p: U×Str(𝕋𝕣)×𝔼𝕝→U): U×𝕋𝕣→V **is**
∀ u: U, Indcd_Fn(U, V, b, s, p) (u, Ω) = b(u) **and** ∀ α: Str(𝕋𝕣), ∀ x: 𝔼𝕝,
Indcd_Fn(U, V, b, s, p)(u, Jn(α, x)) = s(u, Indcd_Fn(U, V, b, s, p)[p(u, α, x), [α]], α, x);

Inductive Def. on T: 𝕋𝕣 **of** N_C(T): ℕ **is** (* Node Count *)
(i) N_C(Ω) = 0;
(ii) N_C (Jn(α, x)) = suc($Ag_{(+, 0)}$(N_C[[α]]));
Corollary 1: ∀ T: 𝕋𝕣, N_C(T) = 0 **iff** T = Ω;

Inductive Def. on T: 𝕋𝕣 **of** ht(T): ℕ **is** (* height *)
(i) ht(Ω) = 0;
(ii) ht(Jn(α, x)) = suc($Ag_{(Max, 0)}$(ht[[α]]));
Corollary 1: ∀ T: 𝕋𝕣, ht(T) = 0 **iff** T = Ω;
Corollary 2: ∀ T: 𝕋𝕣, ht(T) ≤ N_C(T);

Inductive Def. on T: 𝕋𝕣 **of** L_C(T): ℕ **is** (* Leaf Count *)
(i) L_C(Ω) = 0;
(ii) L_C(Jn(α, x)) = $\begin{cases} 1 & \textbf{if } \text{Is\_Leaf}(Jn(\alpha, x)) \\ Ag_{(+, 0)}(L\_C[[\alpha]]) & \textbf{otherwise} \end{cases}$;
Corollary 1: ∀ T: 𝕋𝕣, L_C(T) = 0 **iff** T = Ω;
Corollary 2: ∀ T: 𝕋𝕣, L_C(T) ≤ N_C(T);

Inductive Def. on T: 𝕋𝕣 **of** Occ_Set(T: 𝕋𝕣): 𝕊𝕖𝕥 **is** (* Occurrence Set *)
(i) Occ_Set(Ω) = ∅;
(ii) Occ_Set(Jn(α, x)) = $Ag_{(\cup, \varnothing)}$(Occ_Set[[α]]) ∪ {x};
Corollary 1: ∀ T: 𝕋𝕣, ||Occ_Set(T)||: ℕ;
Corollary 2: ∀ T: 𝕋𝕣, ||Occ_Set(T)|| ≤ N_C(T);

Inductive Def. **on** T: $\mathbb{Tr}$ **of** Occ_Tly(T): $\mathbb{MSet}$ **is** (* Occurrence Tally *)

(i) $Occ\_Tly(\Omega) = \Phi$;

(ii) $Occ\_Tly(Jn(\alpha, x)) = Ag_{(\uplus,\Phi)}(Occ\_Tly[[\alpha]]) \uplus \backslash x /$;

Corollary 1: $Occ\_Tly: \mathbb{Tr} \rightarrow \mathbb{MSet}_{fin}$;

Corollary 2: $\forall$ T: $\mathbb{Tr}$, Occ_Set(T) = Occ_Tly(T);

Corollary 3: $\forall$ T: $\mathbb{Tr}$, N_C(T) = ||Occ_Tly(T)||;

Inductive Def. **on** T: $\mathbb{Tr}$ **of** $(T)^{TRev}$: Tr(Γ) **is** (* Tree Reversal *)

(i) $\Omega^{TRev} = \Omega$;

(ii) $Jn(\alpha, x)^{TRev} = Jn(([[\alpha]]^{TRev})^{Rev}, x)$;

Corollary 1: $\forall$ T: $\mathbb{Tr}$, $(T^{TRev})^{TRev} = T$;

Corollary 2: $\forall$ T: $\mathbb{Tr}$, $N\_C(T^{TRev}) = N\_C(T)$;

Corollary 3: $\forall$ T: $\mathbb{Tr}$, $ht(T^{TRev}) = ht(T)$;

Corollary 4: $\forall$ T: $\mathbb{Tr}$, $L\_C(T^{TRev}) = L\_C(T)$;

Corollary 5: $\forall$ T: $\mathbb{Tr}$, $Occ\_Tly(T^{TRev}) = Occ\_Tly(T)$;

Implicit Defs. Rt_Lab(T: $\mathbb{Tr}\sim\{\Omega\}$): $\mathbb{El}$ **and**

Rt_Brhs(T: $\mathbb{Tr}\sim\{\Omega\}$): Str($\mathbb{Tr}$) **is** (* Root Label and Branches *)

Jn(Rt_Brhs(T), Rt_Lab(T)) = T;

Corollary 1: $\forall$ x: $\mathbb{El}$, $\forall$ α: Str($\mathbb{Tr}$), Rt_Lab(Jn(α, x)) = x **and** Rt_Brhs(Jn(α, x)) = α;

Def. $\mathbb{Site}$ = **Cart_Prod**

Lab: $\mathbb{El}$;

LTS, RTS: Str($\mathbb{Tr}$) (* Left Tree String, Right Tree String *)

end;

Implicit Def. $(S: \mathbb{Site})^{SRev}$: $\mathbb{Site}$ **is** (* Site Reversal *)

S^{SRev}.Lab = S.Lab **and** S^{SRev}.LTS = $([[S.RTS]]^{TRev})^{Rev}$ **and**

S^{SRev}.RTS = $([[S.LTS]]^{TRev})^{Rev}$;

Corollary 1: $\forall$ S: $\mathbb{Site}$, $(S^{SRev})^{SRev} = S$;

Def. 𝕋r_ℙos = **Cart_Prod** (* Tree Position *)
Path: Str(𝕊ite);
Rem_Tr: 𝕋r (* Remainder Tree *)
end;

Implicit Def. (P: 𝕋r_ℙos)^{PRev}: 𝕋r_ℙos **is** (* Position Reversal *)
P^{PRev}.Path = [[P.Path]]^{SRev} **and** P^{PRev}.Rem_TR = P.Rem_TR^{TRev};
Corollary 1: ∀ P: 𝕋r_ℙos, $(P^{PRev})^{PRev}$ = P;

Inductive Def. **on** ρ: Str(𝕊ite) **of** (ρ)Ψ(T: 𝕋r): 𝕋r **is** (* zip operator *)
(i) Λ Ψ T = T;
(ii) ext(ρ, S) Ψ T = ρ Ψ Jn(S.LTS∘⟨T⟩∘S.RTS, S.Lab);
Corollary 1: ∀ ρ, σ: Str(𝕊ite), ∀ T: 𝕋r, (ρ∘σ)ΨT = ρΨ(σΨT);
Corollary 2: ∀ P: 𝕋r_ℙos, (P.Path Ψ P.Rem_Tr)^{TRev} = P^{PRev}.Path Ψ P^{PRev}.Rem_Tr;
Corollary 3: ∀ P: 𝕋r_ℙos, |P.Path| + ht(P.Rem_Tr) ≤ ht(P.Path Ψ P.Rem_Tr);
Corollary 4: ∀ R, S: 𝕊ite, ∀ T, U: 𝕋r, **if** ⟨R⟩ Ψ T = ⟨S⟩ Ψ U **and**
(|R.LTS| = |S.LTS| **or** |R.RTS| = |S.RTS|), **then** R = S **and** T = U;

Def. (T: 𝕋r) Is_Subtree (U: 𝕋r): 𝔹 = (∃ ρ: Str(𝕊ite) ∍ ρΨT = U);
Corollary 1: Is_Partial_Ordering(Is_Subtree);
Corollary 2: ∀ T, U: 𝕋r, **if** T Is_Subtree U, **then** N_C(T) ≤ N_C(U);
Corollary 3: ∀ T, U: 𝕋r, **if** T Is_Subtree U, **then** ht(T) ≤ ht(U);
Corollary 4: ∀ T, U: 𝕋r, **if** T Is_Subtree U, **then** L_C(T) ≤ L_C(U);
Corollary 5: ∀ T, U: 𝕋r, **if** T Is_Subtree U, **then** Occ_Set(T) ⊆ Occ_Set(U);
Corollary 6: ∀ T, U: 𝕋r, **if** T Is_Subtree U, **then** Occ_Tly(T) ⊆ Occ_Tly(U);
Corollary 7: ∀ T, U: 𝕋r, **if** T Is_Subtree U, **then** T^{TRev} Is_Subtree U^{TRev};

Implicit Def. Split_at(i: ℕ, T: 𝕋r~{Ω}): **Cart_Prod** St: 𝕊ite, RT: 𝕋r **end is**
(* produces a Site and a Remainder Tree *)
⟨Split_at(i, T).St⟩ Ψ Split_at(i, T).RT = T **and**
|Split_at(i, T).St.LTS| = min(i, |Split_at(i, T).St.LTS| + |Split_at(i, T).St.RTS|);
Corollary 1: ∀ S: 𝕊ite, ∀ T: 𝕋r, Split_at(|S.LTS|, ⟨S⟩ Ψ T).St = S **and** ∀ i: ℕ,
Split_at(i, ⟨S⟩ Ψ T).RT = T;
Corollary 2: ∀ i: ℕ, ∀ T: 𝕋r~{Ω}, Split_at(i, T).RT Is_Subtree T;
Corollary 3: ∀ i: ℕ, ∀ T: 𝕋r~{Ω}, ht(Split_at(i, T).RT) < ht(T);

Def. Uf_Tr(k: ℕ>0): 𝒫(𝒯r) = { T: 𝕋r | ∀ P: 𝕋r_ℙos, **if** P.Path Ψ P.Rem_Tr = T,
then (P.Rem_Tr = Ω **or** |Rt_Brhs(P.Rem_Tr)| = k) };
Uf_Tr (* Uniform k-way Tree *)
Corollary 1: ∀ k: ℕ>0, ∀ C: 𝒞ls,
if (i) C ⊆ Uf_Tr(k) **and** Ω ∈ C **and**
(ii) ∀ x: 𝔼l, ∀ α: Str(C), **if** |α| = k, **then** Jn(α, x) ∈ C,
then C = Uf_Tr(k);
Corollary 2: ∀ k: ℕ>0, ∀ T: Uf_Tr(k), L_C(T)·k ≤ $k^{ht(T)}$;
Corollary 3: ∀ k: ℕ>0, ∀ T: Uf_Tr(k), P_C(T) = k·N_C(T) + 1;
Corollary 4: ∀ k: ℕ>0, ∀ T: Uf_Tr(k), ht(T) ≤ N_C(T);
Corollary 5: ∀ k: ℕ>0, ∀ T: Uf_Tr(k), N_C(T)·(k ∸ 1) ≤ ($k^{ht(T)}$ ∸ 1);
Corollary 6: ∀ k: ℕ>0, ∀ T: Uf_Tr(k), T^{TRev}: Uf_Tr(k);

Def. Uf_Site(k: ℕ>0): 𝒫(𝕊ite) = { S: 𝕊ite | ⟨S⟩ Ψ Ω ∈ Uf_Tr(k) }; (* Uniform Site *)
Corollary 1: ∀ k: ℕ>0, ∀ S: Uf_Site(k), |S.LTS| + |S.RTS| + 1 = k;

Def. Uf_Tr_Pos(k: ℕ>0): 𝒫(𝕋r_ℙos) = { P: 𝕋r_ℙos | P.Path Ψ P.Rem_Tr ∈ Uf_Tr(k) };
(* Uniform Tree Position *)
Corollary 1: ∀ k: ℕ>0, Ψ: Str(Uf_Site(k))⊠Uf_Tr(k)→Uf_Tr(k);
end General_Tree_Theory;

Appendix E Specification of a Nested List Concept

Concept Nested_List_Template(**type** Label; **eval** Max_Total_Size: Integer);
uses Std_Integer_Fac, Std_Boolean_Fac, Tree_Extractor_Ext **for**
Function_Theory, General_Tree_Theory **with** Relativization_Ext;
requires Max_Total_Size > 0 **which_entails** Max_Total_Size: ℕ;

Family Nstd_Lst_Pos ⊆ **Cart_Prod** (* Nested List Position *)
Path: Str(NL_Site(Label));
Prec, Rem: Str(EStF_Tr(Label)); (* Preceding, Remainder *)
end;

exemplar P;
initialization
ensures P.Path = Λ **and** P.Prec = Λ **and** P.Rem = Λ;
Def const Suture_of(P: Nstd_Lst_Pos): Str(Tr(Label)) = (
P.Path Ξ (P.Prec, P.Rem).VS ∘ P.Path Ξ (P.Prec, P.Rem).US);
Def const List_Node_Ct(P: Nstd_Lst_Pos): ℕ = ($Ag_{(+,\,0)}$(N_C[[Suture_of(P)]]));
Def var Aggregate_Size: ℕ = ($\sum_{r:\,Nstd\_Lst\_Pos.\mathbf{Receptacle}}$ List_Node_Ct(Nstd_Lst_Pos.**Val_in**(r)));
Def var Remaining_Cap: ℤ = (Max_Total_Size ∸ Aggregate_Size);
Constraint Aggregate_Size ≤ Max_Total_Size;

Oper Advance(**updates** P: Nstd_Lst_Pos);
requires |P.Rem| ≥ 1;
ensures P.Prec = #P.Prec∘Prt_btwn(0, 1, #P.Rem) **and**
P.Rem = Prt_btwn(1, |#P.Rem|, #P.Rem) **and** P.Path = #P.Path;

Oper Rem_Length(**rest** P: Nstd_Lst_Pos): Integer; (*Remainder Length *)
ensures Rem_Length = (|P.Rem|);

Oper Pull_Back(**updates** P: Nstd_Lst_Pos);
requires |P.Prec| ≥ 1;
ensures P.Path = #P.Path **and** P.Prec = Prt_btwn(0, |#P.Prec| ∸ 1, #P.Prec) **and**
P.Rem = Prt_btwn(|#P.Prec| ∸ 1, |#P.Prec|, #P.Prec)∘#P.Rem;

Oper Prec_Length(**rest** P: Nstd_Lst_Pos): Integer;
ensures Prec_Length = (|P.Prec|);

Oper Insert_Sublist(**updates** P: Nstd_Lst_Pos; **alt** List_Lab: Label);
requires Remaining_Cap > 0;
ensures P.Rem = ⟨Jn(#List_Lab, Λ)⟩∘#P.Rem **and** P.Prec = #P.Prec **and** P.Path = #P.Path;

Oper Open_Sublist(**updates** P: Nstd_List_Pos);
requires |P.Rem| ≥ 1;
ensures P.Prec = Λ **and** P.Rem = Drop_Dn(#P.Prec, #P.Rem).TS **and** P.Path = #P.Path∘⟨Drop_Dn(#P.Prec, #P.Rem).St⟩;

Oper Close_Sublist(**updates** P: Nstd_Lst_Pos);
requires |P.Path| ≥ 1;
ensures P.Path = Prt_btwn(0, |#P.Path| ∸ 1, #P.Path) **and**
P.Prec = ⋦(Prt_btwn(|#P.Path| ∸ 1, |#P.Path|, #P.Path)).LTS **and** P.Rem = ⟨⋦(Prt_btwn(|#P.Path| ∸ 1, |#P.Path|, #P.Path))⟩ Ξ (#P.Prec, #P.Rem).US;

Oper Path_Length(**rest** P: Nstd_Lst_Pos): Integer;
ensures Path_Length = (|P.Path|);

Oper Delete_Sublist(**updates** P: Nstd_Lst_Pos; **replaces** Lab: Entry);
requires |P.Rem| ≥ 1 **and** Rt_Brhs(⋦(Prt_btwn(0, 1, P.Rem))) = Λ;
ensures P.Rem = Prt_btwn(1, |#P.Rem|, #P.Rem) **and** P.Prec = #P.Prec **and** Lab = Rt_Lab(⋦(Prt_btwn(0, 1, #P.Rem))) **and** P.Path = #P.Path;

Oper Skip_Remainder(**updates** P: Nstd_Lst_Pos);
ensures P.Prec = #P.Prec∘#P.Rem **and** P.Rem = Λ **and** P.Path = #P.Path;

Oper Skip_Preceding(**updates** P: Nstd_Lst_Pos);
ensures P.Rem = #P.Prec∘#P.Rem **and** P.Prec = Λ **and** P.Path = #P.Path;

Oper Reset(**updates** P: Nstd_Lst_Pos);
ensures P.Path = Λ **and** P.Prec = Λ **and** P.Rem = Suture_of(#P);

Oper Swap_Label(**updates** P: Nstd_Lst_Pos; **updates** Lab: Label);
requires P.Rem ≠ Λ;
ensures Lab = Rt_Lab(⋦(Prt_btwn(0, 1, #P.Rem))) **and** P.Rem =
⟨Jn(#Lab, Rt_Brhs(⋦(Prt_btwn(0, 1, #P.Rem))))⟩∘Prt_btwn(1, |#P.Rem|, #P.Rem))
and P.Prec = #P.Prec **and** P.Path = #P.Path;

Oper Swap_Rem(**updates** P, Q: Nstd_Lst_Pos);
ensures P.Rem = #Q.Rem **and** Q.Rem = #P.Rem **and**
P.Prec = #P.Prec **and** Q.Prec = #Q.Prec **and**
P.Path = #P.Path **and** Q.Path = #Q.Path;

Oper Swap_Prec(**updates** P, Q: Nstd_Lst_Pos);
ensures P. Prec = #Q. Prec **and** Q. Prec = #P. Prec **and** P. Rem=#P. Rem **and** Q.
Rem=#Q. Rem **and** P. Path=#P. Path **and** Q. Path = #Q. Path;

Oper Swap_w_List(**updates** P, Q: Nstd_Lst_Pos);
ensures P.Prec = #Q.Prec **and** P.Rem = #Q.Rem **and** Q.Path = #Q.Path∘#P.Path
and Q.Prec = #P.Prec **and** Q.Rem = #P.Rem;

Oper Rem_Capacity(): Integer;
ensures Rem_Capacity = (Remaining_Cap);

Oper Clear(**clr** P: Nstd_Lst_Pos);

end Nested_List_Template;

Appendix F Sample Sequent Proof Using Uni-Prover

```
VC 0_1:
Ensures Clause of Delete_Remainder at Obvious_Deletion_Realiz.rb(4:11)

Goal:
(P'.Path = P.Path)

Given(s):
1. (P'.Path = P.Path)
2. (P'.Rem_Tr = Empty_Tree)
3. (Q'.Rem_Tr = P.Rem_Tr)
4. (Q'.Path = Empty_String)


[Prover Result]: Proved
_____________________

[Label(s)]:
    <= -> 1
    = -> 2
    P'.Path -> 3
    P.Path -> 4
    P'.Rem_Tr -> 5
    Empty_Tree -> 6
    Q'.Rem_Tr -> 7
    P.Rem_Tr -> 8
    Q'.Path -> 9
    Empty_String -> 10

_____________________

[Cluster Argument Array]:
    |nextClusterArg=0||prevClusterArg=0||ccNumber=0||clusterNumber=0||nexIndexWithSameCCInSameLevel=0||alternativeArg=0|
    |nextClusterArg=0||prevClusterArg=0||ccNumber=0||clusterNumber=8||nexIndexWithSameCCInSameLevel=0||alternativeArg=0|

[Cluster Array]:
    |treeNodeLabel=0||indexToArgumentList=0||indexToCongruenceClass=0||nextStandCluster=0||previousStandCluster=0||dominantCluster=0||nextWithSameArg=0|
    |treeNodeLabel=3||indexToArgumentList=1||indexToCongruenceClass=1||nextStandCluster=0||previousStandCluster=0||dominantCluster=1||nextWithSameArg=0|
    |treeNodeLabel=4||indexToArgumentList=1||indexToCongruenceClass=1||nextStandCluster=0||previousStandCluster=0||dominantCluster=2||nextWithSameArg=1|
    |treeNodeLabel=5||indexToArgumentList=1||indexToCongruenceClass=3||nextStandCluster=0||previousStandCluster=0||dominantCluster=3||nextWithSameArg=2|
    |treeNodeLabel=6||indexToArgumentList=1||indexToCongruenceClass=3||nextStandCluster=0||previousStandCluster=0||dominantCluster=4||nextWithSameArg=3|
    |treeNodeLabel=7||indexToArgumentList=1||indexToCongruenceClass=5||nextStandCluster=0||previousStandCluster=0||dominantCluster=5||nextWithSameArg=4|
    |treeNodeLabel=8||indexToArgumentList=1||indexToCongruenceClass=5||nextStandCluster=0||previousStandCluster=0||dominantCluster=6||nextWithSameArg=5|
    |treeNodeLabel=9||indexToArgumentList=1||indexToCongruenceClass=7||nextStandCluster=0||previousStandCluster=0||dominantCluster=7||nextWithSameArg=6|
    |treeNodeLabel=10||indexToArgumentList=1||indexToCongruenceClass=7||nextStandCluster=0||previousStandCluster=0||dominantCluster=8||nextWithSameArg=7|

[Stand Array]:
    |treeNodeLabel=0||firstStandCluster=0||standTag=0||nextCCStand=0||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=3||firstStandCluster=1||standTag=1||nextCCStand=2||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=4||firstStandCluster=2||standTag=2||nextCCStand=0||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=5||firstStandCluster=3||standTag=3||nextCCStand=4||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=6||firstStandCluster=4||standTag=4||nextCCStand=0||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=7||firstStandCluster=5||standTag=5||nextCCStand=6||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=8||firstStandCluster=6||standTag=6||nextCCStand=0||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=9||firstStandCluster=7||standTag=7||nextCCStand=8||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=10||firstStandCluster=8||standTag=8||nextCCStand=0||nextVrtyStand=0||prvVrtyStand=0|

[Class Array]:
    |firstStand=1||classTag=1||lastArgStringPosition=0||dominantCClass=1|
    |firstStand=2||classTag=2||lastArgStringPosition=0||dominantCClass=1|
    |firstStand=3||classTag=3||lastArgStringPosition=0||dominantCClass=3|
    |firstStand=4||classTag=4||lastArgStringPosition=0||dominantCClass=3|
    |firstStand=5||classTag=5||lastArgStringPosition=0||dominantCClass=5|
    |firstStand=6||classTag=6||lastArgStringPosition=0||dominantCClass=5|
    |firstStand=7||classTag=7||lastArgStringPosition=0||dominantCClass=7|
    |firstStand=8||classTag=8||lastArgStringPosition=0||dominantCClass=7|

_____________________
```

```
VC 0_2:
Ensures Clause of Delete_Remainder at Obvious_Deletion_Realiz.rb(4:11)

Goal:
(P'.Rem_Tr = Empty_Tree)

Given(s):
1. (P'.Path = P.Path)
2. (P'.Rem_Tr = Empty_Tree)
3. (Q'.Rem_Tr = P.Rem_Tr)
4. (Q'.Path = Empty_String)


[Prover Result]: Proved
_____________________

[Label(s)]:
    <= -> 1
    = -> 2
    P'.Path -> 3
    P.Path -> 4
    P'.Rem_Tr -> 5
    Empty_Tree -> 6
    Q'.Rem_Tr -> 7
    P.Rem_Tr -> 8
    Q'.Path -> 9
    Empty_String -> 10

_____________________

[Cluster Argument Array]:
    |nextClusterArg=0||prevClusterArg=0||ccNumber=0||clusterNumber=0||nexIndexWithSameCCInSameLevel=0||alternativeArg=0|
    |nextClusterArg=0||prevClusterArg=0||ccNumber=0||clusterNumber=8||nexIndexWithSameCCInSameLevel=0||alternativeArg=0|

[Cluster Array]:
    |treeNodeLabel=0||indexToArgumentList=0||indexToCongruenceClass=0||nextStandCluster=0||previousStandCluster=0||dominantCluster=0||nextWithSameArg=0|
    |treeNodeLabel=3||indexToArgumentList=1||indexToCongruenceClass=1||nextStandCluster=0||previousStandCluster=0||dominantCluster=1||nextWithSameArg=0|
    |treeNodeLabel=4||indexToArgumentList=1||indexToCongruenceClass=1||nextStandCluster=0||previousStandCluster=0||dominantCluster=2||nextWithSameArg=1|
    |treeNodeLabel=5||indexToArgumentList=1||indexToCongruenceClass=3||nextStandCluster=0||previousStandCluster=0||dominantCluster=3||nextWithSameArg=2|
    |treeNodeLabel=6||indexToArgumentList=1||indexToCongruenceClass=3||nextStandCluster=0||previousStandCluster=0||dominantCluster=4||nextWithSameArg=3|
    |treeNodeLabel=7||indexToArgumentList=1||indexToCongruenceClass=5||nextStandCluster=0||previousStandCluster=0||dominantCluster=5||nextWithSameArg=4|
    |treeNodeLabel=8||indexToArgumentList=1||indexToCongruenceClass=5||nextStandCluster=0||previousStandCluster=0||dominantCluster=6||nextWithSameArg=5|
    |treeNodeLabel=9||indexToArgumentList=1||indexToCongruenceClass=7||nextStandCluster=0||previousStandCluster=0||dominantCluster=7||nextWithSameArg=6|
    |treeNodeLabel=10||indexToArgumentList=1||indexToCongruenceClass=7||nextStandCluster=0||previousStandCluster=0||dominantCluster=8||nextWithSameArg=7|

[Stand Array]:
    |treeNodeLabel=0||firstStandCluster=0||standTag=0||nextCCStand=0||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=3||firstStandCluster=1||standTag=1||nextCCStand=2||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=4||firstStandCluster=2||standTag=2||nextCCStand=0||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=5||firstStandCluster=3||standTag=3||nextCCStand=4||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=6||firstStandCluster=4||standTag=4||nextCCStand=0||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=7||firstStandCluster=5||standTag=5||nextCCStand=6||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=8||firstStandCluster=6||standTag=6||nextCCStand=0||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=9||firstStandCluster=7||standTag=7||nextCCStand=8||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=10||firstStandCluster=8||standTag=8||nextCCStand=0||nextVrtyStand=0||prvVrtyStand=0|

[Class Array]:
    |firstStand=1||classTag=1||lastArgStringPosition=0||dominantCClass=1|
    |firstStand=2||classTag=2||lastArgStringPosition=0||dominantCClass=1|
    |firstStand=3||classTag=3||lastArgStringPosition=0||dominantCClass=3|
    |firstStand=4||classTag=4||lastArgStringPosition=0||dominantCClass=3|
    |firstStand=5||classTag=5||lastArgStringPosition=0||dominantCClass=5|
    |firstStand=6||classTag=6||lastArgStringPosition=0||dominantCClass=5|
    |firstStand=7||classTag=7||lastArgStringPosition=0||dominantCClass=7|
    |firstStand=8||classTag=8||lastArgStringPosition=0||dominantCClass=7|

_____________________
```

```
VC 0_3:
Ensures Clause of Delete_Remainder at Obvious_Deletion_Realiz.rb(4:11)

Goal:
((Remaining_Cap + N_C(Zip_Op(Q'.Path, Q'.Rem_Tr))) = (Remaining_Cap + N_C(P.Rem_Tr)))

Given(s):
1. (Q'.Path = Empty_String)
2. (P'.Rem_Tr = Empty_Tree)
3. (Q'.Rem_Tr = P.Rem_Tr)
4. (P'.Path = P.Path)


[Prover Result]: Not Proved  -- Requires theorems to be verified
_____________________

[Label(s)]:
    <= -> 1
    = -> 2
    Q'.Path -> 3
    Empty_String -> 4
    P'.Rem_Tr -> 5
    Empty_Tree -> 6
    Q'.Rem_Tr -> 7
    P.Rem_Tr -> 8
    P'.Path -> 9
    P.Path -> 10
    Remaining_Cap -> 11
    Zip_Op -> 12
    N_C -> 13
    + -> 14


_____________________

[Cluster Argument Array]:
    |nextClusterArg=0||prevClusterArg=0||ccNumber=0||clusterNumber=0||nexIndexWithSameCCInSameLevel=0||alternativeArg=0|
    |nextClusterArg=9||prevClusterArg=0||ccNumber=0||clusterNumber=12||nexIndexWithSameCCInSameLevel=0||alternativeArg=0|
    |nextClusterArg=3||prevClusterArg=1||ccNumber=1||clusterNumber=0||nexIndexWithSameCCInSameLevel=0||alternativeArg=0|
    |nextClusterArg=0||prevClusterArg=2||ccNumber=5||clusterNumber=11||nexIndexWithSameCCInSameLevel=0||alternativeArg=0|
    |nextClusterArg=0||prevClusterArg=1||ccNumber=11||clusterNumber=13||nexIndexWithSameCCInSameLevel=0||alternativeArg=5|
    |nextClusterArg=8||prevClusterArg=1||ccNumber=9||clusterNumber=0||nexIndexWithSameCCInSameLevel=0||alternativeArg=7|
    |nextClusterArg=0||prevClusterArg=5||ccNumber=13||clusterNumber=14||nexIndexWithSameCCInSameLevel=0||alternativeArg=0|
    |nextClusterArg=0||prevClusterArg=1||ccNumber=5||clusterNumber=15||nexIndexWithSameCCInSameLevel=0||alternativeArg=2|
    |nextClusterArg=0||prevClusterArg=5||ccNumber=15||clusterNumber=16||nexIndexWithSameCCInSameLevel=0||alternativeArg=6|
    |nextClusterArg=10||prevClusterArg=1||ccNumber=14||clusterNumber=0||nexIndexWithSameCCInSameLevel=0||alternativeArg=4|
    |nextClusterArg=0||prevClusterArg=9||ccNumber=16||clusterNumber=17||nexIndexWithSameCCInSameLevel=0||alternativeArg=0|

[Cluster Array]:
    |treeNodeLabel=0||indexToArgumentList=0||indexToCongruenceClass=0||nextStandCluster=0||previousStandCluster=0||dominantCluster=0||nextWithSameArg=0|
    |treeNodeLabel=3||indexToArgumentList=1||indexToCongruenceClass=1||nextStandCluster=0||previousStandCluster=0||dominantCluster=1||nextWithSameArg=0|
    |treeNodeLabel=4||indexToArgumentList=1||indexToCongruenceClass=1||nextStandCluster=0||previousStandCluster=0||dominantCluster=2||nextWithSameArg=1|
    |treeNodeLabel=5||indexToArgumentList=1||indexToCongruenceClass=3||nextStandCluster=0||previousStandCluster=0||dominantCluster=3||nextWithSameArg=2|
    |treeNodeLabel=6||indexToArgumentList=1||indexToCongruenceClass=3||nextStandCluster=0||previousStandCluster=0||dominantCluster=4||nextWithSameArg=3|
    |treeNodeLabel=7||indexToArgumentList=1||indexToCongruenceClass=5||nextStandCluster=0||previousStandCluster=0||dominantCluster=5||nextWithSameArg=4|
    |treeNodeLabel=8||indexToArgumentList=1||indexToCongruenceClass=5||nextStandCluster=0||previousStandCluster=0||dominantCluster=6||nextWithSameArg=5|
    |treeNodeLabel=9||indexToArgumentList=1||indexToCongruenceClass=7||nextStandCluster=0||previousStandCluster=0||dominantCluster=7||nextWithSameArg=6|
    |treeNodeLabel=10||indexToArgumentList=1||indexToCongruenceClass=7||nextStandCluster=0||previousStandCluster=0||dominantCluster=8||nextWithSameArg=7|
    |treeNodeLabel=11||indexToArgumentList=1||indexToCongruenceClass=9||nextStandCluster=0||previousStandCluster=0||dominantCluster=9||nextWithSameArg=8|
    |treeNodeLabel=12||indexToArgumentList=1||indexToCongruenceClass=10||nextStandCluster=0||previousStandCluster=0||dominantCluster=10||nextWithSameArg=9|
    |treeNodeLabel=12||indexToArgumentList=3||indexToCongruenceClass=11||nextStandCluster=0||previousStandCluster=0||dominantCluster=11||nextWithSameArg=0|
    |treeNodeLabel=13||indexToArgumentList=1||indexToCongruenceClass=12||nextStandCluster=0||previousStandCluster=0||dominantCluster=12||nextWithSameArg=10|
    |treeNodeLabel=13||indexToArgumentList=4||indexToCongruenceClass=13||nextStandCluster=0||previousStandCluster=0||dominantCluster=13||nextWithSameArg=0|
    |treeNodeLabel=14||indexToArgumentList=6||indexToCongruenceClass=14||nextStandCluster=0||previousStandCluster=0||dominantCluster=14||nextWithSameArg=0|
    |treeNodeLabel=13||indexToArgumentList=7||indexToCongruenceClass=15||nextStandCluster=0||previousStandCluster=0||dominantCluster=15||nextWithSameArg=0|
    |treeNodeLabel=14||indexToArgumentList=8||indexToCongruenceClass=16||nextStandCluster=0||previousStandCluster=0||dominantCluster=16||nextWithSameArg=0|
    |treeNodeLabel=2||indexToArgumentList=10||indexToCongruenceClass=17||nextStandCluster=0||previousStandCluster=0||dominantCluster=17||nextWithSameArg=0|

[Stand Array]:
    |treeNodeLabel=0||firstStandCluster=0||standTag=0||nextCCStand=0||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=3||firstStandCluster=1||standTag=1||nextCCStand=2||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=4||firstStandCluster=2||standTag=2||nextCCStand=0||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=5||firstStandCluster=3||standTag=3||nextCCStand=4||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=6||firstStandCluster=4||standTag=4||nextCCStand=0||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=7||firstStandCluster=5||standTag=5||nextCCStand=6||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=8||firstStandCluster=6||standTag=6||nextCCStand=0||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=9||firstStandCluster=7||standTag=7||nextCCStand=8||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=10||firstStandCluster=8||standTag=8||nextCCStand=0||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=11||firstStandCluster=9||standTag=9||nextCCStand=0||nextVrtyStand=0||prvVrtyStand=0|
    |treeNodeLabel=12||firstStandCluster=10||standTag=10||nextCCStand=0||nextVrtyStand=11||prvVrtyStand=0|
    |treeNodeLabel=12||firstStandCluster=11||standTag=11||nextCCStand=0||nextVrtyStand=0||prvVrtyStand=10|
    |treeNodeLabel=13||firstStandCluster=12||standTag=12||nextCCStand=0||nextVrtyStand=13||prvVrtyStand=0|
    |treeNodeLabel=13||firstStandCluster=13||standTag=13||nextCCStand=0||nextVrtyStand=15||prvVrtyStand=12|
    |treeNodeLabel=14||firstStandCluster=14||standTag=14||nextCCStand=0||nextVrtyStand=16||prvVrtyStand=0|
    |treeNodeLabel=13||firstStandCluster=15||standTag=15||nextCCStand=0||nextVrtyStand=0||prvVrtyStand=13|
    |treeNodeLabel=14||firstStandCluster=16||standTag=16||nextCCStand=0||nextVrtyStand=0||prvVrtyStand=14|
    |treeNodeLabel=2||firstStandCluster=17||standTag=17||nextCCStand=0||nextVrtyStand=0||prvVrtyStand=0|

[Class Array]:
    |firstStand=1||classTag=1||lastArgStringPosition=1||dominantCClass=1|
    |firstStand=2||classTag=2||lastArgStringPosition=0||dominantCClass=1|
    |firstStand=3||classTag=3||lastArgStringPosition=0||dominantCClass=3|
    |firstStand=4||classTag=4||lastArgStringPosition=0||dominantCClass=3|
    |firstStand=5||classTag=5||lastArgStringPosition=2||dominantCClass=5|
    |firstStand=6||classTag=6||lastArgStringPosition=0||dominantCClass=5|
    |firstStand=7||classTag=7||lastArgStringPosition=0||dominantCClass=7|
    |firstStand=8||classTag=8||lastArgStringPosition=0||dominantCClass=7|
    |firstStand=9||classTag=9||lastArgStringPosition=1||dominantCClass=9|
    |firstStand=10||classTag=10||lastArgStringPosition=0||dominantCClass=10|
    |firstStand=11||classTag=11||lastArgStringPosition=1||dominantCClass=11|
    |firstStand=12||classTag=12||lastArgStringPosition=0||dominantCClass=12|
    |firstStand=13||classTag=13||lastArgStringPosition=2||dominantCClass=13|
    |firstStand=14||classTag=14||lastArgStringPosition=1||dominantCClass=14|
    |firstStand=15||classTag=15||lastArgStringPosition=1||dominantCClass=15|
    |firstStand=16||classTag=16||lastArgStringPosition=2||dominantCClass=16|
    |firstStand=17||classTag=17||lastArgStringPosition=0||dominantCClass=17|
```

Bibliography

[1] Bruce M. Adcock. *Working Towards the Verified Software Process*. Phd thesis, Department of Computer Science and Engineering, 2010.

[2] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The KeY platform for verification and analysis of java programs. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools and Experiments*, volume 8471 of *Lecture Notes in Computer Science*, pages 55–71. Springer International Publishing, 2014.

[3] Vytautas Astrauskas, Aurel Bílý, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. The Prusti Project: Formal Verification for Rust. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods*, Lecture Notes in Computer Science, pages 88–108, Cham, 2022. Springer International Publishing.

[4] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):147:1–147:30, October 2019.

[5] Haniel Barbosa. Efficient Instantiation Techniques in SMT (Work In Progress). volume 1635, page 1, July 2016.

[6] Mike Barnett, Bor-Yuh Chang, Robert DeLine, Bart Jacobs, and K. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank de Boer, Marcello Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer Berlin / Heidelberg, 2006. 10.1007/11804192_17.

[7] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, Lecture Notes in Computer Science, pages 49–69, Berlin, Heidelberg, 2005. Springer.

[8] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, Berlin Heidelberg, 2004.

[9] Paolo Bucci, Timothy J. Long, and Bruce W. Weide. Do we really teach abstraction? *ACM SIGCSE Bulletin*, 33(1):26–30, February 2001.

[10] Charles T. Cook. A web-integrated environment for component-based software reasoning, 2011.

[11] Charles T. Cook, Svetlana V. Drachova-Strang, Yu-Shan Sun, Murali Sitaraman, Jeffrey C. Carver, and Joseph Hollingsworth. Specification and reasoning in SE projects using a Web IDE. In *2013 26th International Conference on Software Engineering Education and Training (CSEE&T)*, pages 229–238, May 2013. ISSN: 2377-570X.

[12] Charles T. Cook, Heather Harton, Hampton Smith, and Murali Sitaraman. Specification engineering and modular verification using a web-integrated verifying compiler. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1379–1382, June 2012. ISSN: 1558-1225.

[13] Leonardo de Moura and Nikolaj Bjorner. Generalized, efficient array decision procedures. In *2009 Formal Methods in Computer-Aided Design*, pages 45–52, Austin, TX, November 2009. IEEE.

[14] Leonardo de Moura and Nikolaj Bjørner. Efficient E-Matching for SMT Solvers. In Frank Pfenning, editor, *Automated Deduction – CADE-21*, Lecture Notes in Computer Science, pages 183–198, Berlin, Heidelberg, 2007. Springer.

[15] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 337–340, Berlin, Heidelberg, 2008. Springer.

[16] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, Lecture Notes in Computer Science, pages 378–388, Cham, 2015. Springer International Publishing.

[17] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 52(3):365–473, May 2005.

[18] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 737–744, Cham, 2014. Springer International Publishing.

[19] Bruno Dutertre and Leonardo de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 81–94, Berlin, Heidelberg, 2006. Springer.

[20] Jean-Christophe Filliâtre and Andrei Paskevich. Why3: Where programs meet provers. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 125–128, Berlin, Heidelberg, 2013. Springer-Verlag.

[21] Megan Fowler, Eileen T. Kraemer, Yu-Shan Sun, Murali Sitaraman, Jason O. Hallstrom, and Joseph E. Hollingsworth. Tool-Aided Assessment of Difficulties in Learning Formal Design-by-Contract Assertions. In *Proceedings of the 4th European Conference on Software Engineering Education*, ECSEE '20, pages 52–60, New York, NY, USA, June 2020. Association for Computing Machinery.

[22] Yeting Ge, Clark Barrett, and Cesare Tinelli. Solving quantified verification conditions using satisfiability modulo theories. *Annals of Mathematics and Artificial Intelligence*, 55(1-2):101–122, February 2009.

[23] Douglas E. Harms and Bruce W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Trans. Softw. Eng.*, 17(5):424–435, May 1991.

[24] Heather Harton. *Mechanical and Modular Verification Condition Generation for Object-Based Software*. Phd dissertation, School of Computing, 2011.

[25] Wayne D. Heym. *Computer Program Verification: Improvements for Human Reasoning*. Phd thesis, Department of Computer and Information Science, 1995.

[26] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[27] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, January 2003.

[28] Joseph E. Hollingsworth, Sethu Sreerama, Bruce W. Weide, and Sergey Zhupanov. Part IV: RESOLVE components in Ada and C++. *ACM SIGSOFT Software Engineering Notes*, 19(4):52–63, October 1994.

[29] Jieh Hsiang, Hélène Kirchner, Pierre Lescanne, and Michaël Rusinowitch. The term rewriting approach to automated theorem proving. *The Journal of Logic Programming*, 14(1-2):71–99, October 1992.

[30] James C. King. A PROGRAM VERIFIER. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, September 1969.

[31] Jason Kirschenbaum, Bruce Adcock, Derek Bronish, Hampton Smith, Heather Harton, Murali Sitaraman, and Bruce W. Weide. Verifying Component-Based Software: Deep Mathematics or Simple Bookkeeping? In Stephen H. Edwards and Gregory Kulczycki, editors, *Formal Foundations of Reuse and Domain Engineering*, Lecture Notes in Computer Science, pages 31–40, Berlin, Heidelberg, 2009. Springer.

[32] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, August 2019. Google-Books-ID: 0Vv6DwAAQBAJ.

[33] Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. The 1st verified software competition: Experience report. In Michael Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods*, volume 6664 of *Lecture Notes in Computer Science*, pages 154–168. Springer Berlin Heidelberg, 2011.

[34] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 1–35, Berlin, Heidelberg, 2013. Springer.

[35] Joan Krone. *The Role of Verification in Software Reusability*. Phd thesis, Department of Computer and Information Science, 1988.

[36] Gregory Kulczycki. *Direct Reasoning*. Phd dissertation, School of Computing, 2004.

[37] Gregory Kulczycki, Murali Sitaraman, Joan Krone, Joseph E. Hollingsworth, William F. Ogden, Bruce W. Weide, Paolo Bucci, Charles T. Cook, Svetlana V. Drachova-Strang, Blair Durkee, Heather Harton, Wayne Heym, Dustin Hoffman, Hampton Smith, Yu-Shan Sun, Aditi Tagore, Nighat Yasmin, and Diego Zaccai. A Language for Building Verified Software Components. In John Favaro and Maurizio Morisio, editors, *Safe and Secure Software Reuse*, Lecture Notes in Computer Science, pages 308–314, Berlin, Heidelberg, 2013. Springer.

[38] Gregory Kulczycki, Hampton Smith, Heather Harton, Murali Sitaraman, William F. Ogden, and Joseph E. Hollingsworth. The location linking concept: A basis for verification of code using pointers. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Verified Software: Theories, Tools, Experiments*, volume 7152 of *Lecture Notes in Computer Science*, pages 34–49. Springer Berlin Heidelberg, 2012.

[39] Shuvendu Lahiri and Shaz Qadeer. Back to the future: revisiting precise program verification using SMT solvers. *ACM SIGPLAN Notices*, 43(1):171–182, January 2008.

[40] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a Java Modeling Language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, October 1998. `http://www-dse.doc.ic.ac.uk/~sue/oopsla/cfp.html`.

[41] K. R. M. Leino and Clément Pit-Claudel. Trigger Selection Strategies to Stabilize Program Verifiers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 361–381, Cham, 2016. Springer International Publishing.

[42] K. Rustan M. Leino. Developing verified programs with Dafny. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1488–1490, May 2013. ISSN: 1558-1225.

[43] Nicodemus Mbwambo. A Well-Designed, Tree-Based, Generic Map Component to Challenge the Progress towards Automated Verification. *All Theses*, May.

[44] Bertrand Meyer. Eiffel: A language and environment for software engineering. *Journal of Systems and Software*, 8(3):199–246, June 1988.

[45] Bertrand Meyer. The Start of an Eiffel Standard., 2002.

[46] J. Strother Moore. Milestones from the Pure Lisp theorem prover to ACL2. *Formal Aspects of Computing*, 31(6):699–732, December 2019.

[47] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, Lecture Notes in Computer Science, pages 625–635, Cham, 2021. Springer International Publishing.

[48] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science, pages 41–62, Berlin, Heidelberg, 2016. Springer.

[49] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, November 2006.

[50] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science. Springer-Verlag, Berlin Heidelberg, 2002.

[51] William F. Ogden, Murali Sitaraman, Bruce W. Weide, and Stuart H. Zweben. Part I: the RESOLVE framework and discipline: a research synopsis. *ACM SIGSOFT Software Engineering Notes*, 19(4):23–28, October 1994.

[52] Edward E. Ogheneovo. Revisiting Cook-Levin theorem using NP-Completeness and Circuit-SAT. *International Journal of Advanced Engineering Research and Science*, 7(3):206–213, 2020.

[53] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[54] David L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. In Manfred Broy and Ernst Denert, editors, *Pioneers and Their Contributions to Software Engineering: sd&m Conference on Software Pioneers, Bonn, June 28/29, 2001, Original Historic Contributions*, pages 479–498. Springer, Berlin, Heidelberg, 2001.

[55] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters, and Clark Barrett. Quantifier Instantiation Techniques for Finite Model Finding in SMT. In Maria Paola Bonacina, editor, *Automated Deduction – CADE-24*, Lecture Notes in Computer Science, pages 377–391, Berlin, Heidelberg, 2013. Springer.

[56] Murali Sitaraman, Bruce Adcock, Jeremy Avigad, Derek Bronish, Paolo Bucci, David Frazier, Harvey M. Friedman, Heather Harton, Wayne Heym, Jason Kirschenbaum, Joan Krone, Hampton Smith, and Bruce W. Weide. Building a push-button resolve verifier: Progress and challenges. *Formal Aspects of Computing*, 23(5):607–626, 2011.

[57] Murali Sitaraman and Bruce W. Weide. A Synopsis of Twenty Five Years of RESOLVE PhD Research Efforts. *ACM SIGSOFT Software Engineering Notes*, 43(3):17–17, December 2018.

[58] Hampton Smith. *Engineering Specifications and Mathematics for Verified Software*. Phd dissertation, School of Computing, 2013.

[59] Hampton Smith, Kim Roche, Murali Sitaraman, Joan Krone, and William Ogden. Integrating Math Units and Proof Checking for Specification and Verification. In *Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008)*, pages 59 – 66, Atlanta, Georgia, November 2008.

[60] Hampton Smith, Kim Roche, Murali Sitaraman, Joan Krone, and William F. Ogden. Integrating math units and proof checking for specification and verification. In *Proceedings of the Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008)*, pages 59–66, 2008.

[61] Yu-Shan Sun. Towards Automated Verification of Object-Based Software with Reference Behavior. *All Dissertations*, December 2018.

[62] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova. AutoProof: Auto-Active Functional Verification of Object-Oriented Programs. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 566–580, Berlin, Heidelberg, 2015. Springer.

[63] Bruce W. Weide and Wayne D. Heym. Specification and verification with references. In *Proceedings OOPSLA Workshop on Specification and Verification of Component-Based Systems*, pages 50–59, 2001.

[64] B.W. Weide, W.F. Ogden, and M. Sitaraman. Recasting algorithms to encourage reuse. *IEEE Software*, 11(5):80–88, September 1994.

[65] Benjamin Weiß. *Deductive verification of object-oriented software: dynamic frames, dynamic logic and predicate abstraction*. KIT Scientific Publ, Karlsruhe, 2011. OCLC: 707139970.

[66] Daniel Welch. Scaling Up Automated Verification: A Case Study and a Formalization IDE for Building High Integrity Software. *All Dissertations*, December 2019.

[67] Nighat Yasmin. *Specification and Mechanical Verification of Performance Profiles of Software Components*. Phd thesis, Department of Engineering Science, 2015.

[68] Erik Zawadzki, Geoffrey Gordon, and Andre Platzer. An Instantiation-Based Theorem Prover for First-Order Programming. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 855–863. JMLR Workshop and Conference Proceedings, June 2011.