DECISION PROCEDURES FOR EQUATIONALLY BASED REASONING

BY

JOSEPH D. HENDRIX

B.A., The University of Texas at Austin, 2000
B.S., The University of Texas at Austin, 2000

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2008

Urbana, Illinois

Doctoral Committee:

Professor José Meseguer, Chair
Professor Gul Agha
Professor Deepak Kapur, University of New Mexico
Associate Professor Cesare Tinelli, University of Iowa
Associate Professor Mahesh Viswanathan

Abstract

This work develops new automated reasoning techniques for verifying the correctness of equationally specified programs. These techniques are not just theoretical, but have been implemented, and applied to actual program verification projects. Although the work spans several different areas, a major theme of this work is to develop better techniques at the boundary between decidable and undecidable problems. That is, this work seeks out not just positive decidability results, but ways to extend the underlying techniques to be effective on problems outside of decidable subclasses.

For program verification to succeed, we feel that two important directions must be pursued: (1) considering more expressive logics to allow designers to more easily specify systems, and (2) develop decision procedures that can reason efficiently about these more sophsticated logics. This work pursues both directions, and the main topics addressed include: new decidability and undecidability results for equational tree automata (Chapter 3), order-sorted unification (Chapter 4), sufficient completeness for specifications with partiality and rewriting modulo axioms (Chapter 5), completeness problems for context-sensitive specifications (Chapter 6), coverset induction in membership equational logic (Chapter 7), and a case study for verifying properties of powerlists with the Maude ITP (Chapter 8).

Each of these theoretical topics have lead to the development of new libraries and tools. Two of the tools have already been used in external projects including our tree automata library's integration into the ACTAS protocol verification tool [126], and the order-sorted unification procedures use in the Maude-NRL protocol analyzer [49].

To Tanya Crenshaw

Acknowledgments

I would not have finished this dissertation without the support, encouragement and advice of many people.

My adviser, José Meseguer, has been a consistent source of research ideas and enthusiasm for this work. He has shown me the value of deep thinking as well as a broad experience in different areas. I am grateful for the unwavering financial support that he has provided me through research assistantships and conference travel.

I thank the other members of my dissertation committee. They have all eagerly provided me with advice on both my research and career. I have learned from Gul Agha the importance of concise programming models particularly when considering concurrency. Much of my research follows work developed by Deepak Kapur, and I have learned a great deal from him and his papers. Cesare Tinelli graciously took the time to mentor me on satisfiability modulo theories. Mahesh Viswanathan taught me virtually all I know of machine learning and finite model theory.

I also thank my main research collaborators and other researchers who influenced my work. I wrote my first conference paper while visiting Manuel Clavel. His persistence in encouraging me to write and rewrite again and again and again forced me to become a much better writer. Hitoshi Ohsaki introduced me to to equational tree automata, a subject which I have enjoyed studying immensely, and inviting me to visit Japan for a summer. Santiago Escobar has been a good friend, and developed interesting applications on top of my order-sorted unification procedure. Finally, Nikolaj Bjørner arranged an exciting summer in Seattle at Microsoft where I learned a tremendous amount, met many interesting researchers, and heard some great talks.

I am also grateful for many of the great staff at the University of Illinois. I especially thank Andrea Whitesel has handled many different travel arrangements and administrative task with extraordinary efficiency. Mary Beth Kelley was also extraordinarily helpful in finishing the degree. Without her help on resolving some last minute coursework problems, my graduation would likely have been significantly delayed. In addition to them, I thank the staff and technical support including Fran Bell, Barb Cicone, Shirley Finke, Molly Flesner, David Mussulman, and Chuck Thompson.

My office mates and fellow students over the years for great conversations and interesting research ideas. My office would have been much drab without them. I especially thank my first office mate, Miguel Palomino, who patiently explained many mathematical concepts even when I had already asked before and forgotten. I thank Azadeh Farzan for her friendship over the years. She was the first person to graduate in our research group, and showed me that it was actually possible to get a Ph.D. In addition Mike Katelman, Camilo Rocha, Ralf Sasse, Traian Serbanuta, Ambarish Sridharanarayanan, and Ram Prasad Venkatesan have been a source for great conversation, encouragement, and feedback on my research.

Finally, I thank my parents for their support, advice, and consistent encouragement for graduating. Most of all, I thank Tanya Crenshaw for her friendship, love, and support. She has been there through both good and bad times. For all those times, this thesis is dedicated to her.

Contents

Chapter 1

Introduction

As computer systems have become both more powerful and less expensive, many different applications and services have grown to depend on them. This includes essential safety-critical systems such as power systems, communication networks, and medical devices. These systems have often benefited from computer technology, but they now also depend on the correct operation of computer hardware and the software controlling it. This software has become extraordinarily complex to deal with the diverse requirements of these different applications. Handling this complexity while insuring that the system satisfies all of its requirements has become one of the greatest challenges in software development.

It is widely believed that no single idea is capable of solving this challenge. Numerous software development processes, programming paradigms, and programmer tools have been proposed to help build reliable software that is capable of satisfying its complex requirements. Two of the main approaches to attack this problem have been to: (1) develop new languages, libraries, and development frameworks to enable a developer to more easily specify and implement a system; (2) create analysis and verification tools that enable a developer to verify a system satisfies formally defined requirements.

When it comes to overall system development and validation, these two approaches cannot be considered in isolation. Different languages may have different properties that must be checked for the program to be considered correct, and different analysis techniques may be more or less effective depending on the language. Using a more powerful and highly expressive language may make the validation task easier, by allowing the developer to write a more compact and easier to analyze program, but it may also make that task harder, because existing tools and techniques cannot deal with the new programming constructs in the richer language.

Fundamental to any formal verification attempt is a clear and precise *semantics* to the meaning of both programs and properties. One prominent method to doing that lay in the area of *equational specification*. In this approach, programs are specified *axiomatically* by means of equations $l = r$, and a state of the program is represented by a term t. Programs are *executed* by interpreting the equations as rules $l \rightarrow r$ and replacing instances of l appearing in t with the corresponding instance of r. This approach to program specification is simple

with a clear mathematical semantics, yet quite expressive. Correctness properties of a program can then be expressed in a standard mathematical logic such as first-order logic, while operational properties can be expressed through modal logics such as various *temporal logics* [110].

Although this work spans many different areas, all of it relates to verifying and reasoning about equationally specified programs. A major motivation is to develop techniques for reasoning about specifications with partial operations and advanced algebraic data types. We feel that equational specification represents an elegant and powerful approach for specifying, executing and reasoning about different algorithms, data structures, and systems. However, as the applications and variety of equational specifications has grown, it has been found quite useful to add advanced features to equational specification languages such as support for partiality via sorts and subsorts, and reasoning modulo fundamental structural properties such as associativity and commutatvity. These extra features can allow complex systems to be specified in a significantly simpler and more elegant way, and those systems can be executed by rewrite engines such as Maude that support those features. However, these more expressive formalisms pose a major challenge for *reasoning* about specifications that use these extra features in order to identify errors and verify correctness properties. Both automated reasoning techniques and tools must be extended to handle these more expressive features. A major point of this thesis is to answer that challenge.

In particular, we feel that this work represents a major step forward in reasoning about specifications with partial operations and operations over algebraic data types such as lists, sets, multisets, and trees. We develop new correctness properties and extend existing notions of correctness to more expressive logics in Chapters 5 and 6. We also develop new reasoning techniques for verifying correctness of a specification in Chapters 3, 4, and 7. These techniques are not just theoretical, but have been implemented in a number of different freely available tools for automated reasoning and checking correctness properties in your own specifications. Finally, there is an extended case study using one of the tools extended in this work in Chapter 8.

A diagram depicting the different topics in this work and their connections appear in Figure 1.1. This diagram breaks the topics covered into theoretical results, tools, and a case study using one of the tools. The diagram also references two external protocol verification tools that are not part of this work, but use tools developed as part of this work. In addition, these tools are both used for protocol verification, and illustrate how seemingly disparate techniques such as tree automata and unification may both have applications in the same area.

In the rest of this chapter, we present a very high level overview of the different topics covered by this work and their relationships. We begin with a discussion of topics in the area of equational specification that are relevant to this work. We then discuss two different automated reasoning areas involving equational theories: equational tree automata and order-sorted equational

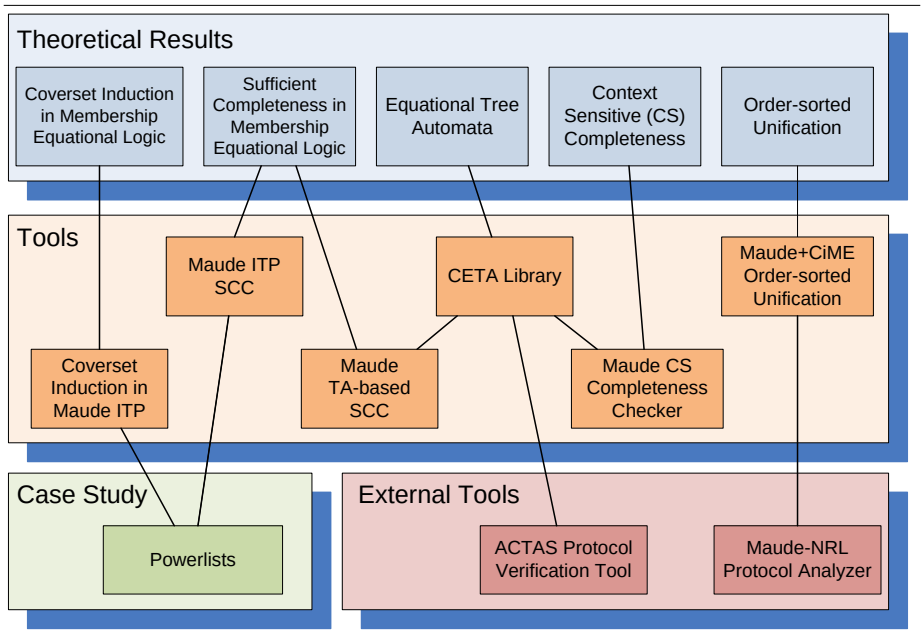


Figure 1.1: Relationships between contributions

unification. We next discuss two different correctness properties for equational specifications: sufficient completeness and different notions of context-sensitive completeness. Finally, we discuss a new theorem proving technique for verifying properties of equational specifications called coverset induction, and how coverset induction can be applied to formally verify the correctness of many different parallel algorithms.

1.1 Equational specification

Equational specification is a simple specification paradigm which strikes a nice balance between expressiveness and verifiability. In equational specification, systems are specified mathematically via *equations* that are executed by *term rewriting*. The equations in a specification can be used to both define functions that operate on data and equivalences between different representations of data. When all of the variables appearing in an equation $l = r$ appear in the left-hand side l, the equation can be interpreted as a oriented rewrite rule $l \rightarrow r$ and used to simplify terms by rewriting. The idea is that the user gives the rewriter an expression to evaluate, and the rewriter replaces subexpressions that match the left-hand side l of a rule $l \rightarrow r$ with the appropriate instance of the right-hand side r. This process is repeated until the expression no longer matches the left-hand side of a rule.

Term rewriting has been heavily studied as a theoretical framework by many textbooks and surveys (e.g., [6, 38, 39]), and is a fundamental algorithm used by

many different theorem provers (e.g., [27, 88, 91, 115, 128]). Term rewriting can be used as a programming paradigm that is quite expressive, yet still possesses a simple algebraic semantics. It can quite naturally model both deterministic and non-deterministic systems, and there are several different programming languages based on these ideas, including ASF-SDF [21], CafeOBJ [51], ELAN [15], Maude [28], and OBJ [52].

Once one has specified a program in a rewrite system, the harder task of verifying that it satisfies its requirements must be confronted. Fortunately, there are a wide variety of systems available for proving properties about an equational specification or its associated rewrite system. For example, there are tools capable of checking termination of a rewrite system (e.g., [36, 44, 61, 104]), confluence of a rewrite system (e.g., [29, 36]), and inductive theorems of the equational specification (e.g., [27, 68, 88]).

As users have gained more experience in specifying different types of systems in these languages, many different extensions have been proposed to make specifying systems easier. Most relevant to my work has been: (1) using more expressive logics such Order-Sorted Logic [63, 64] and Membership-Equational Logic [19, 111] that allow constructor symbols to be *partial* and only defined on relevant data; (2) rewriting *modulo* certain basic axioms of the data such associativity and commutativity; and (3) *conditional* rewriting, where a rule will only be applied if its conditions hold.

These extensions make it easier to express many fundamental algorithms and data structures, but come with a cost — if a user wants to use a formal analysis tool on a specification using one or more of these extensions, the tool must be capable of handling a class of specifications with those extensions. This problem arises quite often in practice, as most theoretical results in term rewriting deal with the simplest logics, yet rewrite engines such as CafeOBJ, Maude, and OBJ are capable of executing specifications with many different features that have not been as deeply studied theoretically. Users of these languages want to take advantage of the features the language offers, but using these features may make validation very difficult, because existing tools capable of verifying the desired property are incapable of handling all of the features the programmer wants to use. The user is often forced to rewrite the specification into a format the formal analysis tool can handle, or to make do with checking fewer properties of the specification.

1.2 Equational tree automata

In applying automated reasoning to verify properties of term rewrite systems, it is often essential to have a framework for reasoning about potentially infinite sets of terms that satisfy properties of interest. Additionally, one may often want to perform operations on those sets such as union, intersection, or complementation. One may also want to check properties like whether a set is empty,

whether a given term belongs to the set, or whether one set is the subset of another. For example, in our later discussion on *sufficient completeness checking*, we want to check whether every term either matches a rule or belongs to a subset of terms called *constructor terms*. A major avenue of research is to identify and develop frameworks capable of reasoning about sets of terms that satisfy interesting properties such as the properties needed for sufficient completeness checking and other applications.

The automata theoretic framework of regular *tree automata* [33] is one major framework that has proven quite useful in many different applications in term rewriting, logic, verification, and databases. Tree automata can be used to recognize potentially infinite sets of terms, and can be equivalently represented as rewrite rules with a particular form, order-sorted theories, or Horn clauses over a monadic signature. The sets of terms that are recognized by tree automata are called *regular tree languages*. Regular tree languages are effectively closed under determinzation, Boolean operations (including intersection, union, and complementation), linear homomorphisms, and inverse-homomorphisms. In addition, they have decidable emptiness, membership, containment, and universality problems.

In many applications, one wants to take the *equational closure* of a language. That is given a language $\mathcal{L}$ and an equational theory $\mathcal{E}$ over the same signature Σ, one wants to reason about the equivalence classes $\mathcal{L}/\mathcal{E} = \{\, [t]_{\mathcal{E}} \mid t \in \mathcal{L} \,\}$ where $[t]_{\mathcal{E}}$ denotes the set of all ground terms equivalent to t modulo $=_{\mathcal{E}}$. In particular, this is important in *rewriting modulo axioms*, because in that extension to term rewriting, the rewrite relation can most naturally be viewed as rewriting $\mathcal{E}$-equivalence classes of terms rather than particular terms.

One approach to representing the equational closure of a language is the *equational tree automata* framework [123]. This framework extends ordinary tree automata by associating an equational theory $\mathcal{E}$ to a regular tree automaton $\mathcal{A}$. Recall that a theory $\mathcal{E}$ is *linear* if for each equation $t = u$, no variable appears more than once in t nor more than once in u. Equational tree automata have the nice property that the language $\mathcal{L}(\mathcal{A}/\mathcal{E})$ recognized an equational tree automata $\mathcal{A}/\mathcal{E}$ over a linear theory $\mathcal{E}$ can be viewed as the *equational closure* of the regular automaton $\mathcal{A}$, that is for each ground term $t \in T_\Sigma$,

$$t \in \mathcal{L}(\mathcal{A}/\mathcal{E}) \iff (\exists u \in T_\Sigma)\, u =_{\mathcal{E}} t \text{ and } u \in \mathcal{L}(\mathcal{A}).$$

The importance of this result is that it means that equational tree automata capture the *minimal* equational closure of regular tree languages when the associated equational theory $\mathcal{E}$ is linear. This result does not hold for *non-linear* theories [123]. However, by using an alternative formalization of equational tree automata with a slightly different semantics that captured the same basic idea, this connection was lifted to non-linear theories in [138].

Unfortunately, when one wants to perform Boolean operations on the equa-

tional closure of tree languages there are two problems: (1) the *equational closure* of a regular tree language is not in general a *regular tree language*; (2) equational tree languages are not closed under Boolean operations. Due to these limitations, in [76] we presented a framework called *Propositional Tree Automata* which recognize the minimal extension to equational tree languages that are closed under Boolean operations. However, propositional tree automata have their own limitation: the emptiness problem is in general undecidable for languages recognized by propositional tree automata.

Due to these inherent undecidability issues, most research on equational tree automata focuses on particular types of axioms and combinations of axioms such as *associativity* (A) $(x + y) + z = x + (y + z)$, *commutativity* (C) $x + y = y + x$, *unit* (U) $x + 0 = x$, and *idempotence* (I) $x + x = x$. For example, it is known that AC tree languages (equational tree languages over a theory where every symbol is associative and commutative or free) are closed under Boolean operations [125]. Likewise, it is known that equational tree languages over a theory with free symbols and an associative symbol are not closed under intersection or complementation [125].

My main contributions on tree automata in Chapter 3 are to develop effective decision procedures for solving the emptiness problem of propositional tree automata for several important theories. We call the emptiness problem for propositional tree automata the *propositional emptiness problem*. Our work has three main contributions:

- Our first result is a negative result that is nevertheless quite important. We show that intersection emptiness and propositional emptiness are *non-modular* problems. This means that even if intersection emptiness or propositional emptiness is decidable for disjoint theories $\mathcal{E}_1$ and $\mathcal{E}_2$, the same problem may be undecidable for the combined theory $\mathcal{E}_1 \cup \mathcal{E}_2$.

- Our second contribution is to develop a *semi-algorithm* which uses to machine learning techniques to solve the propositional emptiness problem for theories with combinations of free, A and AC symbols. This problem is undecidable in general, but our semi-algorithm has been implemented in the CETA library [69] and proven remarkably successful in practice. It is the basis for the implementation of our next generation Maude sufficient completeness checker discussed in Chapter 5.

- Our final result is to develop a decision procedure for solving the propositional emptiness problem for a large subclass of tree automata over theories with free, AC, and ACI theories. Both intersection and propositional emptiness are undecidable in general for this class of theories with these combinations of axioms, but our decidable subclass is large enough to solve two previously open problems.

1.3 Order-sorted equational unification

The goal of a unification algorithm is to generate a complete (preferably small) set of solutions which represent all of the solutions to a system of equations. In *equational unification*, we are given a theory $\mathcal{E}$ and a unification problem $\Gamma \equiv (t_1 = u_1 \wedge \cdots \wedge t_n = u_n)$. A *solution* to Γ is a substitution θ such that $t\theta =_{\mathcal{E}} u\theta$ for each equation $t = u$ in Γ. We call a set of solutions U *complete* if every solution to Γ is an instance of some solution in U. Unfortunately, a unification problem for an arbitrary theory $\mathcal{E}$ may not have a complete finite set of solutions U. However, there are many useful equation theories important in automated deduction for which each equational unification problem has a complete finite set of unifiers [7].

Unification is a fundamental operation in many other tools in automated deduction. This includes paramodulation-based theorem proving [119], computing critical pairs for confluence checking and completion [6], and solving reachability problems using narrowing [47]. Our main interest stems from unification's fundamental role in the Maude-NRL protocol analyzer [48, 49], which uses unification to identify vulnerabilities in security protocols.

In this application, the tool must solve unification problems over *order-sorted theories*. The sort information in the theory is used to restrict the tool to only consider well-formed states. The use of sort information is essential. If we were to drop the sort information by performing unsorted unification, then the tool can fail to successfully verify a protocol or find attacks in a feasible amount of time.

Unfortunately, the use of order-sorted equational unification poses a difficult implementation challenge for the tool. Writing an equational unification procedure is quite complex, so we would strongly prefer to reuse an existing unification procedure. However, the existing equational unification procedures that we are aware of perform unsorted unification and do not support order-sorted unification.

The solution, originally proposed by Meseguer et al. [112], is that under certain conditions, a simple *sort propagation* algorithm can be used to obtain a complete set of *order-sorted unifiers* from a complete set of *unsorted unifiers*. By implementing the sort propagation algorithm, we can use an existing unsorted equational unification procedure to generate a set of unsorted unifiers $\overline{U}$ and obtain a complete set of order-sorted equational unifiers U without much work — at least provided the theory $\mathcal{E}$ satisfies the conditions in [112].

For the Maude-NRL protocol analyzer, there was a problem which we did not discover until long after the algorithm had been implemented and being used extensively for free, AC, and ACU unification by calling the unsorted unification tool CiME [35]. The results in [112] were correct, but one of the conditions was too strong in a rather subtle way, and essentially meant that the technique is unsound for many theories with collapsing equations like identity. The problem

is largely theoretical in nature. However, it is theoretically possible for the approach to fail to return a complete set of unifiers for the theories we were using in the Maude-NRL protocol analyzer

In Chapter 4, we present our solution to this problem, which is to relax the conditions in [112] imposed on the equational theory, but impose an additional condition on the unsorted unifiers which the unsorted unification procedure is allowed to return. Existing unification algorithms and their implementations appear to generate unifiers that always satisfy this extra condition, but we do not rely on that. Instead, the extra condition is easy to check when parsing the unsorted unifiers back into Maude, and we will catch the violation and fail if the unsorted unifiers violate the condition. This means that our tool will never silently give an incomplete set of unifiers, and so far in our testing, we have never seen this extra check fail.

Our main contributions in Chapter 4 are as follows:

- We give a simple rule based algorithm for sort-propagation. The algorithm only consists of three confluent and terminating rules, and it should be possible to modularly compose these rules with other inference rules if desired. This is in contrast with the presentation in [112], where the algorithm was buried in a function definition appearing in the proof of Theorem 34, and not explicitly discussed much beyond that.

- We present a new correctness proof of the algorithm under more relaxed conditions. Relaxing the conditions on the proof is essential to proving the correctness of the approach in our applications with the Maude-NRL protocol analyzer.

- Finally, we show how the abstract technical conditions in the previous proof relate to concrete conditions on Maude specifications with any combination of free, commutative, AC, and ACU symbols. Each of the concrete conditions can either be checked automatically, or obtained by a theory transformation on the Maude specification that guarantees the condition is satisfied.

Although the techniques for order-sorted unification and tree automata are quite different, both are theoretical tools that are sometimes used in similar applications. For example, the unification procedure discussed here is used in the Maude NRL protocol analyzer while the CETA tree automata library [69] discussed previously has been added into the ACTAS tool [126] for protocol verification. Connections between unification techniques and tree automata is a fruitful area of research (e.g., [30, 31]).

1.4 Sufficient completeness checking

For automated reasoning techniques to be useful, one needs applications to apply them to. One such application which we discuss in this work is the *sufficient completeness problem* for equational specifications. Roughly speaking, a specification is *sufficiently complete* if enough equations have been specified so that each operation is fully defined on all relevant data. This concept was introduced by Guttag in [66], and has been heavily studied in the context of unsorted and unconditional specifications (e.g., [32, 89, 90, 120]).

Sufficient completeness is much less well-studied for logics such as membership equational logic (MEL) which have conditional equations and support partiality via sorts. There are several important issues in the case of sorted logics that do not occur in the unsorted case.

- The first issue is that operations may intentionally not be defined on all inputs, but only on the well-sorted inputs that an operator expects.
- The second issue is that the definition of the *data* may in fact depend on defined *functions* on that data. For example, in the powerlist specification which we will discuss later in Chapter 8, powerlists may be concatenated together to construct a new powerlist, but only when they have the same *length*. So in this case, the definition of a powerlist depends on an operation on powerlists.
- The third issue is that symbols may be overloaded, so that the same symbol may be a constructor on one domain and a defined symbol on another. The successor symbol is an example of this: it is a constructor on the natural numbers, but defined on the integers.

In this work, we introduce a notion of sufficient completeness for membership equational logic which naturally addresses these three issues. The basic idea is that instead of defining sufficient completeness in terms of constructor and defined *symbols*, sufficient completeness should be defined in terms of constructor and defined *membership axioms*. A theory is then sufficiently complete with respect to a a set of constructor memberships if there defined memberships are an inductive consequence of the equations together with the constructor memberships. Our definition is not the first definition of sufficient completeness for membership equational logic, since a notion of sufficient completeness for MEL specifications was proposed in [19]. However, that work only addressed the first of the three issues outlined above, and is inadequate for specifications like powerlists.

A definition of sufficient completeness is not necessarily useful by itself without a means for *checking* that a specification is sufficiently complete. Our second contribution is to extend existing approaches for checking sufficient completeness that only work on unsorted specifications to the case of membership equational

logic. Our first step in this work is to note that all existing approaches to sufficient completeness operate by reducing the sufficient completeness problem for specifications satisfying specific properties to a *ground reducibility* problem. In this work, we introduce a property called *defined reducibility* which generalizes ground reducibility to membership equational logic, and show how sufficient completeness and defined reducibility relate under a general set of conditions.

The reduction from sufficient completeness to defined reducibility is one step towards checking sufficient completeness, but by itself it is not enough. We also need tools for checking defined reducibility. Unfortunately, due to the potential for both conditional axioms and rewriting modulo axioms, checking defined reducibility in full generality is quite hard, and beyond the capabilities of any known automated techniques. In this work, we introduce *two different tools* for checking defined reducibility (and consequently sufficient completeness) on different large subclasses of Maude specifications.

Our first sufficient completeness checker first generates a set of proof obligations which together imply defined reducibility. The proof obligations are generated by *narrowing* the defined memberships against the left-hand sides of the equations in the module. The checker then passes those proof obligations to the Maude Inductive Theorem Prover. If the proof obligations are discharged, then the specification is sufficiently complete under quite general conditions.

Our second sufficient completeness checker casts sufficient completeness as a propositional emptiness problem for equational tree automata. This emptiness problem is then forwarded to our equational tree automata library CETA. If CETA shows that the language is empty, then the specification is sufficiently complete. If the language is not empty, then CETA will generate a counterexample showing that the specification is not sufficiently complete.

To summarize, this work makes the following contributions to sufficient completeness checking:

- We define a new notion of sufficient completeness for membership equational logic which naturally addresses many of the issues for sufficient completeness in sorted logics.

- We define a property called *defined reducibility* which generalizes a property for checking sufficient completeness in unsorted specifications called ground reducibility, and show how it relates to our new definition of sufficient completeness.

- Finally, we have developed two sufficient completeness checkers: the first checks conditional specifications by generating a set of proof obligations that are automatically forwarded to the Maude Inductive Theorem Prover (ITP); the second casts sufficient completeness as a equational tree automata decision problem for order-sorted specifications with rewriting modulo axioms.

1.5 Completeness in context-sensitive rewriting

Context-sensitive term rewriting [101, 103, 145] is an extension to term rewriting where certain completeness problems are important. In context-sensitive (CS) rewriting, the rewrite relation is restricted so that it may only occur in a restricted set of contexts, and not at any position in the term. The idea is that a *context-sensitive rewrite system* $\mathcal{R}$ over a signature with operators F is equipped with a mapping $\mu : F \to \mathcal{P}(\mathbb{N})$, called the *replacement map*. The rewrite relation $\to_{\mathcal{R},\mu}$ is defined so that rewriting may only occur in a subterm t_i of a term $f(t_1, \ldots, t_n)$ if $i \in \mu(f)$.

By restricting rewriting in this way and intelligently defining the replacement map μ, we can achieve at least two things. First, we can improve the efficiency of rewriting, so that terms are evaluated only as needed. This efficiency improvement can in fact infinitely speed up normalization by turning a non-terminating specification into a terminating CS specification. Second, we can use the replacement map to model infinite data structures in a lazy way. In this case, the replacement map is typically used to restrict evaluation of the *constructors* to restrict evaluation of data until it is consumed.

Although there has been significant progress in checking properties such as termination and e.g., [60, 102, 145] and confluence [103] of CS rewriting, one area that has not seen as much attention is the *mathematical semantics* of CS specifications. In this work, our first contribution is to define an algebra, called the *CS canonical term algebra* $\mathrm{Can}^{\mu}_{\mathcal{R}/A}$, whose elements are the canonical forms obtained by context sensitive rewriting modulo axioms A.

Our second contribution is to introduce three important CS *completeness problems*, namely:

1. *μ-canonical completeness*, which means that the CS canonical forms and canonical forms of unrestricted rewriting coincide;

2. *μ-semantic completeness*, which means that the CS canonical term algebra $\mathrm{Can}^{\mu}_{\mathcal{R}/A}$ is isomorphic to the initial algebra $T_{\mathcal{E}}$ of the underlying equational theory $\mathcal{E} = \mathcal{R} \cup A$; and

3. *μ-sufficient completeness*, which extends sufficient completeness to the context-sensitive case. The difference here is that it would be *too strong* to require that every term reduces to a constructor term: we only require that defined symbols are reducible if that appear in *replacing positions*.

In addition to defining the CS canonical term algebra and introducing these three completeness notions, we develop techniques for checking all three completeness problems via the equational tree automata techniques presented in Chapter 3. Furthermore, these techniques are implemented in a publicly available tool which can fully automatically check these properties in many Maude specifications.

1.6 Inductive theorem proving

We have already briefly mentioned how inductive theorem proving is useful in sufficient completeness checking of conditional specifications. However, inductive theorem proving has many other applications. For example, there is an enourmous body of work on using inductive theorem provers to formally prove mathematical theorems [143], correctness of hardware (e.g., [131]), compilers (e.g., [98]), and virtual machines (e.g., [99].

The Maude Inductive Theorem Prover (ITP) [27] is designed for proving that a first-order sentence holds in the initial model $T_{\mathcal{E}}$ of a membership equational logic specification $\mathcal{E}$. As with any such theorem prover, a sentence may be true in $T_{\mathcal{E}}$, but not provable with the ITP due to inherent incompleteness issues. However, inductive theorem provers can be extraordinarily useful in many applications, provided they satisfy several requirements, including:

- Support for generating good induction schemes;
- Strong core reasoning tactics and decision procedures;
- A good user interface for enabling the user to understand what the theorem prover is doing, and debug failed proof attempts; and
- A modular design that enables the prover to be combined with other tools.

The ITP already offers many features in support of these four criteria. However, there is always room for improvement. In this work, we have extended the ITP with additional features and improved existing features to better satisfy all four of the previously mentioned requirements. Chief among our new features are the following:

- We have added support for a new induction schema, called coverset induction [146], which uses the fact that a specification is terminating to generate induction schemes that are quite useful in practice.
- We have extended the core reasoning command `auto` to support extra equivalence relations in addition to equality, and automatically propagate facts implied by the transitivity of an equivalence relation.
- To improve the user interface, we have enhanced the ITP's parser to give better error messages for incorrect commands, and added additional commands for displaying information that may be relevant to the current goal.
- Finally, to make it easier to integrate other tools with the ITP, we have heavily refactored the ITP's source code to allow the ITP's command grammar to be extended in a modular way without changing any existing source code. We have also made significant improvements to the source code's documentation and readability. Finally, we have updated the ITP to work with the most recent version of Maude, version 2.3, so that it may interact with other tools supporting Maude 2.3.

1.7 Powerlist case study

The last chapter in this dissertation is an extensive case study using the ITP. In this case study, we have formally modeled Misra's powerlists [113] as a Maude specification in membership equational logic. We have also specified many of the operations on powerlists described in [113] including operations for permuting elements in the powerlist, the fast Fourier and inverse Fourier transformations, Batcher's merge sort algorithm. Finally, we have modeled the two types of adder circuits for powerlists over bits described in [1].

In addition to specifying the algorithms in Maude, we have formally proven many of the properties about powerlists and these algorithms in the Maude ITP. This has been done before in other theorem provers including ACL2 [55–57] and RRL [85, 87]. However, it is widely felt that the specifications and proofs in these tools lack the elegance of the hand proofs in [1, 96, 113]. A major goal of this work has been to specify and reason about powerlists in a natural way, so that the elegance in the hand proofs manifests itself in the machine proofs.

We have successfully proven the main correctness properties for the various permutation operations, Fast Fourier transform, and Batcher sort algorithms described in [113]. We have also modeled the two adder circuits described in [1], and shown that they both correctly implement addition modulo. Our experience with the proofs so far has generally been positive, but also suggests further ways to improve the ITP. Many of the improvements in the previous section were directly motivated by the powerlist case study. Already our proofs follow relatively close to Misra's hand proofs, however our experience in the case study suggests other avenues of future research that are not yet implemented which can improve the ITP even further.

1.8 Conclusions and a look ahead

In many ways, automated reasoning for program verification is a wonderful research topic. Few believe that the problem of buggy programs will be solved anytime soon, but the world has grown to depend on computer systems in so many different ways that even incremental steps can be quite useful. These systems are used in different contexts, and consequently offer an environment where many different techniques are potentially useful. For some highly safety-critical systems, it may be justifiable to employ relatively time-intensive techniques such as inductive theorem proving to obtain programs which provably satisfy their formal requirements. For less safety-critical systems, more highly automated techniques that prove more limited correctness properties such as sufficient completeness may be useful for catching errors early in a systems development to reduce the cost of catching errors later on.

This work spans several different areas of equational specification, term rewriting, tree automata, and inductive theorem proving. One of the key themes

in this work is to develop definitions and automated reasoning techniques for logics with sophisticated mechanisms for expressing partiality, lazy evaluation, and rewriting modulo axioms. These features make program specification easier, but often make automated analysis much harder. By developing better automated reasoning techniques, we hope to help the user to write programs in a natural way, but without giving up useful verification and debugging techniques.

One major direction for future research is to attempt to expand the boundaries of this work from equationally specified functional programs to programs that interact with the world in a much richer way. This requires different specification models such as actors [2], process calculi, or rewriting logic [109]. Nevertheless, even programs in these models often need to compute the value of functions between interactions, and it would be interesting to explore the problem of embedding some of the equational reasoning techniques discussed here into these more general classes of programs. In particular, for rewriting logic-based programs, this generalization seems very natural, since rewriting logic contains membership equational logic as a sublogic, and shares many of its rewriting-based techniques.

Chapter 2

Equational logic

Equational definitions are ubiquitous in computer science. Equations can be used both to define operations and properties of those operations. As a simple example, addition over the natural numbers formed from 0 and successor s may be defined with the two equations:

$$x + 0 = x \qquad\qquad x + s(y) = s(x + y)$$

Furthermore, important properties of addition like associativity and commutativity may be defined equationally:

$$\underbrace{(x + y) + z = x + (y + z)}_{\text{Associativity}} \qquad \underbrace{x + y = y + x}_{\text{Commutativity}}$$

Equational logic offers a simple yet expressive way of describing functions. It possesses both a simple proof theory for deduction as well as a clear model theory for defining semantics. Furthermore, in many applications, the equations $l = r$ that define an operation may be oriented left-to-right as rules $l \to r$, and interpreted by replacing instances of the left-hand side l with the corresponding instance of the right hand side. This approach to equational specification, called *term rewriting*, has been extensively studied by many authors, and constitutes a Turing complete model of computation.

One limitation with equational specifications is that operations are assumed to be *total*, while in practice, many operations are *partial*, and not defined on all inputs. For example, division by zero is typically left undefined in mathematics. Over the years, many different logics have been proposed supporting different degrees of partiality. The two approaches most relevant to this work are order-sorted logic [63, 64] and membership-equational logic [19, 111] (MEL). In both of these logics, *sorts* are associated to the operations in a specification and used to indicate when a term has a well-defined value. Of the two logics, membership-equational logic is much more expressive, so much so that it is undecidable in general whether a term in membership-equational logic has a given sort.

In this chapter, we will survey results in unsorted equational logic, order-sorted logic, and membership-equational logic that are relevant to the later chapters. We will also introduce basic concepts from term-rewriting, and show

how different term rewriting frameworks can be used to execute specifications in unsorted, order-sorted, and membership equational logic. In particular, we will focus on a major extension to term rewriting, called *rewriting modulo axioms*, in which some of the equations in a specification may be treated directly as equations, and not oriented into rules.

Rewriting modulo axioms is essential when the specification contains equations like commutativity $x + y = y + x$ which can not be oriented into a terminating rule. It is also important in applications involving data structures such as lists, sets, and multisets, because it is often the case that the specification will assume underlying equational properties such as associativity or commutativity about the data structure and its operations. Rewriting modulo axioms is also necessary in certain programming styles such as object-oriented modules in Full Maude [42], where objects can send messages to each other that are routed automatically by intelligent use of matching modulo associativity and commutativity.

This chapter is structured as follows: In the next section, we introduce the fundamental concepts in unsorted equational specification. In Section 2.2, we give basic definitions in (unsorted) term rewriting including rewriting modulo axioms. We describe order-sorted logic in Section 2.3, and order-sorted rewriting in Section 2.4. We discuss membership equational logic in Section 2.5, and finally in Section 2.6, we present a rewriting framework that supports the additional features in membership-equational logic called conditional equational rewrite-membership (CERM) systems.

2.1 Unsorted equational theories

We begin with a brief introduction to the terminology of equational specifications, and the notation used in this work. For a more comprehensive introduction to equational specification and term rewriting, see one of the many surveys or books (e.g., [6, 38–40, 122]).

Syntax. An *unsorted (functional) signature* Σ is just a *ranked alphabet*: that is a set of symbols Σ where each function symbol $f \in \Sigma$ has an associated arity $\text{arity}(f) \in \mathbb{N}$. We say that a function symbol $f \in \Sigma$ is a *constant* iff it has arity 0. We let X denote a fixed countably infinite set of *variables* distinct from the symbols in Σ. The set of terms formed from the symbols in Σ and the variables in X is denoted by $T_\Sigma(X)$. That is, $T_\Sigma(X)$ is the smallest set containing: (1) the variables in X, and (2) a term $f(t_1, \ldots, t_n)$ for each symbol $f \in \Sigma$ with arity n and terms $t_1, \ldots, t_n \in T_\Sigma(X)$. Note that the number of arguments n in $f(t_1, \ldots, t_n)$ is 0 if f is a constant. For each term $t \in T_\Sigma(X)$, we let $\text{vars}(t)$ denote the variables appearing in t, i.e.,

$$\text{vars}(x) = \{\, x \,\} \qquad \text{and} \qquad \text{vars}(f(t_1, \ldots, t_n)) = \text{vars}(t_1) \cup \cdots \cup \text{vars}(t_n).$$

Reflexitivity	$\dfrac{}{t =_{\mathcal{E}} t}$
Symmetry	$\dfrac{t =_{\mathcal{E}} u}{u =_{\mathcal{E}} t}$
Transitivity	$\dfrac{t =_{\mathcal{E}} u \qquad u =_{\mathcal{E}} v}{t =_{\mathcal{E}} v}$
Congruence	$\dfrac{t_1 =_{\mathcal{E}} u_1 \quad \ldots \quad t_n =_{\mathcal{E}} u_n}{f(t_1, \ldots, t_n) =_{\mathcal{E}} f(u_1, \ldots, u_n)}$
Replacement	$\dfrac{}{l\theta =_{\mathcal{E}} r\theta}$ with $l = r \in \mathcal{E}$

Figure 2.1: Inference system for the unsorted theory $\mathcal{E}$

If $\text{vars}(t) = \varnothing$, then we say a term t is *ground*, and we let T_Σ denote the set of ground terms. If each variable $x \in \text{vars}(t)$ appears at most once in t, then we say that t is *linear*.

Relative to a signature Σ with variables X, a *substitution* is a mapping $\theta : Y \to T_\Sigma(X)$ with a finite domain $Y \subseteq X$. For each term $t \in T_\Sigma(Y)$, we let $t\theta \in T_\Sigma(X)$ denote the term obtained by substituting each variable x appearing in t with $\theta(x)$, i.e.,

$$f(t_1, \ldots, t_n)\theta = f(t_1\theta, \ldots, t_n\theta) \qquad\qquad x\theta = \theta(x)$$

If every term in the right-hand side of a substitution θ is a ground term, then we say that θ is a *ground substitution*. A *context* C is a term containing a single instance of a special variable $\Box$. Given a term $t \in T_\Sigma(X)$, $C[t]$ denotes the term obtained by replacing $\Box$ with t.

Theories. An unsorted equational theory $\mathcal{E}$ over a signature Σ is a set of equations of the form $l = r$ for terms $l, r \in T_\Sigma(X)$. The essential characteristic of an equational theory $\mathcal{E}$ is not the specific equations in $\mathcal{E}$, but the equivalence relation $=_{\mathcal{E}} \subseteq T_\Sigma(X) \times T_\Sigma(X)$ generated by those equations. This relation $=_{\mathcal{E}}$ is the smallest equivalence relation containing the equations in $\mathcal{E}$ that is closed under substitutions and contexts. Closure under substitutions means that $t =_{\mathcal{E}} u$ implies $t\theta =_{\mathcal{E}} u\theta$ for all substitutions θ. Closure under contexts means that $t =_{\mathcal{E}} u$ implies that $C[t] =_{\mathcal{E}} C[u]$ for all contexts C. Alternatively, $=_{\mathcal{E}}$ may be defined using an inference system. It is not difficult to show that for each pair of terms $t, u \in T_\Sigma(X)$, $t =_{\mathcal{E}} u$ iff there is a proof using the inference rules in Figure 2.1.

Algebras. An *algebra* $\mathcal{A}$ of an unsorted signature Σ consists of a set A (called the *universe* or *carrier set*) together with a function $A_f : A^n \to A$ for each function symbol $f \in \Sigma$ with arity n. Given algebras $\mathcal{A}$ and $\mathcal{B}$ over the same signature Σ, a *homomorphism* $h : \mathcal{A} \to \mathcal{B}$ is a function from the universe of $\mathcal{A}$ to the universe of $\mathcal{B}$ that commutes with the functions. That is, for each function

$f \in \Sigma$ with arity n and elements $a_1, \ldots a_n \in A$,

$$h(A_f(a_1, \ldots, a_n)) = B_f(h(a_1), \ldots, h(a_n)).$$

The set of terms in $T_\Sigma(X)$ may be viewed as an algebra $T_{\Sigma(X)}$, called the *term algebra* generated by X, whose universe is the the terms in $T_\Sigma(X)$, and where for each function $f \in \Sigma$ with arity $n \in \mathbb{N}$, $T_{\Sigma(X),f}$ is the function

$$T_{\Sigma(X),f}(t_1 \ldots t_n) = f(t_1, \ldots, t_n).$$

In the same way, the set of ground terms T_Σ may be viewed as algebra. The algebra T_Σ has the property that for each algebra $\mathcal{A}$, there is a unique homomorphism $h_\mathcal{A} : T_\Sigma \to \mathcal{A}$ where for each term $f(t_1, \ldots, t_n)$,

$$h_\mathcal{A}(f(t_1, \ldots, t_n)) = A_f(h_\mathcal{A}(t_1), \ldots, h_\mathcal{A}(t_n)).$$

Due to this property, the algebra T_Σ is often referred to as an *initial algebra*, because it is an initial object in the category formed with Σ-algebras as objects and Σ-homomorphisms as arrows.

Relative to a set of variables Y and algebra $\mathcal{A}$, a *valuation* ν is a mapping from Y into the universe of $\mathcal{A}$. Additionally, the valuation $\nu : Y \to \mathcal{A}$ for a Σ-algebra $\mathcal{A}$ defines an interpretation of each term $t \in T_\Sigma(Y)$ in $\mathcal{A}$. Specifically, for each term $t \in T_\Sigma(Y)$, we let $t\nu$ denote the element in $\mathcal{A}$ defined by

$$f(t_1, \ldots, t_n)\nu = f(t_1\nu, \ldots, t_n\nu) \qquad\qquad x\nu = \nu(x)$$

It is worth observing that valuations into $T_\Sigma(X)$ are substitutions while valuations into T_Σ are ground substitutions. Furthermore, for each substitution $\theta : X \to T_\Sigma(Y)$, the application of a substitution $t\theta$ is just the interpretation of t under the valuation θ.

An algebra $\mathcal{A}$ *satisfies* an equation $t = u$, denoted $\mathcal{A} \models t = u$, iff for every valuation $\nu : \text{vars}(t) \cup \text{vars}(u) \to \mathcal{A}$, $t\nu = u\nu$. Furthermore, a Σ-algebra $\mathcal{A}$ is a *model* of an equational theory $\mathcal{E}$, denoted $\mathcal{A} \models \mathcal{E}$ iff all equations $t = u \in E$ hold in $\mathcal{A}$. An equational theory may have many models, and every equational theory has at least one model, denoted $T_\mathcal{E}$, whose universe contains equivalence classes of ground terms in T_Σ with respect to $=_\mathcal{E}$. For each ground term $t \in T_\Sigma$, we let $[t]_\mathcal{E}$ denote the set of all ground terms equivalent to t with respect to $=_\mathcal{E}$, that is,

$$[t]_\mathcal{E} = \{\, u \in T_\Sigma \mid t =_\mathcal{E} u \,\}.$$

When the theory $\mathcal{E}$ is clear from the context, we write $[t]$ for $[t]_\mathcal{E}$. The algebra $T_\mathcal{E}$ has the universe $T_\mathcal{E} = \{\, [t]_\mathcal{E} \mid t \in T_\Sigma \,\}$, and for each symbol $f \in \Sigma$ with arity n, $T_{\mathcal{E},f}$ is the function

$$T_{\mathcal{E},f} : [t_1], \ldots, [t_n] \mapsto [f(t_1, \ldots, t_n)].$$

It is not difficult to show that there is a unique homomorphism from $T_{\mathcal{E}}$ to each model of $\mathcal{E}$. Accordingly, $T_{\mathcal{E}}$ is an *initial algebra* in the category of $\mathcal{E}$-models and Σ-homomorphisms.

2.2 Rewriting modulo axioms

Equational theories provide a mathematical interpretation of what the operations mean, but they are not directly executable — in other words, they lack an *operational* interpretation. One classic approach for giving an operational interpretation is to treat each *equation* $t = u$ in a theory $\mathcal{E}$ as an *oriented rule* $t \to u$. Given a term $v \in T_\Sigma(X)$, one then *rewrites* v by replacing instances $C[t\theta]$ of the left hand side of a rule $t \to u$ with the corresponding right-hand side $C[u\theta]$. For many specifications, the equations can be easily oriented so that the rewriting process always terminates with a unique normal form $v\downarrow_{\mathcal{R}}$ for each term $v \in T_\Sigma(X)$.

Rewriting has proven fruitful and is the basis for many tools including: a variety of different rule-based programming languages, including ASF+SDF [21], CafeOBJ [51], Maude [28] and OBJ [52]. Additionally, rewriting is one of the fundamental techniques used by many inductive theorem proving tools including ACL2 [91], the Maude ITP [27], PVS [128] and RRL [88]. Rewriting has also been integrated into conventional programming languages with systems such as Tom [8].

It is well known (see [81, 130] for some prominent early work) that it is not always suitable to simply treat every equation as a rule. Instead, tools should provide *built in* support for specific equations like associativity and commutativity. This is the idea behind *rewriting modulo axioms*: an approach to rewriting where some of the equations are treated directly as equational axioms while the rest of the equations are treated as rules. In the rest of this section, we introduce basic notation for rewriting modulo axioms.

For a signature Σ, a *term rewrite system* (TRS) with rewriting modulo axioms is a pair written $\mathcal{R}/A$ where $\mathcal{R}$ is a set of *rewrite rules* of the form $l \to r$ where $l, r \in T_\Sigma(X)$ are terms such that $\mathrm{vars}(r) \subseteq \mathrm{vars}(l)$, and A is an equational theory for the signature Σ. A term $t \in T_\Sigma(X)$ rewrites to $u \in T_\Sigma(X)$ modulo a set of equational axioms A, denoted $t \to_{\mathcal{R}/A} u$, if there is a rule $l \to r \in \mathcal{R}$, context C, and substitution θ such that $t =_A C[l\theta]$ and $C[r\theta] =_A u$. In general, observe that when rewriting modulo the instance $l\theta$ of the left-hand side of the rule may not appear in t. We let $\to^+_{\mathcal{R}/A}$ denote the transitive closure of $\to_{\mathcal{R}/A}$, and write $t \to^*_{\mathcal{R}/A} u$ if $t \to^+_{\mathcal{R}/A} u$ or $t =_A u$. We write $t \downarrow_{\mathcal{R}/A} u$ if there is a term $v \in T_\Sigma(X)$ such that $t \to^*_{\mathcal{R}/A} v$ and $u \to^*_{\mathcal{R}/A} v$. A term t is *$\mathcal{R}/A$-irreducible* if it cannot be further rewritten. We sometimes refer to an $\mathcal{R}/A$-irreducible term t as being in *normal form*. We write $t \to^!_{\mathcal{R}/A} u$ if $t \to^*_{\mathcal{R}/A} u$ and u is $\mathcal{R}/A$-irreducible.

There are several important subclasses of rules in rewriting. A set of rules

$\mathcal{R}$ is *ground* iff all the terms appearing in rules in $\mathcal{R}$ are ground while $\mathcal{R}$ is *linear* iff all the terms in rules in $\mathcal{R}$ are linear. $\mathcal{R}$ is *left-linear* iff all the terms l appearing in the left-hand side of a rule $l \to r \in \mathcal{R}$ is linear.

There are also several properties of the rewrite relation $\to_{\mathcal{R}}^*$ that are often important. A rewrite system $\mathcal{R}/A$ is *weakly normalizing* iff for each term $t \in T_\Sigma(X)$, there is an $\mathcal{R}/A$-irreducible term $u \in T_\Sigma(X)$ such that $t \to_{\mathcal{R}/A}^* u$. $\mathcal{R}/A$ is *strongly normalizing* if $\to_{\mathcal{R}/A}^+$ is well-founded. A strongly normalizing rewrite system is often called *terminating*. We say that $\mathcal{R}/A$ is *confluent* iff for all terms $t, u, v \in T_\Sigma(X)$, $t \to_{\mathcal{R}/A}^* u$ and $t \to_{\mathcal{R}/A}^* v$ implies $u \downarrow_{\mathcal{R}/A} v$. We say that $\mathcal{R}/A$ is *ground confluent* when it is confluent when only considering ground terms. Sometimes we will say that $\mathcal{R}$ is confluent *modulo* A instead of saying that $\mathcal{R}/A$ is confluent.

If $\mathcal{R}/A$ is terminating and ground confluent, then for all ground terms $t \in T_\Sigma$, there is a $\mathcal{R}/A$-irreducible term $t \downarrow_{\mathcal{R}/A} \in T_\Sigma$ that is unique up to $=_A$. When $\mathcal{R}/A$ is terminating and ground confluent, we let $\mathrm{Can}_{\mathcal{R}/A}$ denote the canonical term algebra whose universe is the set of A-equivalence classes of $\mathcal{R}/A$-irreducible ground terms,

$$\mathrm{Can}_{\mathcal{R}/A} = \{\, [t]_A \mid t \in T_\Sigma \text{ is } \mathcal{R}/A\text{-irreducible.} \,\}$$

and for each symbol $f \in \Sigma$ with arity n, $\mathrm{Can}_{\mathcal{R}/A,f}$ is the function

$$\mathrm{Can}_{\mathcal{R}/A,f} : [t_1], \ldots, [t_n] \mapsto [f(t_1, \ldots, t_n) \downarrow_{\mathcal{R}/A}].$$

We let $\mathcal{E} = \mathcal{R} \cup A$ denote the equational theory extending A with an additional equation $l = r$ for each rule in $\mathcal{R}$. We call this the *underlying equational theory* of $\mathcal{R}/A$. Under the assumption that $\mathcal{R}/A$ is terminating and ground confluent, it is not difficult to show that the algebra $\mathrm{Can}_{\mathcal{R}/A}$ is isomorphic to the initial algebra $T_\mathcal{E}$ with the mapping

$$h : [t]_A \in \mathrm{Can}_{\mathcal{R}/A} \mapsto [t]_\mathcal{E} \in T_\mathcal{E}.$$

If matching modulo A is decidable and $\mathcal{R}/A$ is confluent and terminating, $\mathrm{Can}_{\mathcal{R}/A}$ provides a *computational interpretation* of the algebra $T_\mathcal{E}$, since each equivalence class $[t] \in T_\mathcal{E}$ is represented by the equivalence class of the normal form $[t \downarrow_{\mathcal{R}/A}] \in T_A$ obtained by rewriting.

2.3 Order-sorted logic

In formalizing different systems, it has been found quite useful to extend the basic unsorted rewrite theories to support types. Unsorted specifications are *total* and allow any term to be given as an argument to any function in the specification, yet functions in mathematics and software are often partial, and only defined on a subset of the inputs. A basic example of this is division over

the rationals, where the divisor may not be zero. To deal with partial functions, Goguen proposed order-sorted logic, a logic where sorts may be partially ordered with larger sorts containing all of the elements in smaller sorts, and operators may be overloaded. Order-sorted logic is an extension of many-sorted logic in which a partial order $\leq$, called the *subsort ordering* is associated to the sorts. The subsort ordering builds a notion of subtype and supertype into the logic.

Before diving into the details, we present three order-sorted theories in Figure 2.2 using Maude syntax. The module `BOOL` introduces a single sort `Bool` and constants `true` and `false` representing true and false respectively. The `ctor` attribute is used to indicate that `true` and `false` are *constructors* which can be used to construct new data. Operators used to define operations on data should not be labeled with `ctor`. This distinction is the subject of Chapter 5, and can be ignored for now since it does not affect the mathematical semantics described in this chapter.

The module `NAT` extends `BOOL` by adding an additional sort `Nat` for defining the natural numbers with zero denoted by the constant `0`, and successor denoted by the operator `s`. This module introduces operations for addition `+` and less than or equal $\leq$, and defines those operations with equations. Finally, the module `INT` illustrates two essential features of order-sorted logic: the use of subsorts, and the ability to extend operator definitions to large domains via *subsort overloading*. The module imports both the `Bool` and `Nat` sorts and related operators, and defines an additional sort `Int` for representing the integers. To indicate that every natural number is an integer, `Nat` is declared to be a subsort of `Int`. The module introduces the operator `p` for the predecessor operation, and adds equations so that predecessor and successor cancel out. We use the `ctor` attribute on predecessor, but not on successor, because predecessor is a constructor on negative integers while successor was already previously defined as a constructor on the natural numbers and is defined on all the integers other than the natural numbers. Finally, the module extends the addition and $\leq$ operation to the integers with overloaded operation declarations, and defines additional equations for handling predecessor.

As illustrated by the `INT` example, a key feature of order-sorted logic is that the same symbol may be overloaded with different sorts associated to the domain and range. There are two distinct types of overloading that may occur in order-sorted logic: ad-hoc overloading and subsort overloading. Ad-hoc overloading just means that the same symbol may be used for unrelated operations, while subsort overloading means that an operation may be defined over different related sorts. An example of ad-hoc overloading would be using the same symbol + for addition over the natural numbers and string concatenation. An example of subsort overloading would be using + for addition over both natural numbers and integers. In subsort overloading, the operation must be consistent when applied to the same elements. When given natural numbers as arguments, addition over the integers must yield the same result as the operation for addition

```
fmod BOOL is
  sort Bool .
  ops true false : -> Bool [ctor].
endfm

fmod NAT is protecting BOOL .
  sort Nat .
  op 0 : -> Nat [ctor].
  op s : Nat -> Nat [ctor].

  var M N : Nat .

  op _+_ : Nat Nat -> Nat .
  eq M + s(N) = s(M + N) .
  eq M + 0 = M .

  op _<=_ : Nat Nat -> Bool .
  eq 0 <= N = true .
  eq s(N) <= 0 = false .
  eq s(M) <= s(N) = M <= N .
endfm

fmod INT is protecting NAT .
  sort Int .
  subsort Nat < Int .

  op s : Int -> Int .
  op p : Int -> Int [ctor].

  var I J : Int .

  eq p(s(I)) = I .
  eq s(p(I)) = I .

  op _+_ : Int Int -> Int .
  eq p(I) + J = p(I + J) .
  eq I + p(J) = p(I + J) .

  op _<=_ : Int Int -> Int .
  eq p(I) <= J = I <= s(J) .
  eq I <= p(J) = s(I) <= J .
endfm
```

Figure 2.2: Order-sorted example

over the naturals.

There are different formalizations of order-sorted logic with support for different degrees of ad-hoc and subsort overloading (see [63, 111] for surveys). Ad-hoc overloading can be eliminated by annotating symbols to appropriately disambiguate them, while subsort overloading is an essential feature of order-sorted logic. Our definition of an order-sorted signature below rules out ad-hoc overloading in order to simplify later results.

Definition 2.3.1. *An* order-sorted signature $\Sigma = (S, F, \leq)$ *is a tuple such that:*

- $(S, \leq)$ *is a partial order called the* subsort ordering *which generates the equivalence relation* $\equiv_{\leq} \subseteq S \times S$*, denoting the reflexive, symmetric and transitive closure of* $\leq$*.*
- $F = \{ F_{w,s} \}_{(w,s) \in S^* \times S}$ *is a family of* operators *in which for each overloaded symbol* $f \in F_{s_1 \ldots s_m, s} \cap F_{s'_1 \ldots s'_n, s'}$*, we have that* $m = n$*,* $s \equiv_{\leq} s'$*, and* $s_i \equiv_{\leq} s'_i$ *for all indices* $i \in [1, n]$*.*

For each sort $s \in S$, the *connected component* of s is the equivalence class $[s] \in S/\equiv_{\leq}$. In the `INT` example in Figure 2.2, `Nat` and `Int` belong to the same connected component, while `Bool` belongs to a different connected component containing no other sorts. The restrictions on operators are given so that the sorts associated to the output and each argument position belong to the same connected component.

In order-sorted logic, the *variables* of a signature $\Sigma = (S, F, \leq)$ are a S-sorted family $X = \{ X_s \}_{s \in S}$ where each set X_s is countably infinite and disjoint from both the operators in F and the other sets of variables. If a variable x is in X_s, we say that x *has sort* s, and will sometimes write x_s to indicate that x has sort s. To simplify later statements, we overload the notation for families so that a family such as the variables $X = \{ X_s \}_{s \in S}$ also denotes the set $X = \bigcup_{s \in S} X_s$. Finally, when the signature $\Sigma = (S, F, \leq)$ is clear, we write $f : s_1 \ldots s_n \to s$ for $f \in F_{s_1 \ldots s_n, s}$.

For a signature $\Sigma = (S, F, \leq)$, the *terms* of Σ form a S-indexed family $T_\Sigma(X) = \{ T_\Sigma(X)_s \}_{s \in S}$ where for each $s \in S$, $T_\Sigma(X)_s$ denotes the terms with a sort $s' \leq s$ formed by the operators in Σ and variables in each sort $X_{s'}$ with $s' \leq s$. The *ground terms* of Σ form a S-indexed family $T_\Sigma = \{ T_{\Sigma,s} \}_{s \in S}$ where $T_{\Sigma,s}$ denotes the terms formed solely from the operators in F and not containing any variables in X. An *order-sorted substitution* is a mapping $\theta : Y \to T_\Sigma(X)$ where Y is a finite subset of X and with the condition that $\theta(x_s) \in T_\Sigma(X)_s$ for each variable $x_s \in Y$.

Definition 2.3.2. *A* theory *for an order sorted signature* $\Sigma = (S, F, \leq)$ *is a set* $\mathcal{E}$ *of equations of the form* $l = r$ *with* $l, r \in T_\Sigma(X)$ *having sorts in the same equivalence class in* $S/\equiv_{\leq}$*.*

There are various inference systems in order-sorted logic for deriving equations of the form $t = u$. In this work, we reuse the inference system in Figure 2.1

for the unsorted case, but with the additional restriction that terms are required to be well-sorted and substitutions must be order-sorted substitutions. For the unconditional order-sorted theories we consider, our inference system has the same semantics as the sound and complete inference system presented in [111].

Our definitions for order-sorted algebras and homomorphisms is from [111].

Definition 2.3.3. *An* Σ-algebra $\mathcal{A}$ *for an order-sorted signature* $\Sigma = (S, F, \leq)$ *consists of:*

- *a set* A_s *for each sort* $s \in S$ *such that* $A_s \subseteq A_{s'}$ *for* $s \leq s'$*;*
- *a function* $A_{f:w\to s} : A_w \to A_s$ *for each symbol* $f \in F_{w,s}$ *where* $w = s_1 \ldots s_n$, $A_w = A_{s_1} \times \ldots A_{s_n}$, *and for each symbol* $f \in F_{w,s} \cap F_{w',s'}$ *and common arguments* $\overline{a} \in A_w \cap A_{w'}$,

$$A_{f:w\to s}(\overline{a}) = A_{f:w'\to s'}(\overline{a}).$$

For an order sorted signature Σ, a Σ-*homomorphism* $h : A \to B$ is a family of functions $h = \{h_s : A_s \to B_s\}_{s\in S}$ such that:

- For sorts $s \equiv_{\leq} s'$ and common elements $a \in A_s \cap A_{s'}$,

$$h_s(a) = h_{s'}(a).$$

- For each symbol $f \in F_{s_1\ldots s_n,s}$ and elements $a_1 \in A_{s_1}, \ldots, a_n \in A_{s_n}$,

$$h_s(A_f(a_1, \ldots, a_n)) = B_f(h_{s_1}(a_1), \ldots, h_{s_n}(a_n)).$$

Given an order-sorted theory $\mathcal{E}$ over Σ, an $\mathcal{E}$-algebra is a Σ-algebra satisfying the equations in E. We let T_Σ denote the ground term algebra for Σ, and $T_\mathcal{E}$ denote the $\mathcal{E}$-algebra such that $T_{\mathcal{E},s} = \{[t]_\mathcal{E} \mid t \in T_{\Sigma,s}\}$ for each sort $s \in S$, where $[t]_\mathcal{E}$ denotes the set of ground terms equivalent to t modulo $\mathcal{E}$. Both T_Σ and $T_\mathcal{E}$ are *initial* for the categories of Σ-algebras and $\mathcal{E}$-algebras respectively, so there is a unique homomorphism from T_Σ to any Σ-algebra, and a unique homomorphism from $T_\mathcal{E}$ to any $\mathcal{E}$-algebra. For a Σ-algebra A and ground term $t \in T_\Sigma$, we let $A(t)$ denote the value of t in the unique homomorphism $A : T_\Sigma \to A$, i.e., $A(f(t_1, \ldots, t_n)) = A_f(A(t_1), \ldots, A(t_n))$.

Underlying unsorted theories. Our later results will often depend on the ability to discard the sort information in an order-sorted theory $\mathcal{E}$ and reason in the *underlying unsorted theory* $\overline{\mathcal{E}}$, defined below, that contains the same equations as $\mathcal{E}$, but without reference to specific sorts.

Definition 2.3.4. *Given an order-sorted theory* $\mathcal{E}$ *over a signature* $\Sigma = (S, F, \leq$ *), we let* $\overline{\mathcal{E}}$ *denotes the* underlying unsorted theory $\overline{\mathcal{E}}$ *over an unsorted signature* $\overline{\Sigma}$ *such that*

- $\overline{\Sigma}$ *is an unsorted signature containing an operator* f *with arity* n *iff there is an operator* $f \in F_{s_1 \ldots s_n, s}$ *for some sorts* $s_1, \ldots, s_n, s \in S$.
- $\overline{\mathcal{E}}$ *contains the equations in* $\mathcal{E}$ *with sort information dropped on variables.*

Furthermore, we let $\overline{X} = \bigcup_{s \in S} X_s$ *be the variables associated to* $\overline{\mathcal{E}}$.

This definition will be used in the next section for defining order-sorted rewriting modulo axioms. It will also be used in Chapters 3–6 to simplify different decision problems involving *order-sorted* theories to problems involving only *unsorted* theories.

2.4 Order-sorted rewriting modulo axioms

As in the unsorted case, an order-sorted theory $\mathcal{E}$ may be interpreted operationally by orienting the equations $l = r \in \mathcal{E}$ as rewrite rules $l \to r$ and simplifying expressions from left to right. As in the unsorted case one may perform rewriting modulo axioms such as associativity and commutativity in order-sorted theories. However when performing order-sorted rewriting modulo axioms, the variables in the axioms are typically constrained so that the equational matching algorithm can ignore the sort information. We call axioms that satisfy this constraint *sort-independent* and define the property formally below:

Definition 2.4.1. *An equational theory* $\mathcal{E}$ *over* Σ *is* sort-independent *iff for all well-sorted terms* $t, u \in T_\Sigma(X)$,

$$t =_{\overline{\mathcal{E}}} u \Rightarrow t =_{\mathcal{E}} u.$$

where $\overline{\mathcal{E}}$ *is the underlying unsorted theory as described in Definition 2.3.4.*

As an example of how sort independence is used in practice, we briefly discuss how rewriting modulo axioms is performed in the rewriting engine of the Maude programming language [28]. Maude supports order-sorted rewriting modulo any combination of associativity, commutativity, and identity. However, Maude modules are syntactically required to satisfy certain syntactic properties required to guarantee sort independence. In this section, we call signatures satisfying these syntactic restrictions *Maude compatible*, and formalize the requirements below:

Definition 2.4.2. *An order-sorted signature* $\Sigma = (S, F, \leq)$ *is* Maude compatible *iff each connected component* $[s] \in S/\equiv_{\leq}$ *has a unique maximal sort called the* kind $k_s \in [s]$ *such that*

1. $s' \leq k_s$ *for all* $s' \in [s]$, *and*
2. *for each operator declaration* $f : s_1 \ldots s_n \to s$ *in* F, F *also contains the declaration* $f : k_{s_1} \ldots k_{s_n} \to k_s$.

We call a set of equational axioms A *maximal* when each variable x appearing in an equation $l = r \in Ax$ has a maximal sort k_s. We call a theory A *non-trivial* if it does not contain an equation with the form $x = y$. All of the theories that Maude supports rewriting modulo are both maximal and non-trivial. Thus we are able to show the following result for axioms A supported by Maude.

Proposition 2.4.3. *If A is a maximal non-trivial order-sorted theory over a Maude compatible signature Σ, then A is* sort-independent*.*

Proof. To show that A is sort-independent, we must show that for each pair of well-sorted terms $t, u \in T_\Sigma(X)$

$$t =_{\overline{\mathcal{E}}} u \Rightarrow t =_{\mathcal{E}} u.$$

The proof of this is by induction on the inferences used to show that $t =_{\overline{\mathcal{E}}} u$. Since t and u are well-sorted terms and ad-hoc overloading is disallowed, the different cases turn out to be quite straightforward. The only interesting case is the use of Replacement

$$\frac{}{l\overline{\theta} =_{\overline{\mathcal{E}}} r\overline{\theta}}$$

where $l = r$ is an equation in both $\overline{\mathcal{E}}$ and $\mathcal{E}$. The substitution $\overline{\theta}$ is an unsorted substitution. However, as each variable $x_k \in \mathrm{vars}(l) \cup \mathrm{vars}(r)$, has a maximal sort k, both $l\overline{\theta}$ and $r\overline{\theta}$ are well-sorted. We can use the definition of a Maude compatible signature to show by induction on l and r respectively that $\overline{\theta}(x_k) \in T_\Sigma(X)_k$ for all variables $x_k \in \mathrm{vars}(l)$ and $x_k \in \mathrm{vars}(r)$ respectively. It follows that $\overline{\theta}$ is also an order-sorted substitution, and therefore $l\overline{\theta} =_{\mathcal{E}} r\overline{\theta}$. □

This digression into Maude compatible signatures and axioms was merely to illustrate a major way that sort-independence is achieved in practice. For this thesis, we will state most results in terms of the sort-independence property rather than the particular way that sort-independence is achieved in Maude. The sort-independence property is assumed in our definition of order-sorted term rewrite systems below:

Definition 2.4.4. *An* order-sorted term rewrite system *for a signature $\Sigma = (S, F, \leq)$ is a pair $\mathcal{R}/A$ such that*

- *A is a set of sort-independent Σ-equations, and*
- *$\mathcal{R}$ is a set of rewrite rules of the form $l \to r$ with $l, r \in T_\Sigma(X)_s$ for some $s \in S$ and* $\mathrm{vars}(r) \subseteq \mathrm{vars}(l)$*.*

As in the unsorted case, each order-sorted TRS $\mathcal{R}/A$ defines a rewrite relation $\to_{\mathcal{R}/A}$. In this case, rewriting occurs on well-sorted terms, and given terms $t, u \in T_\Sigma(X)$, we write $t \to_{\mathcal{R}/A} u$ iff there exists a context C, order-sorted substitution θ and rule $l \to r \in \mathcal{R}$ such that $t =_A C[l\theta]$ and $C[r\theta] = u$. In addition, properties of unsorted term rewrite systems such as weak-normalization,

```
fmod NAT-LIST is protecting NAT .
  sort NeList List .
  subsorts Nat < NeList < List .

  op nil : -> List [ctor].
  op __ : NeList NeList -> NeList [ctor assoc id: nil].
  op __ : List List -> List [assoc id: nil].

  var N : Nat .   var L : List .

  op head : NeList -> Nat .
  eq head(N L) = N .
  op end : NeList -> Nat .
  eq end(L N) = N .

  op reverse : List -> List .
  eq reverse(N L) = reverse(L) N .
  eq reverse(nil) = nil .
endfm
```

Figure 2.3: Rewriting modulo axioms example

termination and confluence can be extended to order-sorted term rewrite systems in an obvious way. Additionally, it is often important that if a term t has a particular sort s, i.e., $t \in T_\Sigma(X)_s$, then any term it rewrites to can be further rewritten to a term with sort s. We call this property *sort-preservation*. Formally, $\mathcal{R}/A$ is *sort-preserving* if for each sort $s \in S$, term $t \in T_\Sigma(X)_s$, and $u \in T_\Sigma(X)$, if $t \rightarrow^*_{\mathcal{R}/A} u$ then, there exists a $v \in T_\Sigma(X)$ such that $u \rightarrow^*_{\mathcal{R}/A} v$ and $v \in T_\Sigma(X)_s$.

As an example of an order-sorted specification that uses rewriting modulo axioms, we present in Figure 2.3 a specification of lists of natural numbers with an associative append operator. In this specification, the operators consist of those imported from the natural numbers module, along with a constant `nil` for the empty list, and an append operator `__`. The operator attributes `assoc` and `id: nil` define the axioms in A. Specifically, A contains the following axioms:

$$(xy)z = x(yz) \qquad x\,\texttt{nil} = x \qquad \texttt{nil}\,x = x$$

The equation declarations define the rules in R. The append operator `__` is subsort overloaded to indicate that it is yields a non-empty list when given non-empty lists as arguments, and a list more generally. The `ctor` attribute is used to indicate that certain declarations are used to *construct* data, and as mentioned previously this is explained in much more detail in Chapter 5.

With a traditional description of lists, one would use a `cons` operator, and would need to express `append` and `end` as recursive operations that walked down the structure of a list. By rewriting modulo associativity, we are able

to treat `append` directly as a constructor, and define `end` with a single non-recursive equation. Thanks to Maude's support for efficient rewriting modulo associativity, the above specification will run faster on large lists than one using `cons`.

2.5 Membership equational logic

Although order-sorted logic can encode many partial operations, it is still fairly limited in the degree of partiality it supports. Order-sorted logic can not, for example, express the idea that the sum of two vectors is only defined when the vectors have an equal length. To deal with more general notions of partiality, one can use *membership equational logic (MEL)* [111]. In MEL, atomic predicates are either equations $t = u$ or memberships $t : s$ which can be read as saying t has sort s, where t having a sort is equivalent to t being *defined*. In addition, MEL allows the axioms to be conditional Horn clauses over equations and memberships. For example, the axiom $x + y$: vector **if** $\text{len}(x) = \text{len}(y)$ might be used in a specification to say that the sum of two vectors x and y is a vector if the lengths are equal. MEL has a proof theory that is sound and complete with respect to a model theory [19], and can be used to encode order-sorted logic [111].

Membership equational logic is essentially Horn logic with equality and unary predicates. The logic has two levels of typing: *kinds* type operator declarations in a signature, and *sorts* type terms using membership axioms in a particular theory. We review the basic terminology of MEL below (see [19] or [111] for a more comprehensive introduction).

Definition 2.5.1. *A* membership equational logic *(MEL)* signature *is a triple* $\Sigma = (K, F, S)$*, in which:*

- K *is a set of* kinds*;*
- $F = \{F_{\overline{k},k}\}_{(\overline{k},k)\in K^*\times K}$ *is a* K*-kinded family of function symbols such that* $F_{\overline{k},k}$ *and* $F_{\overline{k},k'}$ *are disjoint for distinct* $k, k' \in K$*; and*
- $S = \{S_k\}_{k\in K}$ *is a disjoint* K*-kinded family of sets of* sorts*.*

In membership equational logic, the *variables* are typed by the kinds. Specifically, each signature $\Sigma = (K, F, S)$ is equipped with a K-index family of variables $X = \{ X_k \}_{k\in K}$ which are pairwise disjoint and also distinct from constant in F. When the variables X are clear from the context, we write x_k to denote that x is a variable with kind k, i.e., $x \in X_k$. The *terms* of a MEL signature Σ are a K-indexed family $T_\Sigma(X) = \{ T_\Sigma(X)_k \}_{k\in K}$ where each set $T_\Sigma(X)_k$ denotes the set of well-kinded terms with kind k formed from the function symbols in F and variables in X. The variables of a term $t \in T_\Sigma(X)$ are denoted by $\text{vars}(t)$. We denote the *ground terms* by $T_\Sigma = \{ T_{\Sigma,k} \}$ where $T_{\Sigma,k}$ denotes the

Reflexitivity	$\dfrac{}{t = t}$
Symmetry	$\dfrac{t = u}{u = t}$
Transitivity	$\dfrac{t = u \qquad u = v}{t = v}$
Congruence	$\dfrac{t_1 = u_1 \quad \ldots \quad t_n = u_n}{f(t_1, \ldots, t_n) = f(u_1, \ldots, u_n)}$
Replacement	$\dfrac{\alpha_1\theta \quad \ldots \quad \alpha_n\theta}{l\theta = r\theta}$ with $l = r$ **if** $\alpha_1 \wedge \cdots \wedge \alpha_n$ in $\mathcal{E}$
Subject Reduction	$\dfrac{t = u \quad u : s}{t : s}$
Membership	$\dfrac{\alpha_1\theta \quad \ldots \quad \alpha_n\theta}{t\theta : s}$ with $t : s$ **if** $\alpha_1 \wedge \cdots \wedge \alpha_n$ in $\mathcal{E}$

Figure 2.4: Inference system for MEL theory $\mathcal{E}$

set of all terms of kind k that do not contain variables. A MEL Σ-*substitution* is a mapping $\theta : Y \to T_\Sigma(X)$ with a finite domain $Y \subseteq X$ and the condition that $\theta(x_k) \in T_\Sigma(X)_k$ for each variable $x_k \in Y$.

A Σ-*equation* is a formula $t = u$ with $t, u \in T_\Sigma(X)_k$ for some $k \in K$. A Σ-*membership* is a formula $t : s$ with $t \in T_\Sigma(X)_k$ and $s \in S_k$. Σ-*sentences* are universally quantified Horn clauses of the form $(\forall Y)\ \alpha$ **if** $\alpha_1 \wedge \cdots \wedge \alpha_n$ where α and α_i $(i \in [1, n])$ are either Σ-equations or Σ-memberships with variables in Y. We will typically leave the variables Y implicit. If α is a Σ-equation, the sentence is a *conditional equation*; if α is a Σ-membership, the sentence is a *conditional membership*. A MEL theory over a signature Σ is a set of (possibly conditional) equations and memberships. As explained in [19], there is a sound and complete inference system to derive all valid atomic formulas of a MEL theory. We reproduce this inference system in Figure 2.4, and write $\mathcal{E} \vdash \alpha$ if α is an atomic formula with the form $t = u$ or $t : s$ which can be derived using the inference rules in the figure. For consistency with the unsorted notation, we write $t =_\mathcal{E} u$ iff $\mathcal{E} \vdash t = u$, and let $[t]_\mathcal{E}$ denote the set of all ground terms equivalent to a term $t \in T_{\Sigma,k}$ by the equations in $\mathcal{E}$, i.e.,

$$[t]_\mathcal{E} = \{\, u \in T_{\Sigma,k} \mid \mathcal{A} \vdash t = u \,\}.$$

Algebras in membership equational logic are defined as follows:

Definition 2.5.2. *An* algebra $\mathcal{A}$ *for a signature* $\Sigma = (K, F, S)$ *consists of*

- *a carrier set* $\mathcal{A}_k$ *for each kind* $k \in K$*,*

- *a function* $\mathcal{A}_f : \mathcal{A}_{k_1} \times \cdots \times \mathcal{A}_{k_n} \to \mathcal{A}_k$ *for each symbol* $f \in F_{k_1 \ldots k_n, k}$, *and*
- *a set* $\mathcal{A}_s \subseteq \mathcal{A}_k$ *for each sort* $s \in S_k$.

Furthermore, a Σ-*homomorphism* $h : \mathcal{A} \to \mathcal{B}$ from a Σ-algebra $\mathcal{A}$ to Σ-algebra $\mathcal{B}$ is a K-indexed family of mappings $h = \{ h_k \}_{k \in K}$ where each mapping is a function $h_k : \mathcal{A}_k \to \mathcal{B}_k$ such that:

- For each symbol $f \in F_{k_1 \ldots k_n, k}$,

$$h_k(\mathcal{A}_f(a_1, \ldots, a_n)) = \mathcal{B}_f(h_{k_1}(a_1), \ldots, h_{k_n}(a_n)).$$

- For each sort $s \in S_k$ and element $a \in \mathcal{A}_s$, $h_k(a) \in \mathcal{B}_s$.

Given a set of variables $Y \subseteq X$, a *valuation* in MEL is a mapping $\nu : Y \to \mathcal{A}$ such that for each variable $x_k \in Y$, $\nu(x_k) \in \mathcal{A}_k$. As in the unsorted case, valuations can be extended to interpret each term $t \in T_\Sigma(Y)$ For each term $t \in T_\Sigma(Y)$, we let $t\nu$ denote the element in $\mathcal{A}$ defined by

$$f(t_1, \ldots, t_n)\nu = f(t_1\nu, \ldots, t_n\nu) \qquad\qquad x\nu = \nu(x)$$

Each MEL atomic formula α has the form $t = u$ or $t : s$. We write $\mathcal{A}, \nu \vdash t = u$ iff $t\nu = u\nu$, and $\mathcal{A}, \nu \vdash t : s$ iff $t\nu \in \mathcal{A}_s$. A Σ-algebra $\mathcal{A}$ is a *model* of a MEL Σ-theory $\mathcal{E}$ iff for each clause α **if** $\alpha_1 \wedge \cdots \wedge \alpha_n$ in $\mathcal{E}$,

$$\mathcal{A}, \nu \models \alpha_1 \wedge \cdots \wedge \mathcal{A}, \nu \models \alpha_n \implies \mathcal{A}, \nu \models \alpha.$$

For a given MEL theory $\mathcal{E}$, we can define the initial algebra $T_\mathcal{E}$ as the algebra such that:

- For each kind $k \in K$, $T_{\mathcal{E},k} = \{ [t]_\mathcal{E} \mid t \in T_{\Sigma,k} \}$.
- For each symbol $f : k_1 \ldots k_n \to k$, $T_{\mathcal{E},f}$ is the function

$$T_{\mathcal{E},f} : [t_1] \ldots [t_n] \mapsto [f(t_1, \ldots, t_n)].$$

- For each sort $s_k \in S$, $T_{\mathcal{E},s} \subseteq T_{\mathcal{E},k}$ is the set

$$T_{\mathcal{E},s} = \{ [t] \in T_{\mathcal{E},k} \mid \mathcal{E} \vdash t : s \}.$$

2.6 Conditional rewrite-membership systems

Executing MEL specifications requires more general rewriting techniques to deal with the memberships and conditions supported by the logic. This was done in [19] by introducing a rewriting framework called conditional rewriting/membership (CRM) systems. In this work, equations were interpreted in two different ways. An equation $l = r$ appearing in the consequent of a rule

```
fmod POWERLIST is protecting NAT .
  sort Pow .
  op [_] : Nat -> Pow [ctor] .
  op _|_ : [Pow] [Pow] -> [Pow] .

  vars M N : Nat .
  vars P Q R S : Pow .

  op len : Pow -> Nat .
  eq len(P | Q) = len(P) + len(Q) .
  eq len([N]) = 1 .

  cmb (P | Q) : Pow if len(P) = len(Q) .

  op _x_ : [Pow] [Pow] -> [Pow] .
  cmb (P x Q) : Pow if len(P) = len(Q) [metadata "dfn"].
  eq (P | Q) x (R | S) = (P x R) | (Q x S) .
  eq [M] x [N] = [M] | [N] .
endfm
```

Figure 2.5: Powerlist example

$l = r$ **if** $\alpha_1 \wedge \ldots \alpha_n \in \mathcal{R}$ is interpreted as a rewrite rule $l \to r$. An equation $t = u$ appearing in one of the conditions $\alpha_1, \ldots, \alpha_n$ of a rule α **if** $\alpha_1 \wedge \ldots \alpha_n \in \mathcal{R}$ is interpreted as a *join condition* $t \downarrow u$. The idea is that terms t and u are joinable in a CRM system $\mathcal{R}$, denoted $\mathcal{R} \vdash t \downarrow u$ iff they can be rewritten to the same term. More formally, $\mathcal{R} \vdash t \downarrow u$ iff there is a term $v \in T_\Sigma(X)$ such that $\mathcal{R} \vdash t \to^* v$ and $\mathcal{R} \vdash u \to^* v$.

To illustrate a CRM system, we present a formalization of unnested powerlists [113] over natural numbers. Unnested powerlists are lists of length 2^n for some $n \in \mathbb{N}$ formed by two constructors: an operator | called *tie* that appends two lists with the same length together, and an operator $\times$ called *zip* that interleaves two powerlists together. For example, both $(1 \mid 2) \mid (3 \mid 4)$ and $(1 \mid 2) \times (3 \mid 4) = (1 \mid 3) \mid (2 \mid 4)$ are well-formed powerlists, while $(4 \mid 3) \times (1)$ and $(1 \mid 2) \mid (3)$ are not well-formed. We only discuss unnested powerlists in this chapter. For a more complete treatment of powerlists that includes nested powerlists, see our case study in Chapter 8.

We formalize unnested powerlists in the `POWERLIST` module defined in Figure 2.5. The module `POWERLIST` imports the predefined module `NAT` from Figure 2.2. We introduce the sort `Pow`, which we will reserve for those terms representing powerlists; Maude automatically introduces also the kind `[Pow]` to denote the kind of the sort `Pow`. We also introduce four operators: `[_]` for representing the operation that forms powerlist elements; `_|_` for representing the powerlist *tie* operation; `_x_` for representing the powerlist *zip* operation; and `len` for representing the operation that computes the length of a powerlist. Since not all terms constructed with the operators `_|_` and `_x_` will represent pow-

erlists, we declare those operators at the kind level. For example, $([2] \mid [3]) \times [4]$ is not a powerlist. This is represented in `POWERLIST` by the fact that the term `([2] | [3]) x [4]` has kind `[Pow]`, but it does not belong to the sort `Pow`. On the other hand, since we want to use the `[_]` operator to *construct* powerlists (specifically, powerlists with only one element), we declare this operator at the sort level and with the `ctor` attribute. Finally, since we expect that the `len` operator applied to a powerlist will always evaluate to a natural number, we declare this operator at the sort level, but without the `ctor` attribute.

In the variable declaration section, we associate the sort `Nat` to the variables `M` and `N`, and the sort `Pow` to the variables `P`, `Q`, `R`, and `S`. By doing this, we are in fact declaring: (i) that `M` and `N` are variables of the kind `[Nat]`, and `P`, `Q`, `R`, and `S` of the kind `[Pow]`, and (ii) that in all memberships and equations in which those variables appear, there is an implicit extra condition stating that those variables only range over the set of terms belonging to their associated sort. Finally, in the membership declaration section, we declare that both the *tie* and the *zip* of two powerlists are powerlists if they have equal length; however, since we do not want to use the zip operator as a *constructor* for powerlists, but rather as a *defined* function, we declare the membership for the zip operator with the `dfn` metadata. In fact, if we go back to the operator declarations section, we can realize that the declarations:

```
op [_] : Nat -> Pow [ctor] .
op len : Pow -> Nat .
```

is just syntactic sugar for the following declarations:

```
op [_] : [Nat] -> [Pow] .
mb [N]: Pow .
op len : [Pow] -> [Nat] .
mb len(P): Nat  [metadata "dfn"].
```

The CRM rewriting framework given in [19] is quite general, but lacks support for a couple different features that are important in many specifications and supported by Maude. The first limitation of CRM systems is that they do not allow rewriting modulo axioms. A second limitation is that all equations in the condition of a CRM system are interpreted as join conditions and variables not in the left-hand side of the equation may not appear in the condition. Maude allows an equation $t = u$ in the condition of a formula to be interpreted as an *oriented* rule $t \to u$, where the right hand side may contain extra variables. Oriented conditions strictly generalize join conditions. When oriented conditions are allowed, we may replace a join condition $t \downarrow u$ by introducing an extra constant tt and binary operator eq along with a rule $\mathrm{eq}(x, x) \to \mathrm{tt}$. The join condition $t \downarrow u$ can then be expressed as the oriented condition $\mathrm{eq}(t, u) \to \mathrm{tt}$.

To support these further extensions, we defined a class of rewrite systems called Conditional Equational Rewrite Membership (CERM) systems in [74].

Equivalence	$\dfrac{t =_A u}{t \rightarrow^* u}$
Replacement	$\dfrac{t =_A C[l\theta] \quad \alpha_1\theta \quad \ldots \quad \alpha_n\theta}{t \rightarrow C[r\theta]}$ with $l \rightarrow r$ **if** $\alpha_1 \wedge \cdots \wedge \alpha_n$ in $\mathcal{R}$
Transitivity	$\dfrac{t \rightarrow u \quad u \rightarrow^* v}{t \rightarrow^* v}$
Membership	$\dfrac{t =_A l\theta \quad \alpha_1\theta \quad \ldots \quad \alpha_n\theta}{t : s}$ with $l : s$ **if** $\alpha_1 \wedge \cdots \wedge \alpha_n$ in $\mathcal{R}$
Subject Reduction	$\dfrac{t \rightarrow u \quad u : s}{t : s}$

Note that $\alpha\theta$ denotes the atomic formula $t\theta \rightarrow^* u\theta$ when α has the form $t \rightarrow u$, and $t\theta : s$ when α has the form $t : s$.

Figure 2.6: Inference system for $\mathcal{R}/A$

CERM systems support all of the features of CRM systems as well as oriented rules in conditions and matching modulo unconditional equations.

Definition 2.6.1. *A* conditional equational rewrite/membership (CERM) system *over a MEL signature $\Sigma = (K, F, S)$ is a pair $\mathcal{R}/A$ in which:*

- *A is a set of unconditional Σ-equations which are* kind-independent *which means that for all well-typed terms $t, u \in T_\Sigma(X)$,*

$$t =_{\overline{A}} u \Rightarrow t =_A u.$$

- *$\mathcal{R}$ is a set containing two types of rules:*

$$l : s \textbf{ if } \bigwedge u_i \rightarrow v_i \wedge \bigwedge w_j : s_j \qquad l \rightarrow r \textbf{ if } \bigwedge u_i \rightarrow v_i \wedge \bigwedge w_j : s_j$$

Membership Rule *Rewrite Rule*

$l, r \in T_\Sigma(X)_k$ and $s \in S_k$ for some $k \in K$, each $u_i, v_i \in T_\Sigma(X)_{k_i}$ for some $k_i \in K$, and each $w_j \in T_\Sigma(X)_{k_j}$ and $s_j \in S_{k_j}$ for some $k_j \in K$.

Conditional equational rewrite systems provide a unifying for defining the operational semantics of systems with conditional memberships such as the `POWERLIST` example and systems with rewriting modulo axioms such as the `NAT-LIST` example in Figure 2.3. Properties such as weak-normalization, termination and confluence can be extended to CERM systems easily.

When α is an atomic formula, $\mathcal{R}/A \vdash \alpha$ denotes that α can be derived from the inferences rules in Figure 2.6 for $\mathcal{R}/A$. This inference system refines and

extends the one used [19, Figure 7]. Specifically, we define refined notions of rewriting and membership directly in the proof theory, combine several rules into a single rule to allow simpler proofs later on, and allow rewrites to take place *modulo* the axioms A.

When $\mathcal{R}/A$ is weakly normalizing and confluent, then for each term $t \in T_\Sigma(X)$ there is an $\mathcal{R}/A$-irreducible term $t \downarrow_{\mathcal{R}/A} \in T_\Sigma(X)$ such that $t \to^!_{\mathcal{R}/A} t \downarrow_{\mathcal{R}/A}$ that is unique up to equivalence modulo $=_A$. We can use this fact to define the canonical term algebra $\mathrm{Can}_{\mathcal{R}/A}$ as the algebra over the $\mathcal{R}/A$-irreducible equivalence classes in T_A as follows:

- For each kind $k \in K$,

$$\mathrm{Can}_{\mathcal{R}/A,k} = \{\, [t] \in T_A \mid t \text{ is } \mathcal{R}/A\text{-irreducible} \,\}.$$

- For each function $f : k_1 \dots k_n \to k$, $\mathrm{Can}_{\mathcal{R}/A,f}$ is the mapping

$$\mathrm{Can}_{\mathcal{R}/A,f} : ([t_1] \dots [t_n]) \mapsto [f(t_1, \dots, t_n) \downarrow_{\mathcal{R}/A}].$$

- For each sort sort $s_k \in S$,

$$\mathrm{Can}_{\mathcal{R}/A,s} = \{\, [t] \in \mathrm{Can}_{\mathcal{R}/A,k} \mid \mathcal{R}/A \vdash t : s \,\}.$$

We let $\mathcal{E} = \mathcal{R} \cup A$ denote the MEL theory in which every atomic formula $t \to u$ appearing in a clause in $\mathcal{R}$ is replaced with the equation $t = u$. In the unsorted case, the algebras $T_\mathcal{E}$ and $\mathrm{Can}_{\mathcal{R}/A}$ are isomorphic provided that $\mathcal{R}/A$ is confluent and weakly normalizing. This is *not true* in general for CERM systems. In addition to confluence and weak normalization, we require two additional properties: $\mathcal{R}/A$ must be *sort-preserving* and *pattern-based*.

Sort-preserving. Sort preservation means that a term t which can be proven to have a sort $s \in S$ may only be rewritten to terms that also can be proven to have sort s. Specifically, a CERM system $\mathcal{R}/A$ is *sort-preserving* relative to variables $Z \subseteq X$ iff reducing a term $t \in T_\Sigma(Z)$ with sort s is guaranteed to result in a term with sort s, i.e., for all $t, u \in T_\Sigma(Z)$ and sorts $s \in S$,

$$\mathcal{R}/A \vdash t : s \wedge \mathcal{R}/A \vdash t \to u \implies \mathcal{R}/A \vdash u : s.$$

This property was called *sort-decreasingness* in [19]. In this definition, note that the term u may be further rewritten before checking whether it has sort s. When $\mathcal{R}$ is sort-preserving relative to $Z = \varnothing$, then we say $\mathcal{R}$ is ground sort-preserving.

Pattern based. If we only consider conditional rewrite systems with join conditions, sort-preservation along with weak normalization and confluence is enough. However rewrite systems with oriented conditions required an additional property. To see this, let Σ be a signature with constants a, b and c all

having the same kind k and no sorts. Then let $A = \varnothing$ and let $\mathcal{R}$ be the rewrite system with the following rules:

$$b \to a \qquad\qquad c \to b \textbf{ if } a \to b$$

This rewrite system is clearly terminating, ground confluent, and ground sort-preserving. However, $\mathrm{Can}_{\mathcal{R}/A}$ contains two elements $\{\, a \,\}$ and $\{\, c \,\}$ while the initial algebra $T_{\mathcal{R}\cup A}$ contains only a single element $\{\, a, b, c \,\}$ and is therefore not isomorphic to $\mathrm{Can}_{\mathcal{R}/A}$.

We can achieve the desired isomorphism between $\mathrm{Can}_{\mathcal{R}/A}$ and $T_{\mathcal{R}\cup A}$ by restricting our attention to theories where each term v appearing in an oriented clause $u \to v$ appearing in the condition of a rule in $\mathcal{R}$ satisfies certain constraints captured in the definition below:

Definition 2.6.2. *A CERM system $\mathcal{R}/A$ is* pattern-based *relative to variables Z when for each rule in $\mathcal{R}$ of the form*

$$\alpha \textbf{ if } \bigwedge_{i\in[1,m]} u_i \to v_i \wedge \bigwedge_{j\in[1,n]} w_j : s_j$$

with α of the form $l \to r$ or $l : s$, we have that each $v_i \in \{\, v_1, \ldots, v_m \,\}$ is a pattern. *Specifically, we require:*

- *The variables in v_i are fresh, i.e.,*

$$\mathrm{vars}(v_i) \cap \big(\mathrm{vars}(l) \cup \bigcup_{1\le k\le i} \mathrm{vars}(u_k) \cup \bigcup_{1\le k<i} \mathrm{vars}(v_k)\big) = \varnothing.$$

- *Any rewriting of a term $v_i\theta$ occurs below the pattern v_i, i.e., for each substitution $\theta : Y \to T_\Sigma(Z)$, if $\mathcal{R}/A \vdash v_i\theta \to^* t$, then there is a substitution $\phi : Y \to T_\Sigma(Z)$ such that $t =_A v_i\phi$.*

The restrictions on variables are based on the idea that conditions in a rule are resolved sequentially, and the variables introduced in the right hand side v_i of an oriented condition $u_i \to v_i$ may not have appeared in terms evaluated earlier. For CERM systems with extra variables in the condition, the order of the conditions may be critical. It is worth noting that the previous definition allows variables to be introduced in the right-hand side r or one of the terms u_i. Without the additional requirement that no extra variables are $\mathrm{vars}(r) \subseteq l \cup \bigcup_{1\le k\le m} \mathrm{vars}(v_k)$ and each $\mathrm{vars}(u_i) \subseteq l \cup \bigcup_{1\le k<i} \mathrm{vars}(v_k)$, pattern-based systems are not directly executable, unless a strategy to instantiate the additional free variables in r and each u_i is given. Nevertheless, our results hold in generality without this extra requirement.

If $\mathcal{R}/A$ is pattern-based, then we can obtain a substitution ϕ satisfying the *oriented conditions* in a rule α **if** $u_1 \to v_1 \wedge \cdots \wedge u_m \to v_m \wedge \ldots$ from a substi-

tution which satisfies the conditions when they are treated as *join conditions*

$$\alpha \textbf{ if } u_1 \downarrow v_1 \wedge \cdots \wedge u_m \downarrow v_m \wedge \ldots.$$

This is shown in the following lemma.

Lemma 2.6.3. *Let $\mathcal{R}/A$ be a CERM system that is pattern-based with respect to variables $Z \subseteq X$ containing a rule*

$$(\forall Y)\, \alpha \textbf{ if } \bigwedge_{i\in[1,m]} u_i \to v_i \wedge \bigwedge_{j\in[1,n]} w_j : s_j.$$

If $\theta : Y \to T_\Sigma(Z)$ is a substitution such that $\mathcal{R}/A \vdash t_i\theta \downarrow u_i\theta$ for $i \in [1,m]$, then there is a substitution $\phi : Y \to T_\Sigma(Z)$ such that:

- *$\mathcal{R}/A \vdash \theta(y) \to^* \phi(y)$ for each $y \in Y$ and*
- *$\mathcal{R}/A \vdash u_i\phi \to^* v_i\phi$ for $i \in [1,m]$.*

Proof. If $\mathcal{R}/A \vdash u_i\theta \to^* v_i\theta$ for all $i \in [1,m]$, then we let $\phi = \theta$ and we are done. Otherwise, there must be an index k such that $\mathcal{R}/A \vdash u_i\phi \to^* v_i\phi$ for all $i < k$ and $\mathcal{R}/A \nvdash u_k\phi \to^* v_k\phi$.

As $\mathcal{R} \vdash u_k\phi \downarrow v_k\phi$, there must be a term t such that $\mathcal{R} \vdash u_k\phi \to^* t$ and $\mathcal{R} \vdash v_k\phi \to^* t$. Moreover, since $\mathcal{R}$ is pattern-based, we know t has the form $t =_A v_k\psi$ for some substitution ψ. Consequently, $\mathcal{R} \vdash u_k\phi \to^* v_k\psi$.

Let ϕ be the substitution such that $\phi(y) = \psi(y)$ if $y \in \mathrm{vars}(v_i)$ and otherwise $\phi(y) = \theta(y)$. As v_i is a pattern, it is easy to show that $u_i\phi = u_i\theta$ for $i \leq k$, $v_i\phi = v_i\theta$ for $i < k$, and $v_i\phi = v_i\psi$. Collectively, this implies that $\mathcal{R} \vdash u_i\phi \to^* v_i\phi$ for all $i \leq k$. We then repeat this process for indices greater than k, and since the total number of indices if finite, we are done. □

With the additional constraint that $\mathcal{R}$ is pattern-based, we can show that the inference system for $\mathcal{R}$ constitutes a complete proof theory [74].

Theorem 2.6.4. *Let $\mathcal{R}/A$ be a confluent, sort-preserving and pattern-based CERM system, and let $\mathcal{E} = \mathcal{R} \cup A$ denote the underlying MEL theory. For all terms $t, u \in T_\Sigma(X)$ and $s \in S$,*

$$\mathcal{E} \vdash t = u \iff \mathcal{R}/A \vdash t \downarrow u \qquad \textit{and} \qquad \mathcal{E} \vdash t : s \iff \mathcal{R}/A \vdash t : s$$

Proof. We can show that $\mathcal{R}/A \vdash t \downarrow u$ implies $\mathcal{E} \vdash t = u$ and $\mathcal{R}/A \vdash t : s$ implies $\mathcal{E} \vdash t : s$ by simultaneous structural induction on CERM proofs formed from the rules in Figure 2.6. The other direction can be shown by structural induction on proofs formed from the rules in Figure 2.4. We consider each of the inference rules which may appear at the top separately:

Subject Reduction

$$\frac{t = u \quad u : s}{t : s}$$

By induction, we know that $\mathcal{R}/A \vdash t \downarrow u$ and $\mathcal{R}/A \vdash u : s$. Consequently, there must be a $v \in T_\Sigma(X)$ such that $\mathcal{R}/A \vdash t \to^* v$ and $\mathcal{R}/A \vdash u \to^* v$. As $\mathcal{R}$ is sort-preserving, $\mathcal{R}/A \vdash v : s$, and thus $\mathcal{R}/A \vdash t : s$.

Membership

$$\frac{u_1\theta = v_1\theta \ \ldots \ u_m\theta = v_m\theta \quad w_1\theta : s_1 \ \ldots \ w_n\theta : s_n}{t\theta : s}$$

By induction $\mathcal{R}/A \vdash u_i\theta \downarrow v_i\theta$ for $i \in [1, m]$, and $\mathcal{R}/A \vdash w_j\theta : s_j$ for $j \in [1, n]$. By definition of $\mathcal{E}$, $\mathcal{R}$ must contain the membership rule:

$$t : s \textbf{ if } \bigwedge_{i=1}^{m} u_i \to v_i \wedge \bigwedge_{j=1}^{n} w_j : s_j.$$

As $\mathcal{R}/A$ is pattern-based, by Lemma 2.6.3, there is a substitution $\phi : Y \to T_\Sigma(X)$ such that $\mathcal{R}/A \vdash \theta(y) \to^* \phi(y)$ for each $y \in Y$, and $\mathcal{R}/A \vdash u_i\phi \to^* v_i\phi$ for each $i \in [1, m]$. It is not difficult to show that for each $j \in [1, n]$, $\mathcal{R}/A \vdash w_j\theta \to^* w_j\phi$ and furthermore $\mathcal{R}/A \vdash w_j\phi : s_j$ as $\mathcal{R}/A$ is sort-preserving. It follows both that $\mathcal{R}/A \vdash t\theta \to^* t\phi$ and $\mathcal{R}/A \vdash t\phi : s$. Therefore, $\mathcal{R}/A \vdash t\theta : s$.

Reflexivity

$$\frac{}{t = t}$$

Clearly $\mathcal{R}/A \vdash t \to^* t$.

Symmetry

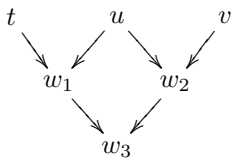
$$\frac{t = u}{u = t}$$

By induction we have $\mathcal{R}/A \vdash t \downarrow u$, which is commutative, and thus $\mathcal{R}/A \vdash u \downarrow t$.

Transitivity

$$\frac{t = u \quad u = v}{t = v}$$

$\mathcal{R}/A \vdash t \downarrow u$ and $\mathcal{R}/A \vdash u \downarrow v$ by induction. Using confluence, we can construct the diagram:



Congruence:

$$\frac{t_1 = u_1 \quad \ldots \quad t_n = u_n}{f(t_1, \ldots, t_n) = f(u_1, \ldots, u_n)}$$

By induction, $\mathcal{R}/A \vdash t_i \downarrow u_i$ for each $i \in [1,n]$. Consequently, there must be a term $v_i \in T_\Sigma(X)$ such that $\mathcal{R}/A \vdash t_i \rightarrow v_i$ and $\mathcal{R}/A \vdash u_i \rightarrow v_i$. It is not difficult to show both that $\mathcal{R}/A \vdash f(t_1,\ldots,t_n) \rightarrow^* f(v_1,\ldots,v_n)$ and $\mathcal{R}/A \vdash f(u_1,\ldots,u_n) \rightarrow^* f(v_1,\ldots,v_n)$. Consequently, $\mathcal{R}/A \vdash f(t_1,\ldots,t_n) \downarrow f(u_1,\ldots,u_n)$.

Replacement:

$$\frac{u_1\theta = v_1\theta \ \ldots \ u_m\theta = v_m\theta \quad w_1\theta : s_1 \ \ldots \ w_n\theta : s_n}{t\theta = t'\theta}$$

If the equation used is in E, then the proof is trivial. Otherwise using an inductive argument identical to that of the **Membership** rule, we can show there must exist a substitution ϕ such that $\mathcal{R}/A \vdash t\theta \rightarrow^* t\phi$ and $\mathcal{R}/A \vdash t\phi \rightarrow^* t'\phi$. It is not difficult to show both that $\mathcal{R}/A \vdash t\theta \rightarrow^* t'\phi$ and $\mathcal{R}/A \vdash t'\theta \rightarrow^* t'\phi$. Consequently, $\mathcal{R}/A \vdash t\theta \downarrow t'\theta$.

□

If $\mathcal{R}/A$ is additionally weakly normalizing, then we have the expected agreement between $\mathrm{Can}_{\mathcal{R}/A}$ and $T_\mathcal{E}$.

Corollary 2.6.5. *If $\mathcal{R}/A$ is a weakly normalizing, ground confluent, ground sort-preserving and ground pattern-based CERM system over a signature Σ and $\mathcal{E} = \mathcal{R} \cup A$, then $\mathrm{Can}_{\mathcal{R}/A}$ is isomorphic to $T_\mathcal{E}$.*

Proof. Let $\Sigma = (K, F, S)$. We define $h : \mathrm{Can}_{\mathcal{R}/A} \rightarrow T_\mathcal{E}$ to be the homomorphism where each mapping h_k is the function

$$h_k : [t]_A \in \mathrm{Can}_{\mathcal{R}/A} \mapsto [t]_\mathcal{E} \in T_\mathcal{E}.$$

We easily have that h is a Σ-homomorphism, and our main task is to show that it is an isomorphism.

As $\mathcal{R}/A$ is weakly-normalizing and ground confluent, for each term $t \in T_\Sigma$ there is a unique normal form $[t \downarrow_{\mathcal{R}/A}] \in \mathrm{Can}_{\mathcal{R}/A}$. Moreover, from Theorem 2.6.4, we know that $\mathcal{E} \vdash t = u$ iff $[t \downarrow_{\mathcal{R}/A}] = [u \downarrow_{\mathcal{R}/A}]$. We also know that for each sort $s \in S$, $[t] \in T_{\mathcal{E},s}$ iff $[t \downarrow_{\mathcal{R}/A}] \in \mathrm{Can}_{\mathcal{R}/A,s}$. It follows that we can define the inverse of h^{-1} of h is a Σ-homomorphism as well, and consequently $\mathrm{Can}_{\mathcal{R}/A}$ and $T_\mathcal{E}$ are isomorphic. □

This last proof shows conditions under which the mathematical semantics of membership equational logic theories and operational semantics of CERM systems coincide. These results are heavily used in Chapter 5 in order to solve an important correctness problem for MEL specifications. However, before discussing different problems in rewriting, we prefer to focus on automated techniques for solving problems.

Chapter 3

Equational tree automata

Tree automata are a theoretical tool with applications in many areas of computer science. Perhaps the fundamental motivation for studying tree automata is that they offer a finite representation of potentially infinite sets of trees with good closure and decidability properties. The sets of trees recognizable by tree automata are called *regular tree languages*. Both the emptiness and membership problems are decidable for regular tree languages, and regular tree languages are closed under the Boolean operations of intersection, union, and complementation as well as under signature homomorphisms [33].

Tree automata were originally introduced in the context of circuit verification, but have proven quite useful in decidability results in logic and term rewriting [33]. Some other recent applications include sufficient completeness of algebraic specifications [32, 75], protocol verification [4, 50, 59], consistency of semi-structured documents [79, 117], type inference [41, 53], querying in databases [133, 144], and theorem proving [94]. Many different frameworks have been proposed for addressing these applications, and each framework must balance the often competing goals of expressive power and tractability of different operations.

In many of these applications, tree automata are used to represent *algebraic terms* formed from different operator symbols, and it is often important that these sets are closed with respect to properties of the operators. For example, in protocol verification, one may use a regular tree language to represent the potential knowledge of an intruder. In this case, if the intruder can learn the value of the sum $(x + y) + z$, then the tree language should contain the term $(x+y)+z$. However, the language should contain algebraically equivalent terms such as $x + (y + z)$ or $(z + x) + y$, as the intruder who knows $(x + y) + z$ can easily infer the other terms. Given a theory $\mathcal{E}$ and a tree language $\mathcal{L}$, we call the set of terms equivalent modulo $\mathcal{E}$ to a term in $\mathcal{L}$ the *equational closure* of $\mathcal{L}$.

Unfortunately, the equational closure of a language is rarely regular. For example, the set of terms equivalent modulo associativity to a term in a regular tree language is not in general a regular tree language [123]. As a consequence, users of tree automata techniques are often forced to resort to complicated and specialized ways of encoding the problem as a tree automaton problem [14]. Many extensions of tree automata have been suggested to address this prob-

lem, including multitree automata by Lugiez [105], two-way alternating tree automata by Verma [136, 138], and equational tree automata by Ohsaki [123].

In the completeness applications which we discuss later in Chapters 5 and 6, the most important properties for a tree automata framework are: (1) an effective procedure for emptiness checking, (2) closure under Boolean operations, and (3) closure under equational congruences. Regular tree automata have a decidable emptiness problem and are closed under Boolean operations, however they are not closed in general under equational congruences. Equational tree automata add closure under equational congruences, but are not closed under Boolean operations. Multitree automata [105] satisfy all three properties, but are only defined for AC theories and the other frameworks lack closure under Boolean operations. Due to these problems, we proposed the *propositional tree automata* (PTA) framework in [76]. PTA are closed under both an equational theory and Boolean operations — but have an undecidable emptiness problem.

As few properties are decidable for arbitrary theories, our work, like most work on equational tree automata, focuses on equational theories with particular combinations of axioms. In particular, we focus on theories containing particular combinations of associativity (A), commutativity (C), and idempotence (I). The main focus of this chapter is to solve the emptiness problem for a Boolean combination of equational tree automata over theories with specific combinations of axioms. We call this problem the *propositional emptiness problem* as it is equivalent to the emptiness problem for propositional tree automata. We focus on this problem, because it generalizes many other tree automata problems, and it is useful for the applications discussed in Chapters 5 and 6.

This restriction to particular theories is unavoidable due to decidability issues, but leaves open the question as to whether these results can be *combined*. For example, using known closure properties, it is easy to show that intersection emptiness is decidable for tree automata over a theory $\mathcal{E}_{\mathrm{AC}}$ with an AC symbol and free symbols [125] as well as a theory $\mathcal{E}_{\mathrm{ACI}}$ with an ACI symbol and free symbols [136]. Does this imply decidability of intersection emptiness for tree automata over the combined theory $\mathcal{E}_{\mathrm{AC}} \cup \mathcal{E}_{\mathrm{ACI}}$?

In fact, we show that intersection emptiness is *undecidable* for tree automata over $\mathcal{E}_{\mathrm{AC}} \cup \mathcal{E}_{\mathrm{ACI}}$. We obtain this result by showing that every *alternating tree language* [138] over a theory $\mathcal{E}$ can be effectively expressed as the intersection of two regular tree languages over a theory $\mathcal{E}'$ containing $\mathcal{E}$ and an additional ACI symbol. Since the emptiness problem for alternating AC-tree automata is undecidable [138], it follows that intersection emptiness is undecidable for regular tree automata over $\mathcal{E}_{\mathrm{AC}} \cup \mathcal{E}_{\mathrm{ACI}}$.

Our result implies that decidability of intersection emptiness is a *non-modular* property — even for disjoint combinations of theories. Modularity is an important property to have, because it aids in the process of decomposing complex problems into simpler parts which can be reasoned about separately. For example, the Nelson-Oppen [118] and Shostak [134] combination methods have

been fundamental to the development of general-purpose theorem provers that combine the capabilities of many different decision procedures. Given the importance of modularity, we decided to study two specific equational combinations of tree automata: (1) theories with any combination of associativity and commutative equations including free, associative (A), commutative (C), and associative and commutative (AC) symbols; (2) restricted subclasses of automata with AC and ACI symbols.

Combinations of associativity and commutativity. In our first combination problem, we study the propositional emptiness problem for theories where symbols may satisfy any combination of associativity and commutativity axioms. We show how previously known results easily imply that the propositional emptiness problem is undecidable for theories with an associative symbol that is not commutative, but decidable for AC symbols (i.e., symbols that are associative and commutative).

For arbitrary combinations of associativity and commutativity, we show that tree automata and machine learning techniques can be combined to create a semi-decision procedure which can always show non-emptiness, and can show emptiness under certain regularity conditions. Our algorithm has been implemented in a tree automata software library, called *CETA* [69], that can check the emptiness of propositional tree automata modulo associativity, commutativity, and identity. Although it is a semi-decision procedure and not guaranteed to terminate, the CETA library has been quite successful in practice when used to solve the applications we discuss later in Chapter 5 and 6.

AC intersection free automata. Our next contribution in this chapter is to define a restricted class of tree automata over a theory $\mathcal{E}$ with AC and ACI symbols. We further show that the propositional emptiness problem is decidable for this class. We call tree automata in the restricted class *AC intersection free* tree automata and require that each ACI symbol $+$ in $\mathcal{E}$ satisfies one of two constraints: (1) either the clauses in the automaton where $+$ appears must satisfy certain syntactic restrictions to avoid simulating the intersection clauses of alternating tree automata; or (2) the idempotence equation $x + x = x$ in $\mathcal{E}$ must be treated as a rewrite rule $x + x \to x$ as in the *tree automata with normalization* framework of [124].

In the tree automata with normalization framework, some of the equations in $\mathcal{E}$ may be treated as rewrite rules in a confluent and terminating rewrite theory $\mathcal{R}$. Rather than computing the congruence closure of the tree language modulo $\mathcal{E}$, terms are first normalized by rewriting with $\mathcal{R}$ modulo the remaining equations $\mathcal{E}' \subseteq \mathcal{E}$, and then checked for membership in the underlying equational tree languages $\mathcal{L}(\mathcal{A}/\mathcal{E}')$. This framework has different semantics than standard equational tree automata, but is often able to obtain better closure and decidability properties [124].

An important consequence of our decidability result on AC intersection free automata is that it solves two open problems: (1) We show that the emptiness

problem is decidable for tree automata with normalization over idempotence rules and AC equations. This problem was mentioned in [124] and left unsolved. (2) We show that the propositional emptiness problem is decidable for equation tree automata over the theory $\mathcal{E}_{\text{ACI}}$ containing a single ACI symbol and arbitrary free symbols. This problem is interesting, because equational tree automata over $\mathcal{E}_{\text{ACI}}$ are not closed under complementation [137]. Its decidability also has a further implication: propositional emptiness is a non-modular property. Our earlier undecidability result implies that propositional emptiness is undecidable for equational tree automata over $\mathcal{E}_{\text{AC}} \cup \mathcal{E}_{\text{ACI}}$, while propositional emptiness is decidable for $\mathcal{E}_{\text{AC}}$ [125].

One underlying goal in this work is to develop better tree automata techniques for *non-linear theories*. This is important in applications such as sufficient completeness checking where existing techniques either do not support rewriting modulo axioms [32] or are restricted to left-linear rewrite rules [75]. Although sufficient completeness checking is undecidable in general for specifications with non-linear rules and rewriting modulo AC [90], our decidability results show that sufficient completeness is decidable modulo AC when every non-linear rule in the specification has the form $f(x, x) \to r$. It would be interesting to see if the techniques presented here can be extended to other forms of non-linear rules.

The rest of this chapter is organized as follows: In Section 3.1, we introduce the basic definitions for equational tree automata. In Section 3.2, we present our result showing that intersection emptiness is a non-modular property for equational tree automata. We then investigate two specific combinations of theories: the combination of A and AC symbols in Section 3.3 and the combination of AC and ACI symbols in Section 3.4. Finally, we conclude this chapter with a discussion of directions for future research in Section 3.5. Many of the results in this chapter have previously appeared in [72, 76].

3.1 Equational tree automata definitions

We treat tree automata as collections of Horn clauses of particular forms as in [138]. Given a Σ-theory $\mathcal{E}$, a *regular $\mathcal{E}$-tree automaton* $\mathcal{A}$ is a finite set of Horn clauses each with the form:

$$p(f(x_1, \ldots, x_n)) \Leftarrow p_1(x_1), \ldots, p_n(x_n) \qquad \text{regular clause}$$

where $f \in \Sigma$ has arity n and $p, p_1, \ldots, p_n$ are elements of a finite set of unary predicate symbols Q called the *states* of the automaton. In some definitions, tree automata may also contain *ϵ-clauses* of the form $p(x) \Leftarrow q(x)$, but these can be eliminated without loss of expressive power [33]. We write $\mathcal{A}/\mathcal{E} \vdash p(t)$ if $p(t)$ is entailed by the axioms in $\mathcal{A} \cup \mathcal{E}$. There are a variety of different inference systems for entailment with equivalent semantics. When it is necessary to refer

$$\text{Equivalence} \qquad \frac{t =_{\mathcal{E}} u \qquad \mathcal{A}/\mathcal{E} \vdash p(u)}{\mathcal{A}/\mathcal{E} \vdash p(t)}$$

$$\text{Membership} \qquad \frac{\mathcal{A}/\mathcal{E} \vdash \alpha_1\theta \qquad \ldots \qquad \mathcal{A}/\mathcal{E} \vdash \alpha_n\theta}{\mathcal{A}/\mathcal{E} \vdash \alpha\theta} \quad \text{if } \alpha \Leftarrow \alpha_1 \ldots \alpha_n \in \mathcal{A}$$

Figure 3.1: Inference System for $\mathcal{A}/\mathcal{E}$

to a specific inference steps, we use the inference rules in Figure 3.1.

We keep the acceptance condition separate from the automaton itself, and since the automaton only recognizes languages that are closed modulo $\mathcal{E}$, we define languages as subsets of $T_{\mathcal{E}}$ rather than T_Σ. For each state p belonging to $\mathcal{A}$, the *language recognized by p in $\mathcal{A}$*, denoted $\mathcal{L}_p(\mathcal{A}/\mathcal{E}) \subseteq T_{\mathcal{E}}$, is defined by

$$\mathcal{L}_p(\mathcal{A}/\mathcal{E}) = \{\, [t] \in T_{\mathcal{E}} \mid \mathcal{A}/\mathcal{E} \vdash p(t) \,\}. \tag{3.1}$$

For an equational theory $\mathcal{E} = \varnothing$ with no equations, we write $\mathcal{A} \vdash p(t)$ for $\mathcal{A}/\mathcal{E} \vdash p(t)$ and $\mathcal{L}_p(\mathcal{A})$ for $\mathcal{L}_p(\mathcal{A}/\mathcal{E})$.

One important result from [138] about regular $\mathcal{E}$-tree automata is the following:

Theorem 3.1.1. *For each theory $\mathcal{E}$ and regular $\mathcal{E}$-tree automaton $\mathcal{A}$,*

$$\mathcal{A}/\mathcal{E} \vdash p(t) \iff (\exists u \in [t]_{\mathcal{E}})\, \mathcal{A} \vdash p(u).$$

□

This theorem implies that an equational tree language can be alternatively defined as the quotient of a regular tree language.

Corollary 3.1.2. *For each theory $\mathcal{E}$ and regular $\mathcal{E}$-tree automaton $\mathcal{A}$,*

$$\mathcal{L}_p(\mathcal{A}/\mathcal{E}) = \{\, [t] \in T_{\mathcal{E}} \mid (\exists u \in \mathcal{L}_p(\mathcal{A})\, t =_{\mathcal{E}} u \,\}.$$

For an arbitrary theory $\mathcal{E}$, the class of languages recognized by regular $\mathcal{E}$-tree automata is closed under union, but not under intersection or complementation [123]. Motivated by this fact, we introduced *propositional tree automata* in [76]. This framework is an extension to equational tree automata framework that is effectively closed under Boolean operations in all theories. The key idea is to use a propositional formula rather than a set of final states as the acceptance condition for defining the language recognized by the automaton. We present a slightly simpler formalization that preserves the basic idea. Given a tree automaton $\mathcal{A}$ with states Q, we extend the definition (3.1) of a language

$\mathcal{L}_p(\mathcal{A}/\mathcal{E})$ recognized by a state p to languages $\mathcal{L}_\phi(\mathcal{A}/\mathcal{E})$ recognized by a propositional formula ϕ constructed from atomic predicates Q and Boolean connectives $\wedge$ and $\neg$:

$$\mathcal{L}_{\phi_1 \wedge \phi_2}(\mathcal{A}/\mathcal{E}) = \mathcal{L}_{\phi_1}(\mathcal{A}/\mathcal{E}) \cap \mathcal{L}_{\phi_2}(\mathcal{A}/\mathcal{E}) \qquad \mathcal{L}_{\neg\phi_1}(\mathcal{A}/\mathcal{E}) = T_{\mathcal{E}} - \mathcal{L}_{\phi_1}(\mathcal{A}/\mathcal{E}).$$

As we will later see in our discussion of decision problems, there is a drawback of using propositional tree automata — the emptiness problem is undecidable in general. However, propositional tree automata over the same theory do have trivial algorithms for performing Boolean operations. Given a propositional tree language $\mathcal{L}_\phi(\mathcal{A}/\mathcal{E})$, the complement is just the language $\mathcal{L}_\phi(\mathcal{A}/\mathcal{E})$. Given two propositional tree languages $\mathcal{L}_{\phi_1}(\mathcal{A}_1/\mathcal{E})$ and $\mathcal{L}_{\phi_2}(\mathcal{A}_2/\mathcal{E})$, one can use renaming to guarantee the states of $\mathcal{A}'$ and ϕ' are disjoint from those in $\mathcal{A}$. It is then not difficult to show that the intersection of $\mathcal{L}_{\phi_1}(\mathcal{A}_1/\mathcal{E})$ and $\mathcal{L}_{\phi_2}(\mathcal{A}_2/\mathcal{E})$ is the language $\mathcal{L}_{\phi_1 \wedge \phi_2}(\mathcal{A}_1 \uplus \mathcal{A}_2/\mathcal{E})$ where $\mathcal{A}_1 \uplus \mathcal{A}_2$ denotes the disjoint union of $\mathcal{A}_1$ and $\mathcal{A}_2$.

Example. One important relationship is that every order-sorted signature $\Sigma = (S, F, \leq)$ can be viewed as a regular tree automaton $\mathcal{A}_\Sigma$ whose states are the sorts S. Specifically, we map each operator declaration $f : s_1 \ldots s_n \rightarrow s$ to a regular clause

$$s(f(x_1, \ldots, x_n)) \Leftarrow s_1(x_1), \ldots, s_n(x_n),$$

and we map each subsort declaration $s < s'$ to an ϵ-clause $s'(x) \Leftarrow s(x)$. It is not difficult to show then that for each sort $s \in S$, the set of terms with sort s, $T_\Sigma(X)_s$, is identical to the language $\mathcal{L}_s(\mathcal{A}_\Sigma)$. This mapping between order-sorted signatures and regular tree automata can be done in the other direction by viewing the states of the automaton as sorts and the regular rules in the automaton as operator declarations.

As an example, consider the `NAT-LIST` module defined in Figure 2.3 of Section 2.4. In the automaton $\mathcal{A}_{\texttt{NAT-LIST}}$ representing the signature of `NAT-LIST`, the states are the sorts `Nat`, `NeList` and `List`. The subsort declarations `Nat` $<$ `NeList` and `NeList` $<$ `List` induce the ϵ-clauses:

$$\texttt{List}(x) \Leftarrow \texttt{NeList}(x) \qquad\qquad \texttt{NeList}(x) \Leftarrow \texttt{Nat}(x)$$

The operator declarations induce the regular clauses:

$$\begin{aligned}
\texttt{Nat}(\texttt{0}) &\\
\texttt{Nat}(\texttt{s}\ x) &\Leftarrow \texttt{Nat}(x)\\
\texttt{List}(\texttt{nil}) &\\
\texttt{NeList}(xy) &\Leftarrow \texttt{NeList}(x), \texttt{NeList}(y)\\
\texttt{List}(xy) &\Leftarrow \texttt{List}(x), \texttt{List}(y)\\
\texttt{Nat}(\texttt{head}(x)) &\Leftarrow \texttt{NeList}(x)\\
\texttt{Nat}(\texttt{end}(x)) &\Leftarrow \texttt{NeList}(x)\\
\texttt{List}(\texttt{reverse}(x)) &\Leftarrow \texttt{List}(x)
\end{aligned}$$

For *sort-independent* order-sorted theories $\mathcal{E}$, this connection can be extended to *equational* tree automata. Let $\overline{\mathcal{E}}$ denote the unsorted theory obtained by dropping the sort information from $\mathcal{E}$, and recall Definition 2.4.1 which states that an order-sorted theory is sort-independent if for each pair of well sorted terms $t, u \in T_\Sigma(X)$,

$$t =_\mathcal{E} u \iff t =_{\overline{\mathcal{E}}} u.$$

For each sort $s \in S$, by Theorem 3.1.1, the language $\mathcal{L}_s(\mathcal{A}_\Sigma/\overline{\mathcal{E}})$ is the equivalence classes modulo $=_{\overline{\mathcal{E}}}$ of terms in $\mathcal{L}_s(\mathcal{A}_\Sigma)$ As $\mathcal{L}_s(\mathcal{A}_\Sigma) = T_{\Sigma,s}$, our assumption that $\mathcal{E}$ is sort-independent means that the set of well-sorted equivalence classes $T_{\mathcal{E},s}$ is isomorphic to $\mathcal{L}_s(\mathcal{A}_\Sigma/\overline{\mathcal{E}})$ with the bijective mapping

$$h : [t]_\mathcal{E} \in T_{\mathcal{E},s} \mapsto [t]_{\overline{\mathcal{E}}} \in \mathcal{L}_s(\mathcal{A}_\Sigma/\overline{\mathcal{E}}).$$

This relationship between sort-independent order-sorted theories is fundamental to our tree automata-based sufficient completeness checker (Chapter 5) and canonical completeness checker (Chapter 6). These chapters show how different decision problems can be cast as decision problems for propositional tree languages. As a simple example, we could use tree automata techniques described later in this chapter to check that every list in `NAT-LIST` is equivalent modulo the axioms to either `nil` or a non-empty list. We first let $\overline{\mathcal{E}}_{\texttt{NAT-LIST}}$ denote the unsorted theory containing the associativity and identity axioms in `NAT-LIST`, and then define the automaton $\mathcal{B}$ containing the rules in $\mathcal{A}_{\texttt{NAT-LIST}}$ as well an additional state `Nil` and rule $\texttt{Nil}(\texttt{nil})$. It is not difficult to see that the language

$$\mathcal{L}_{\texttt{List}\wedge\neg\texttt{Nil}\wedge\neg\texttt{NeList}}(\mathcal{B}/\overline{\mathcal{E}}_{\texttt{NAT-LIST}})$$

accepts the equivalence class of ground terms that are lists, but not equivalent to `nil` or `NeList`. For this simple example, it is easy to see that this language is empty by hand, however the results in this section can be used to automatically check emptiness of much more complicated examples.

Decision problems. We now define a few of the many of decision problems that

have been studied in the context of tree automata and equational tree automata. The *membership problem* for $\mathcal{E}$ is the problem of deciding for an equivalence class $[t] \in T_{\mathcal{E}}$, $\mathcal{E}$-tree automaton $\mathcal{A}$ and state p in $\mathcal{A}$ whether $[t] \in \mathcal{L}_p(\mathcal{A}/\mathcal{E})$. Membership is undecidable for arbitrary theories since otherwise one could solve arbitrary equivalences $t =_{\mathcal{E}} u$. However, it is decidable in the case where each equivalence class $[t]_{\mathcal{E}}$ is finite. In this case, one can enumerate the elements of $[t]_{\mathcal{E}}$ and check each for membership in $\mathcal{L}_p(\mathcal{A})$.

The *emptiness problem* for $\mathcal{E}$ is the problem of deciding for an $\mathcal{E}$-tree automaton $\mathcal{A}$ and state p whether $\mathcal{L}_p(\mathcal{A}/\mathcal{E}) = \varnothing$. This problem is decidable in linear time for an arbitrary theory $\mathcal{E}$. Corollary 3.1.2 implies that $\mathcal{L}_p(\mathcal{A}/\mathcal{E}) = \varnothing$ iff $\mathcal{L}_p(\mathcal{A}) = \varnothing$ and the question of whether $\mathcal{L}_p(\mathcal{A}) = \varnothing$ is decidable in linear time using standard tree automata techniques [33].

The *intersection emptiness problem* for $\mathcal{E}$ is the problem of deciding for an $\mathcal{E}$-tree automaton $\mathcal{A}$ and states $p_1, \ldots, p_n$ of $\mathcal{A}$ whether $\mathcal{L}_{p_1}(\mathcal{A}/\mathcal{E}) \cap \cdots \cap \mathcal{L}_{p_n}(\mathcal{A}/\mathcal{E}) = \varnothing$. Finally, the *propositional emptiness problem* for $\mathcal{E}$ is the problem of deciding for an $\mathcal{E}$-tree automaton $\mathcal{A}$ with states Q and propositional formula ϕ over atomic predicates Q whether $\mathcal{L}_\phi(\mathcal{A}/\mathcal{E}) = \varnothing$. Both the intersection emptiness and propositional emptiness problems are decidable for regular equational tree automata over a theory $\mathcal{E}_{\mathrm{AC}}$ with AC and free symbols [123]. In contrast, both intersection emptiness and propositional emptiness are undecidable for regular equational tree automata over a theory $\mathcal{E}_{\mathrm{A}}$ with associative and free symbols [125]. As an example of a tree automata framework where intersection emptiness is decidable and propositional emptiness is undecidable, we refer the reader to the *monotone AC tree automata* framework of [127].

3.2 Non-modularity of intersection emptiness

One extension to tree automata is the alternating tree automata framework of [135], which was extended to the equational case in [138]. In a Horn-clause representation, an *alternating tree automaton* is a tree automaton which in addition to regular clauses, may also contain *intersection clauses* of the form:

$$p(x) \Leftarrow p_1(x), p_2(x) \qquad \text{intersection clause.}$$

Alternating $\mathcal{E}$-tree automata are closed under both intersection and union, but are not always closed under complementation. If $\mathcal{E}$ is the free theory, i.e., $\mathcal{E} = \varnothing$, then the class of languages recognized by alternating and regular automata coincide. However, this is often not the case for other theories. For example, alternating AC-tree automata are strictly more powerful than regular AC-automata. In particular, the emptiness problem is undecidable for alternating AC-tree automata [138].

Our first new result in this section is to show that every alternating $\mathcal{E}$-tree language is isomorphic to the intersection of two regular $\mathcal{E}'$-tree languages where

$\mathcal{E}'$ is the theory obtained by adding a fresh ACI symbol $\circ$ to $\mathcal{E}$.

Theorem 3.2.1. *Let $\mathcal{E}$ and $\mathcal{E}$' be equational theories over the signatures Σ and Σ' respectively such that $\mathcal{E}'$ contains the symbols and equations in $\mathcal{E}$ and adds a fresh ACI operator $\circ$.*

Given an alternating $\mathcal{E}$-tree automaton $\mathcal{A}$ with states Q, one can effectively construct a regular $\mathcal{E}'$-tree automaton $\mathcal{B}$ containing the states Q and an additional fresh state k such that

- *For all $p \in Q$ and $t \in T_\Sigma$, $\mathcal{A}/\mathcal{E} \vdash p(t) \iff \mathcal{B}/\mathcal{E}' \vdash p(t)$.*
- *For all $t \in T_{\Sigma'}$, $\mathcal{B}/\mathcal{E}' \vdash k(t) \iff (\exists u \in T_\Sigma)\, t =_{\mathcal{E}'} u$.*

Proof. Let $\mathcal{B}$ be the automaton only containing the following clauses:

- $\mathcal{B}$ contains all of the clauses in $\mathcal{A}$ that are not intersection clauses.
- For each intersection clause $p(x) \Leftarrow p_1(x), p_2(x)$ in $\mathcal{A}$, $\mathcal{B}$ contains the clause

$$p(x_1 \circ x_2) \Leftarrow p_1(x_1), p_2(x_2).$$

- For each symbol $f \in \Sigma$ with arity n, $\mathcal{B}$ contains the clause

$$k(f(x_1, \ldots, x_n)) \Leftarrow k(x_1), \ldots, k(x_n).$$

We first show that $\mathcal{A}/\mathcal{E} \vdash p(t)$ implies $\mathcal{B}/\mathcal{E}' \vdash p(t)$ for all $p \in Q$. Since $\mathcal{B}$ contains all the clauses in $\mathcal{A}$ other than the intersection clauses, all we need to show is that $\mathcal{B} \cup \mathcal{E}'$ entails each intersection clause $q(x) \Leftarrow q_1(x), q_2(x)$ in $\mathcal{A}$. This is immediate, because $\mathcal{B}$ must contain the clause $q(x_1 \circ x_2) \Leftarrow q_1(x_1), q_2(x_2)$, and so $\mathcal{B}$ entails $q(x \circ x) \Leftarrow q_1(x), q_2(x)$. The theory $\mathcal{E}'$ contain the axiom $x \circ x = x$, and thus $\mathcal{B} \cup \mathcal{E}'$ entails $q(x) \Leftarrow q_1(x), q_2(x)$.

We now show that $\mathcal{B}/\mathcal{E}' \vdash p(t)$ implies $\mathcal{A}/\mathcal{E} \vdash p(t)$ for all $p \in Q$. If $\mathcal{B}/\mathcal{E}' \vdash p(t)$ then by Theorem 3.1.1, there is a term $u \in T_{\Sigma'}$ such that $t =_{\mathcal{E}'} u$ such that $\mathcal{B} \vdash p(u)$. To prove that $\mathcal{A}/\mathcal{E} \vdash p(t)$, we construct a term $v \in T_\Sigma$ such that $v =_{\mathcal{E}'} u$ and $\mathcal{A}/\mathcal{E} \vdash p(v)$. Since $t =_{\mathcal{E}'} u =_{\mathcal{E}'} v$ and neither t nor v contain the added symbol $\circ$, it is not difficult to show that $t =_{\mathcal{E}} v$, and thus $\mathcal{A}/\mathcal{E} \vdash p(t)$.

We construct $v \in T_\Sigma$ in a bottom-up fashion from the proof that $\mathcal{B} \vdash p(u)$. Each inference step with the form

$$\frac{\mathcal{B} \vdash q_1(u_1) \quad \ldots \quad \mathcal{B} \vdash q_n(u_n)}{\mathcal{B} \vdash q(f(u_1, \ldots, u_n))}$$

referencing a symbol $f \neq \circ$ has a direct corresponding inference step using the clauses in $\mathcal{A}$. For each $i \in [1, n]$, we first obtain the term $v_i \in T_\Sigma$ such that $v_i =_{\mathcal{E}'} u_i$ and $\mathcal{A}/\mathcal{E} \vdash q_i(v_i)$. If we let $v = f(v_1, \ldots, v_n)$, then clearly, $u = f(u_1, \ldots, u_n) =_{\mathcal{E}'} v$ and $\mathcal{A}/\mathcal{E} \vdash q(v)$.

On the other hand, given an inference step of the form

$$\frac{\mathcal{B} \vdash q_1(u_1) \qquad \mathcal{B} \vdash q_2(u_2)}{\mathcal{B} \vdash q(u_1 \circ u_2)}$$

with $q(x_1 \circ x_2) \Leftarrow q_1(x_1), q_2(x_2)$ in $\mathcal{B}$, we first observe that $u_1 =_{\mathcal{E}'} u_2 =_{\mathcal{E}'} u_1 \circ u_2$, because $u_1 \circ u_2$ is a subterm of u, and u is equivalent to $t \in T_\Sigma$ which does not contain the symbol $\circ$. By induction, we know that for $i \in [1, 2]$, there is a term $v_i \in T_\Sigma$ such that $u_i =_{\mathcal{E}'} v_i$ and $\mathcal{A}/\mathcal{E} \vdash q_i(v_i)$. As $v_1 =_{\mathcal{E}'} u_1 =_{\mathcal{E}'} u_2 =_{\mathcal{E}'} v_2$ and both v_1 and v_2 are in T_Σ, it follows that $v_1 =_{\mathcal{E}} v_2$, and thus $\mathcal{A}/\mathcal{E} \vdash p_2(v_1)$. By using the intersection clause $p(x) \Leftarrow p_1(x), p_2(x)$ in $\mathcal{A}$, it follows that $\mathcal{A}/\mathcal{E} \vdash p(v_1)$ and thus we are done as $v_1 =_{\mathcal{E}} u_1 =_{\mathcal{E}} u_1 \circ u_2$.

Finally, we show that $\mathcal{B}/\mathcal{E}' \vdash k(t)$ if and only if there exists a term $u \in T_\Sigma$ not containing the fresh operator $\circ$ such that $t =_{\mathcal{E}'} u$. This follows by first observing that $\mathcal{B} \vdash k(u)$ iff u is in T_Σ, and so by Theorem 3.1.1,

$$\mathcal{B}/\mathcal{E}' \vdash k(t) \iff (\exists u \in [t]_{\mathcal{E}'})\, \mathcal{B} \vdash k(u) \iff T_\Sigma \cap [t]_{\mathcal{E}'} \neq \varnothing.$$

□

From this theorem, it follows that for each state $p \in Q$, the languages $\mathcal{L}_p(\mathcal{A}/\mathcal{E})$ and $\mathcal{L}_p(\mathcal{B}/\mathcal{E}') \cap \mathcal{L}_k(\mathcal{B}/\mathcal{E}')$ are isomorphic with the bijective mapping

$$h_p \ : \ [t]_{\mathcal{E}} \in \mathcal{L}_p(\mathcal{A}/\mathcal{E}) \ \mapsto \ [t]_{\mathcal{E}'} \in \mathcal{L}_p(\mathcal{B}/\mathcal{E}') \cap \mathcal{L}_k(\mathcal{B}/\mathcal{E}').$$

Although this connection between alternating and regular languages seems worth further study, our main interest in this result is that it allows us to use the result in [138] about the undecidability of emptiness for alternating AC-tree automata to show that intersection emptiness is undecidable for regular tree automata over a theory $\mathcal{E}$ with both AC and ACI symbols.

Corollary 3.2.2. *If $\mathcal{E}$ is an equational theory with at least 4 constants, an AC symbol, and an ACI symbol, then the intersection emptiness problem for regular tree automata over $\mathcal{E}$ is undecidable.*

Proof. Let $\mathcal{E}_{\mathrm{AC}}$ denote the equational theory obtained by removing the ACI symbol from $\mathcal{E}$. The theory $\mathcal{E}_{\mathrm{AC}}$ is *torsion-free* according to the definition in [138] with regard to the 4 constants, and consequently the emptiness problem is undecidable for alternating $\mathcal{E}_{\mathrm{AC}}$-tree automata by Prop. 11 in [138]. By Theorem 3.2.1, for each alternating automaton $\mathcal{A}$, we can construct a regular $\mathcal{E}$-tree automaton $\mathcal{B}$ such that $\mathcal{L}_p(\mathcal{A}/\mathcal{E}_{\mathrm{AC}}) = \varnothing$ iff $\mathcal{L}_p(\mathcal{B}/\mathcal{E}) \cap \mathcal{L}_k(\mathcal{B}/\mathcal{E}) = \varnothing$. □

The theory $\mathcal{E}$ in the previous statement can be partitioned into disjoint theories $\mathcal{E}_{\mathrm{AC}}$ and $\mathcal{E}_{\mathrm{ACI}}$ where $\mathcal{E}_{\mathrm{AC}}$ contains the AC symbol and $\mathcal{E}_{\mathrm{ACI}}$ contains the ACI symbol and the constants are split freely between them. Intersection emptiness is decidable for both $\mathcal{E}_{\mathrm{AC}}$ [125] and $\mathcal{E}_{\mathrm{ACI}}$ [136], but as the previous statement shows it is undecidable for $\mathcal{E} = \mathcal{E}_{\mathrm{AC}} \cup \mathcal{E}_{\mathrm{ACI}}$. It follows that intersection

emptiness is a *non-modular* property for equational tree automata even for combinations of disjoint theories.

3.3 A+AC propositional emptiness

In this section, we study the *propositional emptiness problem* for equational tree automata — given an automaton $\mathcal{A}$ over a theory $\mathcal{E}$ and propositional formula ϕ over the states in $\mathcal{A}$, does $\mathcal{L}_\phi(\mathcal{A}/\mathcal{E}) = \varnothing$? This problem is undecidable in general, and computationally quite hard even for specific theories where it is decidable. Even in the free case, the problem is EXPTIME-complete. The tree automata universality problem ($\mathcal{L}_q(\mathcal{A}) = T_\Sigma$) is EXPTIME-complete [33, Theorem 14]. This problem can be converted in linear time into the propositional emptiness problem of $\mathcal{L}_{\neg q}(\mathcal{A})$.

In the AC case, regular equational tree automata are known to be closed under Boolean operations [133], and the emptiness problem is decidable [125]. It follows that propositional emptiness is decidable for tree automata over an AC-theory. In the A case (associativity without commutativity), the emptiness problem is undecidable [76]:

Theorem 3.3.1. *The propositional emptiness problem for tree automata over a theory $\mathcal{E}_\mathrm{A}$ with a single associative symbol and constants is undecidable.*

Proof. It was shown in [123] that it is undecidable whether $\mathcal{L}_q(\mathcal{B}/\mathcal{E}_\mathrm{A}) = T_{\mathcal{E}_\mathrm{A}}$ for an arbitrary regular $\mathcal{E}_\mathrm{A}$-equational tree automaton $\mathcal{B}$ containing a state q. This problem is equivalent to checking whether $\mathcal{L}_{\neg q}(\mathcal{B}/\mathcal{E}_\mathrm{A}) = \varnothing$. □

Despite the lack of decidability, our goal in this section is to develop a semi-decision procedure that works well in practice for a theory $\mathcal{E}$ containing: (1) *free symbols* $\Sigma_\varnothing$; (2) symbols Σ_AC that are *associative and commutative* (AC) in $\mathcal{E}$; and (3) symbols Σ_A that appear in an associativity axiom in $\mathcal{E}$, but no other axioms. If Σ_A is empty, we can obtain a decision procedure. Due to undecidability, we cannot hope for a decision procedure for associative symbols. However, by using techniques from machine learning, we have developed a semi-algorithm which may not terminate, but can always show non-emptiness, and show emptiness if the language is empty and certain regularity conditions are satisfied by each language recognized by states in the automaton.

Our decision procedure can be further extended to handle *commutative* symbols that are not associative by a completion process which adds additional clauses to $\mathcal{A}$ and then treating commutative symbols as free symbols. Specifically, for each clause $p(f(x_1, x_2)) \Leftarrow p_1(x_1), p_2(x_2)$ in $\mathcal{A}$ where f is commutative, we add the implied clause $p(f(x_1, x_2)) \Leftarrow p_2(x_1), p_1(x_2)$. After performing this completion, we can treat f as a free symbol without affecting the languages accepted by each state.

Our semi-algorithm generalizes the subset construction algorithm for regular tree automata in the free case presented in [33, Theorem 1.1.9]. For a given

automaton $\mathcal{A}$ with states Q over a theory $\mathcal{E}$ and each equivalence class $[t] \in T_{\mathcal{E}}$, we define the *states* of $[t]$ to be the set $\text{states}_{\mathcal{A}/\mathcal{E}}([t]) \subseteq Q$ such that

$$\text{states}_{\mathcal{A}/\mathcal{E}}([t]) = \{\, p \in Q \mid \mathcal{A}/\mathcal{E} \vdash p(t) \,\}.$$

In the free case, a subset construction algorithm for regular tree automata [33], constructs the set $R_{\mathcal{A}} \subseteq \mathcal{P}(Q)$ of reachable states:

$$R_{\mathcal{A}} = \{\, P \subseteq Q \mid (\exists [t] \in T_{\mathcal{E}})\, \text{states}_{\mathcal{A}/\varnothing}([t]) = P \,\}.$$

By computing this set, we can decide if $\mathcal{L}_{\phi}(\mathcal{A}) \neq \varnothing$ by checking whether there is a set $P \in R_{\mathcal{A}}$ such that $P \models \phi$ — where $P \models \phi$ is defined inductively:

$$P \models \phi_1 \wedge \phi_2 \text{ iff } P \models \phi_1 \text{ and } P \models \phi_2 \qquad P \models \neg\phi \text{ iff } P \not\models \phi \qquad P \models p \text{ iff } p \in P.$$

For associative symbols, it does not seem possible to construct $R_{\mathcal{A}}$ directly. Let $\equiv_{\mathcal{A}/\mathcal{E}} \subseteq T_{\Sigma} \times T_{\Sigma}$ be the equivalence relation over terms where $t \equiv_{\mathcal{A}/\mathcal{E}} u$ iff. $\text{states}_{\mathcal{A}/\mathcal{E}}([t]) = \text{states}_{\mathcal{A}/\mathcal{E}}([u])$. For tree automata, the correctness of subset construction typically relies on the fact that $\equiv_{\mathcal{A}}$ is a congruence with respect to contexts. i.e. $s \equiv_{\mathcal{A}} t$ implies $C[s] \equiv_{\mathcal{A}} C[t]$ for all contexts C. However, this fact *does not* hold in the case when the root of s or t is an associative symbol $\bullet$ and the hole $\Box$ appearing in C appears immediately beneath f. Due to this complication, our subset construction algorithm for A and AC symbols maintains additional information.

Rather than the set of reachable sets of states $R_{\mathcal{A}}$, our subset construction algorithm constructs a set of reachable *profiles* of $\mathcal{A}$.

Definition 3.3.2. *Let* $\text{profile}_{\mathcal{A}/\mathcal{E}} : T_{\mathcal{E}} \to \Sigma \times \mathcal{P}(Q)$ *be the function such that:*

$$\text{profile}_{\mathcal{A}/\mathcal{E}}([t]) = (\text{root}(t), \text{states}_{\mathcal{A}/\mathcal{E}}([t])).$$

Since the only equations in $\mathcal{E}$ are associativity and commutativity axioms, each equivalence class $[t] \in T_{\mathcal{E}}$ has a unique root symbol which is the same for all terms $u \in [t]$. We let $D_{\mathcal{A}}$ denote the set of profiles of $T_{\mathcal{E}}$, that is

$$D_{\mathcal{A}} = \{\, \text{profile}_{\mathcal{A}/\mathcal{E}}([t]) \in \Sigma \times \mathcal{P}(Q) \mid [t] \in T_{\mathcal{E}} \,\}.$$

The set $D_{\mathcal{A}}$ is finite, because the total number of profiles is finite. However, when $\mathcal{E}$ contains associative symbols, $D_{\mathcal{A}}$ may not be computable. Otherwise, we could easily decide a propositional emptiness problem $\mathcal{L}_{\phi}(\mathcal{A})$ by computing $D_{\mathcal{A}}$, and checking whether there exists a pair $(f, P) \in D_{\mathcal{A}}$ such that $P \models \phi$.

Our approach is to incrementally construct $D_{\mathcal{A}}$. We start with the empty set $D_0 = \varnothing$, and apply inference rules until completion to form increasingly larger sets $D_1 \subseteq D_2 \subseteq \cdots \subseteq D_{\mathcal{A}}$. Before we can present the inference rules, we must present several definitions related to the different types of symbols in $\mathcal{E}$.

3.3.1 Free symbols

For each free symbol $f \in \Sigma$, we define a function states_f which computes the states of a term $f(t_1, \ldots, t_n)$ when the states for each term t_i are already known:

Definition 3.3.3. *Given a free symbol $f \in \Sigma$ with arity n, we define the function* $\text{states}_f : \mathcal{P}(Q)^n \to \mathcal{P}(Q)$ *so that for $P_1, \ldots, P_n \subseteq Q$,*

$$\text{states}_f(P_1, \ldots, P_n) = \{\, q \in Q \mid (\exists p_1 \in P_1, \ldots, p_n \in P_n)\ q(f(x_1, \ldots, x_n)) \Leftarrow p_1(x_1), \ldots, p_n(x_n) \in \mathcal{A} \,\}.$$

The following lemma relates $\text{states}_{\mathcal{A}/\mathcal{E}}$ and states_f:

Lemma 3.3.4. *For each term $t = f(t_1, \ldots, t_n)$ where $f \in \Sigma$ is free in $\mathcal{E}$,*

$$\text{states}_{\mathcal{A}/\mathcal{E}}(t) = \text{states}_f(P_1, \ldots, P_n)$$

where $P_i = \text{states}_{\mathcal{A}/\mathcal{E}}([t_i])$ for $i \in [1, n]$,.

Proof. We first show that if $p \in \text{states}_{\mathcal{A}/\mathcal{E}}([t])$, then $p \in \text{states}_f(P_1, \ldots, P_n)$. By Theorem 3.1.1 we know that $p \in \text{states}_{\mathcal{A}/\mathcal{E}}([t])$ iff there is a term $u =_{\mathcal{E}} t$ such that $\mathcal{A} \vdash p(u)$. Since u is equivalent modulo $\mathcal{E}$ to a term whose root symbol is the free symbol f, u must have the form $u = f(u_1, \ldots, u_n)$ with $u_i =_{\mathcal{E}} t_i$ for $i \in [1, n]$. It follows by the definition of states_f that $p \in \text{states}_f(P_1, \ldots, P_n)$.

On the other hand, if $p \in \text{states}_f(P_1, \ldots, P_n)$, then it is easy to show that $\mathcal{A}/\mathcal{E} \vdash p(f(t_1, \ldots, t_n))$. Consequently, $p \in \text{states}_{\mathcal{A}/\mathcal{E}}([t])$. □

3.3.2 A and AC symbols

For each symbol $\bullet \in \Sigma$ that is associative in $\mathcal{E}$ and possibly commutative, we define a context-free grammar $G(\bullet)$. Intuitively, the grammar captures inferences in the automaton $\mathcal{A}$ over *flattened* terms of the form $t_1 \bullet \cdots \bullet t_n$ where $\text{root}(t_i) \neq \bullet$ for $i \in [1, n]$.

Definition 3.3.5. *For an associative symbol $\bullet \in \Sigma$, $G(+)$ is the context free grammar with terminals $D(\bullet) = (\Sigma - \{\bullet\}) \times \mathcal{P}(Q)$, nonterminals Q, and production rules*

$$G(\bullet) = \{\, p := p_1 p_2 \mid p(x_1 \bullet x_2) \Leftarrow p_1(x_1), p_2(x_2) \in \mathcal{A} \,\} \cup \{\, p := (f, P) \mid (f, P) \in D(\bullet) \wedge p \in P \,\}.$$

Let G denote a context free grammar with terminals Σ and nonterminals Q. For each state $p \in Q$, we let $\mathcal{L}_p(G) \subseteq \Sigma^*$ denote the language generated from p using the rules in G. We let $\# : \Sigma^* \to \mathbb{N}^\Sigma$ denote the function that maps each string in Σ^* to the vector counting the number of occurrences of each terminal

symbol, and we let $\mathcal{S}_p(G) \subseteq \mathbb{N}^\Sigma$. This set is called the *Parikh image* [129] of p in G, and is defined as follows:

$$\mathcal{S}_p(G) = \{\, \#(w) \mid w \in \mathcal{L}_p(G) \,\}.$$

Our results in this section concern the relationship between the grammar $G(\bullet)$ and the automaton $\mathcal{A}$. We first show that parse trees in $G(\bullet)$ correspond to proofs in $\mathcal{A}/\mathcal{E}$:

Lemma 3.3.6. *For each term $t = t_1 \bullet \cdots \bullet t_n \in T_\Sigma$ where $\bullet$ is an associative symbol, and $\mathrm{root}(t_i) \neq \bullet$ for $i \in [1, n]$,*

$$\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_1]) \ldots \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_n]) \in \mathcal{L}_p(G(\bullet)) \implies \mathcal{A}/\mathcal{E} \vdash p(t).$$

Proof. As the production rules in $G(\bullet)$ come directly from the clauses in $\mathcal{A}$, this follows by structural induction on the parse tree used to show that

$$\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_1) \ldots \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_n])) \in \mathcal{L}_p(G(\bullet)).$$

□

In the other direction, we show the following result about $G(\bullet)$:

Lemma 3.3.7. *For each associative symbol $\bullet \in \Sigma$, if $\mathcal{A}/\mathcal{E} \vdash p(t)$, then there is a term $t_1 \bullet \cdots \bullet t_n =_\mathcal{E} t$ such that $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_1]) \ldots \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_n]) \in \mathcal{L}_p(G(\bullet))$ and $\mathrm{root}(t_i) \neq \bullet$ for $i \in [1, n]$.*

Proof. This follows by structural induction on the proof used to show $\mathcal{A}/\mathcal{E} \vdash p(t)$. □

If $\bullet \in \Sigma$ that is associative and not commutative, we can use Lemmas 3.3.6 and 3.3.7 to show the following characterization of $\mathcal{L}_p(G(\bullet))$.

Lemma 3.3.8. *For each term $t = t_1 \bullet \cdots \bullet t_n \in T_\Sigma$ where $\bullet \in \Sigma_\mathrm{A}$ and $\mathrm{root}(t_i) \neq \bullet$ for $i \in [1, n]$,*

$$\mathcal{A}/\mathcal{E} \vdash p(t) \iff \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_1]) \ldots \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_n]) \in \mathcal{L}_p(G(\bullet)).$$

Proof. By lemma 3.3.6,

$$\mathcal{A}/\mathcal{E} \vdash p(t) \to \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_1]) \ldots \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_n]) \in \mathcal{L}_p(G(\bullet)).$$

On the other hand, if $\mathcal{A}/\mathcal{E} \vdash p(t)$ then by Lemma 3.3.7, there is a term $u_1 \bullet \cdots \bullet u_m =_\mathcal{E} t$ such that $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([u_1]) \ldots \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([u_m]) \in \mathcal{L}_p(G(\bullet))$ and $\mathrm{root}(u_i) \neq \bullet$ for $i \in [1, m]$. As $t_1 \bullet \cdots \bullet t_n =_\mathcal{E} u_1 \bullet \cdots \bullet u_m$, and $\bullet$ only appears in an associativity equation, it follows that $m = n$ and $t_i =_\mathcal{E} u_i$ for all $i \in [1, n]$.

Consequently,

$$\text{profile}_{\mathcal{A}/\mathcal{E}}([t_1]) \dots \text{profile}_{\mathcal{A}/\mathcal{E}}([t_n]) \in \mathcal{L}_p(G(\bullet)).$$

□

From the previous lemma and the definition of $\mathcal{L}_P(G(\bullet))$, we can make the following observation:

Corollary 3.3.9. *For each term* $t = t_1 \bullet \cdots \bullet t_n \in T_\Sigma$ *where* $\bullet \in \Sigma_{\text{A}}$ *and* $\text{root}(t_i) \neq \bullet$ *for* $i \in [1, n]$*,*

$$\begin{aligned}\text{profile}_{\mathcal{A}/\mathcal{E}}(t) = (\bullet, P) \\ \iff n \geq 2 \wedge \text{profile}_{\mathcal{A}/\mathcal{E}}([t_1]) \dots \text{profile}_{\mathcal{A}/\mathcal{E}}([t_n]) \in \mathcal{L}_P(G(\bullet)).\end{aligned}$$

□

On the other hand, given a symbol $+$ that is both associative and commutative, then we can use Lemmas 3.3.6 and 3.3.7 to show the following characterization of $\mathcal{S}_p(G(+))$.

Lemma 3.3.10. *For each term* $t = t_1 + \cdots + t_n \in T_\Sigma$ *where* $+ \in \Sigma_{\text{AC}}$ *and* $\text{root}(t_i) \neq +$ *for* $i \in [1, n]$*,*

$$\mathcal{A}/\mathcal{E} \vdash p(t) \iff \#(\text{profile}_{\mathcal{A}/\mathcal{E}}([t_1]) \dots \text{profile}_{\mathcal{A}/\mathcal{E}}([t_n])) \in \mathcal{S}_p(G(+)).$$

Proof. By lemma 3.3.6,

$$\mathcal{A}/\mathcal{E} \vdash p(t) \rightarrow \text{profile}_{\mathcal{A}/\mathcal{E}}([t_1]) \dots \text{profile}_{\mathcal{A}/\mathcal{E}}([t_n]) \in \mathcal{L}_p(G(+)).$$

On the other hand, if $\mathcal{A}/\mathcal{E} \vdash p(t)$ then by Lemma 3.3.7, there is a term $u_1 + \cdots + u_m =_\mathcal{E} t$ such that $\text{profile}_{\mathcal{A}/\mathcal{E}}([u_1]) \dots, \text{profile}_{\mathcal{A}/\mathcal{E}}([u_m]) \in \mathcal{L}_p(G(+))$ and $\text{root}(u_i) \neq +$ for $i \in [1, m]$. As $t_1 + \cdots + t_n =_\mathcal{E} u_1 + \cdots + u_m$, and $+ \in \Sigma_{\text{AC}}$, it follows that $m = n$ and $\#(\text{profile}_{\mathcal{A}/\mathcal{E}}([u_1]), \dots, \text{profile}_{\mathcal{A}/\mathcal{E}}([u_m])) = \#(\text{profile}_{\mathcal{A}/\mathcal{E}}([t_1]), \dots, \text{profile}_{\mathcal{A}/\mathcal{E}}([t_n]))$. Consequently,

$$\#(\text{profile}_{\mathcal{A}/\mathcal{E}}([t_1]), \dots, \text{profile}_{\mathcal{A}/\mathcal{E}}([t_n])) \in \mathcal{S}_p(G(+)).$$

□

From this lemma and the definition of $\mathcal{S}_P(G(+))$, we can make the following observation:

Corollary 3.3.11. *For each term* $t = t_1 + \cdots + t_n \in T_\Sigma$ *where* $+ \in \Sigma_{\text{AC}}$ *and*

$\text{root}(t_i) \neq +$ *for* $i \in [1, n]$,

$$\begin{aligned}\text{profile}_{\mathcal{A}/\mathcal{E}}(t) = (+, P)\\ \iff n \geq 2 \wedge \#(\text{profile}_{\mathcal{A}/\mathcal{E}}([t_1]) \ldots \text{profile}_{\mathcal{A}/\mathcal{E}}([t_n])) \in \mathcal{S}_P(G(+)).\end{aligned}$$

□

For a subset $P \subseteq Q$ of nonterminals, we let $\mathcal{L}_P(G(\bullet))$ denote the strings appearing in the language $\mathcal{L}_q(G(\bullet))$ of a state $q \in Q$ iff $q \in P$. We similarly define $\mathcal{S}_P(G(+))$ to denote the vectors appearing in the Parikh image $\mathcal{S}_q(G(+))$ of a state $q \in Q$ iff $q \in P$. Specifically,

$$\begin{aligned}\mathcal{L}_P(G(\bullet)) &= \bigcap_{q \in P} \mathcal{L}_q(G(\bullet)) \; - \bigcup_{q \in (Q-P)} \mathcal{L}_q(G(\bullet)), \text{ and}\\ \mathcal{S}_P(G(+)) &= \bigcap_{q \in P} \mathcal{S}_q(G(+)) \; - \bigcup_{q \in (Q-P)} \mathcal{S}_q(G(+)).\end{aligned}$$

As context-free grammars are not closed under intersection and complementation, $\mathcal{L}_P(G(\bullet))$ is not in general a context-free language and checking emptiness of $\mathcal{L}_P(G(\bullet))$ is undecidable. In contrast, $\mathcal{S}_P(G(+))$ is a semi-linear set [129], and semi-linear sets have a decidable emptiness problem.

3.3.3 Constructing subsets

In our algorithm, we start with $D_0 = \varnothing$, and then compute D_{i+1} from D_i using the inference rules described in Figure 3.2. Each step adds a profile in $D_{\mathcal{A}}$ not in D_i. We let D_* denote the set obtained by applying the rules until completion. This set must exist, because the total number of profiles in $F \times \mathcal{P}(Q)$ is finite.

Our goal in the remainder of this section is to show that $D_* = D_{\mathcal{A}}$. We start by showing the following lemma.

Lemma 3.3.12. *Given $D_i \subseteq D_{\mathcal{A}}$, if D_{i+1} is obtained by an applying one of the rules in Figure 3.2, then $D_{i+1} \subseteq D_{\mathcal{A}}$.*

Proof. To prove this, we consider separately each of the possible rules that may be used to form D_{i+1}.

We first consider the rule for free symbols:

$$\frac{\text{choose free symbol } f \in \Sigma \text{ and } (f_1, P_1), \ldots, (f_n, P_n) \in D_i}{D_{i+1} := D_i \uplus \{\, (f, \text{states}_f(P_1, \ldots, P_n)) \,\}}$$

As $D_i \subseteq D_{\mathcal{A}}$, for each pair $(f_j, P_j) \in D_i$ with $j \in [1, n]$, there is an equivalence class $[t_j] \in T_{\mathcal{E}}$ such that $\text{profile}_{\mathcal{A}/\mathcal{E}}([t_j]) = (f_j, P_j)$. By Lemma 3.3.4, $\text{states}_f(P_1, \ldots, P_n) = \text{states}_{\mathcal{A}/\mathcal{E}}(f(t_1, \ldots, t_n))$. Clearly, $\text{root}(f(t_1, \ldots, t_n)) = f$, and thus $D_{i+1} \subseteq D_{\mathcal{A}}$.

We apply the rules below starting from the initial profile set $D_0 = \varnothing$ to construct D_{i+1} from D_i with the condition that a rule may only be applied if the resulting set D_{i+1} is a strict superset of D_i. The rules are applied until completion to obtain the set D_*.

$$\frac{\text{choose free symbol } f \in \Sigma \text{ and } (f_1, P_1), \ldots, (f_n, P_n) \in D_i}{D_{i+1} := D_i \uplus \{\, (f, \text{states}_f(P_1, \ldots, P_n)) \,\}}$$

$$\frac{\text{choose A symbol } \bullet \in \Sigma \text{ and } P \subseteq Q \text{ s.t. } D^{2+}_{i,\neg\bullet} \cap \mathcal{L}_P(G(\bullet)) \neq \varnothing}{D_{i+1} := D_i \uplus \{\, (\bullet, P) \,\}}$$

$$\frac{\text{choose AC symbol } + \in \Sigma \text{ and } P \subseteq Q \text{ s.t. } \mathbb{N}^{D_{i,\neg+},2+} \cap \mathcal{S}_P(G(+)) \neq \varnothing}{D_{i+1} := D_i \uplus \{\, (+, P) \,\}}$$

where for an A or AC symbol $+ \in \Sigma$, $D_{i,\neg+} = \{\, (f, P) \in D_i \mid f \neq + \,\}$, $D^{2+}_{i,\neg\bullet}$ denotes the strings over over $D_{i,\neg\bullet}$ containing at least two letters, and $\mathbb{N}^{D_{i,\neg+},2+}$ denotes vectors of natural numbers indexed by $D_{i,\neg+}$ whose elements sum up to at least 2.

Figure 3.2: Inference System for Constructing D_*

We next consider the rule for associative symbols:

$$\frac{\text{choose A symbol } \bullet \in \Sigma \text{ and } P \subseteq Q \text{ s.t. } D^{2+}_{i,\neg\bullet} \cap \mathcal{L}_P(G(\bullet))}{D_{i+1} := D_i \uplus \{\, (\bullet, P) \,\}}$$

As $D^{2+}_i \cap \mathcal{L}_P(G(\bullet)) \neq \varnothing$, there must be a string $d_1 \ldots d_n \in D^*_{i,\neg\bullet}$ such that $n \geq 2$ and $d_1 \ldots d_n \in \mathcal{L}_P(G(\bullet))$. As $D_i \subseteq D_{\mathcal{A}}$, for each profile D_j with $j \in [1, n]$, there is an equivalence class $[t_j] \in T_{\mathcal{E}}$ such that $\text{profile}_{\mathcal{A}/\mathcal{E}}([t_j]) = d_j$. Moreover, as d_j is in $D_{i,\neg\bullet}$, we know that $\text{root}([t_j]) \neq \bullet$. If we let $t = t_1 \bullet \cdots \bullet t_n$, it follows by Cor. 3.3.9 that $\text{profile}_{\mathcal{A}/\mathcal{E}}(t) = (\bullet, P)$. Consequently, $D_{i+1} \subseteq D_{\mathcal{A}}$.

Finally, we consider the rule for AC symbols:

$$\frac{\text{choose AC symbol } + \in \Sigma \text{ and } P \subseteq Q \text{ s.t. } \mathbb{N}^{D_{i,\neg+},2+} \cap \mathcal{S}_P(G(+)) \neq \varnothing}{D_{i+1} := D_i \uplus \{\, (+, P) \,\}}$$

As $\mathbb{N}^{D_i,2+} \cap \mathcal{S}_P(G(+)) \neq \varnothing$, there must be a string $d_1 \ldots d_n \in D^*_{i,\neg+}$ such that $n \geq 2$ and $\#(d_1 \ldots d_n) \in \mathcal{S}_P(G(+))$. As $D_i \subseteq D_{\mathcal{A}}$, for each profile D_j with $j \in [1, n]$, there is an equivalence class $[t_j] \in T_{\mathcal{E}}$ such that $\text{profile}_{\mathcal{A}/\mathcal{E}}([t_j]) = d_j$. Moreover, as d_j is in $D_{i,\neg+}$, we know that $\text{root}([t_j]) \neq +$. If we let $t = t_1 + \cdots + t_n$, it follows by Cor. 3.3.11 that $\text{profile}_{\mathcal{A}/\mathcal{E}}(t) = (+, P)$. Consequently, $D_{i+1} \subseteq D_{\mathcal{A}}$. □

The previous lemma can be used to show that D_* is a subset of $D_{\mathcal{A}}$. We use the next lemma to show that $D_{\mathcal{A}}$ is a subset of D_*.

Lemma 3.3.13. *For each term $t \in T_\Sigma$, $\text{profile}_{\mathcal{A}/\mathcal{E}}([t]) \in D_*$.*

Proof. We prove this by structural induction on the term t. We consider three separate cases, depending on whether the root symbol of t is a free symbol, an associative symbol, or an AC symbol.

We first consider the case where t has the form $f(t_1, \ldots, t_n)$ with f a free symbol. We let $\text{profile}_{\mathcal{A}/\mathcal{E}}([t_j]) = (f_j, P_j)$ for each $j \in [1, n]$, and note that $(f_j, P_j) \in D_*$ by our induction hypothesis. By Lemma 3.3.4,

$$\text{profile}_{\mathcal{A}/\mathcal{E}}([f(t_1, \ldots, t_n)]) = (f, \text{states}_f(P_1, \ldots, P_n)).$$

As the rules in Figure 3.2 can no longer be applied to D_*, we know that D_* contains $(f, \text{states}_f(P_1, \ldots, P_n))$. Thus $\text{profile}_{\mathcal{A}/\mathcal{E}}([t]) \in D_*$.

We next consider the case where t has the form $t_1 \bullet \cdots \bullet t_n$ where $\bullet$ is an associative symbol in Σ_{A}, $n \geq 2$ and $\text{root}(t_j) \neq \bullet$ for $j \in [1, n]$. We know that $\text{profile}_{\mathcal{A}/\mathcal{E}}([t]) = (\bullet, P)$ for some set $P \subseteq Q$. By our induction hypothesis, $\text{profile}_{\mathcal{A}/\mathcal{E}}([t_j]) \in D_*$ for $j \in [1, n]$, As $\text{root}([t_j]) \neq \bullet$, we know that $\text{profile}_{\mathcal{A}/\mathcal{E}}([t_j]) \in D_{*,\neg\bullet}$ for $j \in [1, n]$, Finally, by Cor. 3.3.9, we know that

$$\text{profile}_{\mathcal{A}/\mathcal{E}}([t_1]) \ldots \text{profile}_{\mathcal{A}/\mathcal{E}}([t_n]) \in D^{2+}_{i,\neg\bullet} \cap \mathcal{L}_P(G(\bullet)).$$

As the rules in Figure 3.2 can no longer be applied to D_*, we know that D_* contains (f, P), and thus $\text{profile}_{\mathcal{A}/\mathcal{E}}([t]) \in D_*$.

Finally, we next consider the case where t has the form $t_1 + \cdots + t_n$ where $+$ is an AC symbol in Σ_{AC}, $n \geq 2$ and $\text{root}(t_j) \neq +$ for $j \in [1, n]$. We know that $\text{profile}_{\mathcal{A}/\mathcal{E}}([t]) = (+, P)$ for some set $P \subseteq Q$. By our induction hypothesis, $\text{profile}_{\mathcal{A}/\mathcal{E}}([t_j]) \in D_*$ for $j \in [1, n]$, As $\text{root}([t_j]) \neq +$, we know that $\text{profile}_{\mathcal{A}/\mathcal{E}}([t_j]) \in D_{*,\neg+}$ for $j \in [1, n]$, Finally, by Cor. 3.3.11, we know that

$$\#(\text{profile}_{\mathcal{A}/\mathcal{E}}([t_1]) \ldots \text{profile}_{\mathcal{A}/\mathcal{E}}([t_n])) \in \mathbb{N}^{D_{i,\neg+},2+} \cap \mathcal{S}_P(G(+)).$$

As the rules in Figure 3.2 can no longer be applied to D_*, we know that D_* contains (f, P), and thus $\text{profile}_{\mathcal{A}/\mathcal{E}}([t]) \in D_*$. □

Together Lemmas 3.3.12 and 3.3.13 easily imply that $D_* = D_{\mathcal{A}}$, and thus assure of us the effectiveness of the inference system.

Theorem 3.3.14. *Let $\mathcal{E}$ be a theory with only free, A, and AC symbols, and let $\mathcal{A}$ be a regular $\mathcal{E}$-automaton. The set of profiles $D_* \subseteq F \times \mathcal{P}(Q)$ obtained from the rules in Figure 3.2 is the set of profiles $D_{\mathcal{A}}$, i.e.,*

$$D_* = D_{\mathcal{A}}.$$

□

When every associative symbol in $\mathcal{E}$ is commutative, the conditions in the rules are decidable, and thus the previous theorem implies that the propositional emptiness problem is decidable for AC-theories. The undecidability of regular

PTA with associative symbols crops up in testing the emptiness of $D^{2+}_{i,\neg\bullet} \cap \mathcal{L}_P(G(\bullet))$. The next section discusses a semialgorithm for solving this emptiness constraint.

3.3.4 Solving language equations for associativity

Since at present the emptiness testing with monotone rules for associative symbols is beyond the goal of our project, we have developed an approach that is likely to work well in practice for the *regular* case with associative symbols. Our approach rests on an interactive semi-algorithm for each associative symbol $+ \in \Sigma_{\mathrm{A}}$ which has access to the mapping D_i as it is being generated and performs two actions simultaneously: (1) recursively enumerates pairs (P, f) not in D_i for which $D(+)^{2+} \cap \mathcal{L}_P(G(+))$ is non-empty; and (2) applies machine learning techniques to attempt construction of a family $\{ \mathcal{M}_p \}_{p \in Q}$ of deterministic finite automata for which $\mathcal{L}(\mathcal{M}_p) = \mathcal{L}_p(G(+))$ for all $p \in Q$. If the first action succeeds, the semi-algorithm constructs the next D_{i+1} from D_i. If the second action succeeds, we can decide for each subset of P states, the condition $D(+)^{2+} \cap \mathcal{L}(G_{f,D_{\mathcal{A}}}(P)) = \varnothing$ in the rule (2). We then can either obtain D_{i+1} or prove that the conditional rule for $+$ can no longer be applied.

A naïve approach to the first action is quite simple. We recursively enumerate the strings in $D(+)^{2+}$ in order of increasing length to form the infinite sequence $w_1, w_2, \ldots$, and parse each string w_i to get the complete set of states $P_i = \{ p \in Q \mid w \in \mathcal{L}_p(G(+)) \}$. If $(f, P_i) \notin D_i$, then let $D_{i+1} = \{ (f, P_i) \} \cup D_i$. Handling the second action is more complicated. First, observe that we can enumerate the set of finite automata in order of increasing length. Because recursively enumerable sets are closed under finite products, we can even enumerate finite families of automata $\{ \mathcal{M}_p \}_{p \in Q}$. The difficult part then lies in checking whether $\mathcal{L}(\mathcal{M}_p) = \mathcal{L}_p(G(+))$ for all $p \in Q$. It is well known that given a *single* finite automaton $\mathcal{M}$ and a context-free grammar G, it is *undecidable* whether $\mathcal{L}(\mathcal{M}) = \mathcal{L}(G)$ [77, Theorem 8.12(3)]. However, this result is just for a single automaton, and does not imply the undecidability of our problem. Perhaps somewhat surprisingly, given a context-free grammar G with nonterminals Q in Chomsky normal form, and a family of automata $\{ \mathcal{M}_p \}_{p \in Q}$, the question whether $\mathcal{L}(\mathcal{M}_p) = \mathcal{L}_p(G(+))$ for all $p \in Q$ is decidable.

The decidability of this problem is a direct consequence of Theorem 2.3 in [5]. Before explaining that result, however, it is necessary to shift our perspective of context-free grammars from viewing them as collections of *production rules* to viewing them as systems of *language equations*.

Definition 3.3.15. *Let G be a context-free grammar with nonterminals Q and terminals Σ. The* system of equations *generated by G is the family of equations $\{ q = P_q \}_{q \in Q}$ in which for each nonterminal $q \in Q$, P_q is the formula $P_q = w_1 \mid \cdots \mid w_n$ where $q := w_1, \ldots, q := w_n$ are the production rules in G whose left-hand side equals q.*

Given a system of equations with nonterminals Q and terminals Σ, a *substitution* is a mapping $\theta : Q \to \mathcal{P}(\Sigma^*)$ associating each state $q \in Q$ to a language $\theta(q) \subseteq \Sigma^*$. A substitution θ can be *applied* to a language formula P, yielding a language $P\theta \subseteq \Sigma^*$ which is defined using the axioms:

$$P\theta = \begin{cases} \{\, a \,\} & \text{if } P = a \text{ for some } a \in \Sigma, \\ \theta(q) & \text{if } P = q \text{ for some } q \in Q, \\ S\theta \cup T\theta & \text{if } P = (S \,|\, T), \\ \{\, st \mid s \in S\theta \ \wedge \ t \in T\theta \,\} & \text{if } P = (S \,.\, T). \end{cases}$$

We may assume associativity of $|$ and $.$ in the above definition. Here $S \,.\, T$ denotes the concatenation of S and T. A substitution $\theta : Q \to \mathcal{P}(\Sigma^*)$ is a *solution* to the system of equations $\{\, q = P_q \,\}_{q \in Q}$ if and only if $\theta(q) = P_q\theta$ for all $q \in Q$. It is known that each system of equations generated by G has a *least solution*, namely $\theta_{\mathcal{L}} : q \in Q \mapsto \mathcal{L}_q(G)$. This solution is the least solution as $\theta_{\mathcal{L}}(q) \subseteq \psi(q)$ for all solutions $\psi : Q \to \mathcal{P}(\Sigma^*)$ and $q \in Q$. For grammars in Chomsky normal form, we can use the following theorem to help check whether an arbitrary solution is the least solution. Note that this is an easy consequence of Theorem 2.3 in [5].

Theorem 3.3.16. *If G is a context-free grammar in Chomsky normal form, then there is a* unique *solution θ to the system of equations generated by G in which $\epsilon \notin \theta(q)$ for any $q \in Q$.* □

In this theorem, ϵ denotes the empty string. As θ in the previous theorem is unique, it must be the least solution

Given a context-free grammar in Chomsky normal form G and a family of finite automata $\{\, \mathcal{M}_q \,\}_{q \in Q}$, we can use Theorem 3.3.16 to check whether $\mathcal{L}(\mathcal{M}_q) = \mathcal{L}_q(G)$ for all $q \in Q$.

Theorem 3.3.17. *Let G be a context-free grammar in Chomsky normal form with nonterminals Q. If $\mathcal{L}_q(G)$ is regular for all $q \in Q$, there is a constructable set of finite automata $\{\, \mathcal{M}_q \,\}_{q \in Q}$ for which $\mathcal{L}(\mathcal{M}_q) = \mathcal{L}_q(G)$.*

Proof. We recursively enumerate the families of finite automata $\{\, \mathcal{M}_q \,\}_{q \in Q}$ and check if $\mathcal{L}(\mathcal{M}_q) = \mathcal{L}_q(G)$ for each $q \in Q$. If we let $\psi : Q \to \mathcal{P}(\Sigma^*)$ be the substitution $q \mapsto \mathcal{L}(\mathcal{M}_q)$, then the problem of checking whether $\mathcal{L}(\mathcal{M}_q) = \mathcal{L}_q(G)$ for all $q \in Q$ reduces to deciding whether ψ is the unique solution satisfying Theorem 3.3.16. For each equation $q = P_q$, we can construct the automaton $\mathcal{M}_{P_q}$ with $\mathcal{L}(\mathcal{M}_{P_q}) = P_q\psi$ due to the effective closure of regular languages under union and concatenation. Moreover, one can check whether $\mathcal{L}(\mathcal{M}_q) = \mathcal{L}(\mathcal{M}_{P_q})$ for each $q \in Q$ using the standard approaches for testing the equivalence of finite automata. So clearly we can check whether ψ is a solution. But it is also trivial to check whether $\epsilon \notin \mathcal{L}(\mathcal{M}_q)$ for each $q \in Q$. Thus it is decidable whether

ψ satisfies the conditions in Theorem 3.3.16. If it does, then $\psi(q)$ must equal $\mathcal{L}_q(G)$ for each $q \in Q$. □

The key problem discussed in this section is determining whether the language $\mathcal{L}_q(G)$ is regular for each nonterminal $q \in Q$. One would expect this problem to be undecidable. Surprisingly, despite searching several texts, we could not find a decidability result for this problem. If $\mathcal{L}_q(G)$ is regular for each nonterminal $q \in Q$, Theorem 3.3.17 shows that we can always show that by generating an equivalent family of finite automata. The other case is not so clear. Undecidability results for context-free languages such as Greibach's theorem [77, Section 8.7] do not apply, since they concern single context-free languages and this property concerns every nonterminal in a grammar. Theorem 3.3.17's result itself relied heavily upon the assumption that every nonterminal generates a regular language. The same approach does not work to construct a finite automata corresponding to a single nonterminal in G due to the undecidability of the equivalence problem for context-free grammars and regular languages.

3.3.5 Angluin's algorithm

Though technically sound, if one were to implement the semi-algorithm using the naïve approach outlined in the previous section, the efficiency would be quite poor. Enumerating finite automata in order of increasing size takes exponential time relative to the size of the automaton. Each family of finite automata would need to be checked for equivalence, and this also takes exponential time. Unfortunately, we don't see a way to improve the exponential time required to check equivalence, but by applying techniques from learning theory, we decrease the number of equivalence queries we make so that if the algorithm eventually succeeds, we will have only required a polynomial number of queries relative to the size of the accepting family of automata eventually found.

A well-known algorithm in machine learning is Angluin's algorithm [3] for learning regular languages with oracles. For an arbitrary language L, this algorithm attempts to construct a finite automaton $\mathcal{M}$ such that $\mathcal{L}(\mathcal{M}) = L$ by asking questions to two oracles: a *membership* oracle that answers whether a string $u \in \Sigma^*$ is in L; an *equivalence* oracle that answers whether $\mathcal{L}(\mathcal{M}) = L$ and if not, provides a *counterexample* string $u \in \Sigma^*$ in the symmetric difference of L and $\mathcal{L}(\mathcal{M})$, i.e. $u \in L \oplus \mathcal{L}(\mathcal{M})$ with $L \oplus \mathcal{L}(\mathcal{M}) = (L - \mathcal{L}(\mathcal{M})) \cup (\mathcal{L}(\mathcal{M}) - L)$. Angluin's algorithm will terminate only if L is regular. However, given the appropriate oracles, one can attempt to apply it with any language, even languages not known to be regular. We roughly sketch below how Angluin's algorithm works. Readers are recommended to consult [92] for further details.

First we recall the definition of *Nerode's right congruence*: given a language $L \subseteq \Sigma^*$, the equivalence relation $\sim_L$ over Σ^* is the relation such that for $u, v \in \Sigma^*$, $u \sim_L v$ if and only if for all $w \in \Sigma^*$, $uw \in L \iff vw \in L$. It is known that a language L is regular if and only if the number of equivalence

classes $|\Sigma^*/\!\sim_L|$ is finite. Angluin's algorithm maintains a data structure that stores two constructs: (1) a finite set $S \subseteq \Sigma^*$ of strings, each belonging to a distinct equivalence class in $\Sigma^*/\!\sim_L$, and (2) a finite set $D \subseteq \Sigma^*$ of distinguishing strings which in conjunction with the membership oracle, allows the algorithm to classify an arbitrary string into one of the known equivalence classes.

Initially, $S = \{ \epsilon \}$ and $D = \varnothing$. Using the membership oracle in conjunction with S and D, the algorithm constructs a deterministic finite automaton $\mathcal{M}$ such that $\mathcal{L}(\mathcal{M}) = L$ when $S = \Sigma^*/\!\sim_L$. The algorithm then queries the equivalence oracle which either succeeds and we are done, or returns a counterexample which can be analyzed to reveal at least one additional equivalence class representative in $\Sigma^*/\!\sim_L$ that is not in S. If L is regular, eventually the algorithm will learn all of the equivalence classes in $\Sigma^*/\!\sim_L$. If L is not regular, $\Sigma^*/\!\sim_L$ must be infinite and so the algorithm will not terminate.

Given a finite family of regular languages $\{ L_q \}_{q \in Q}$, Angluin's algorithm can be easily generalized to simultaneously learn a finite family of automata $\{ \mathcal{M}_q \}_{q \in Q}$ such that $\mathcal{L}(\mathcal{M}_q) = L_q$ for all $q \in Q$. In this version, there must be a membership oracle for each language L_q, and an equivalence oracle which given a family $\{ \mathcal{M}_q \}_{q \in Q}$, returns true if $L_q = \mathcal{L}(\mathcal{M}_q)$ for all $q \in Q$, or a pair (q, u) where $q \in Q$, and u is a counterexample in $L_q \oplus \mathcal{L}(\mathcal{M}_q)$. The generalized algorithm will terminate when L_q is regular for each $q \in Q$.

In the context of this paper, we use Angluin's algorithm in conjunction with the grammar $G(+)$ over the terminals D_i and nonterminals Q. The algorithm attempts to construct a family of finite automata $M = \{ \mathcal{M}_q \}_{q \in Q}$ for which $\mathcal{L}(\mathcal{M}_q) = D_i^{2+} \cap \mathcal{L}_q(G(+))$. If the process succeeds, we can easily determine whether $D_i^{2+} \cap \mathcal{L}_P(G(+)) = \varnothing$ for each profile $d \in \Sigma \times \mathcal{P}(Q) \setminus D_i$ using standard techniques for finite automata. If we discover that $D_i^{2+} \cap \mathcal{L}_P(G(+)) \neq \varnothing$, we set $D_{i+1} := D_i \uplus \{ (P, +) \}$ and repeat the process for D_{i+1}.

To apply Angluin's algorithm, we need to provide the membership and equivalence oracles needed for a context-free grammar G with nonterminals Q and terminals Σ. The membership oracle for each nonterminal $q \in Q$ is implemented by a context-free language parser that parses a string $u \in \Sigma^*$ and returns true if $u \in \mathcal{L}_q(G)$. Given the family $\{ \mathcal{M}_q \}_{q \in Q}$, our equivalence oracle forms the mapping $\theta : q \mapsto \mathcal{L}(\mathcal{M}_q)$ and checks if it is the solution to the equations generated by G satisfying Theorem 3.3.16. If θ is not the solution, the equivalence oracle must analyze the mapping to return a counterexample. The algorithm we use is presented in Figure 3.3. Correctness of the oracle is shown in the following theorem:

Theorem 3.3.18. *Given a context-free grammar G in Chomsky normal form with nonterminals Q and terminals Σ, and a family $\{ \mathcal{M}_q \}_{q \in Q}$ of finite automata over Σ, the algorithm* check_equiv *in Figure 3.3*

- *returns* true *if $\mathcal{L}_q(G) = \mathcal{L}(\mathcal{M}_q)$ for all $q \in Q$; and otherwise,*
- *returns a pair (q, w) such that $w \in \mathcal{L}_q(G) \oplus \mathcal{L}(\mathcal{M}_q)$.*

PROCEDURE check_equiv

INPUT G : a CFG with terminals Σ and nonterminals Q
$\{\mathcal{M}_q\}_{q \in Q}$: a family of finite automata over Σ

OUTPUT true or (q, u) for some $q \in Q$ and $u \in \Sigma^*$

```
let θ be the substitution q ↦ L(M_q);
for each q ∈ Q do
  if ε ∈ L(M_q) then return (q, ε);

  if L(M_q) ≠ P_Q θ then
    choose u ∈ L(M_q) ⊕ P_q θ
    if u ∈ L(M_q) ⊕ L_q(G) then return (q, u)
    else
      for each q := pp' ∈ P and u = st do
        if s ∈ L(M_p) ⊕ L_p(G) then return (p, s);
        if t ∈ L(M_p') ⊕ L_p'(G) then return (p', t)
      od;
od;
return true
```

Figure 3.3: Checking language equivalence

Proof. It is easy to verify that the procedure terminates and that the pair returned by each return statement is indeed a counterexample. The non-trivial part of this theorem is that if the outer loop terminates without returning a pair, check_equiv should return true. This property is obtained by showing that if $\mathcal{L}(\mathcal{M}_q) \neq \mathcal{L}_q(G)$ for some $q \in Q$ executed by the outer loop, then the body of the loop is guaranteed to return a pair.

The string $u \in \Sigma^*$ chosen in the body is in the symmetric difference of $\mathcal{L}(\mathcal{M}_q)$ and $P_q\theta$. If $u \in \mathcal{L}(\mathcal{M}_q) \oplus \mathcal{L}_q(G)$ (or vice versa), then the body returns (q, u). Otherwise, if $u \in \mathcal{L}(\mathcal{M}_q) \iff u \in \mathcal{L}_q(G)$, then then $u \in P_q\theta \oplus \mathcal{L}_q(G)$.

Let ψ be the substitution $q \mapsto \mathcal{L}_q(G)$. Since ψ is a solution to the equations generated by G, $\mathcal{L}_q(G) = \psi(q) = P_q\theta$. So $u \in P_q\theta \oplus P_q\psi$. We will show that the inner for loop must return a value when $u \in P_q\theta - P_q\psi$ — the proof in the other case when $u \in P_q\psi - P_q\theta$ is similar.

If the rules in G whose left-hand-side is q are $q := p_1p'_1, \ldots, q := p_np'_n$, then P_q is of the form $P_q = p_1p'_1 \mid \cdots \mid p_np'_n$. So $u \in P_q\theta$ implies that $u \in (p_ip'_i)\theta$ for some i. Likewise, as $u \notin P_q\psi$ and $p_ip'_i\psi \subseteq P_q\psi$, it easily follows that $u \notin (p_ip'_i)\psi$. Since $u \in (p_ip'_i)\theta$, we can partition it into strings $s, t \in \Sigma^*$ such that $u = st$, $s \in \theta(p_i)$, and $t \in \theta(p'_i)$. In addition, since $u = st$ and $u \notin (p_ip'_i)\psi$, either $s \notin \psi(p_i)$ or $t \notin \psi(p'_i)$. Thus by the definition of ψ, there is a rule $q := p_ip'_i$ in P and strings $s, t \in \Sigma^*$ such that $u := st$ and either $s \in \mathcal{L}(\mathcal{M}_{p_i}) - \mathcal{L}_{p_i}(G))$ or $t \in \mathcal{L}(\mathcal{M}_{p'_i}) - \mathcal{L}_{p'_i}(G)$. A similar argument in this case where $u \in P_q\psi - P_q\theta$ shows that the inner loop will always return a pair when executed. □

When equipped with context-free language parsers as membership oracles and check_equiv as an equivalence oracle, Angluin's algorithm accomplishes the same goal as the simple enumeration-based algorithm used to prove Theorem 3.3.17. However, this approach reduces the complexity from double to single exponential time. In searching for a solution, the enumeration algorithm used in Theorem 3.3.17 checks equivalence of every family of finite automata in order of increasing size. The total number of equivalence checks will be exponential relative to the size of the final output. Since each equivalence check itself takes exponential time, the enumeration algorithm takes double exponential time relative to the size of the final output. In contrast, Angluin's algorithm makes a number of oracle queries that is polynomial [3] to the size of the final output. The equivalence oracle itself takes exponential time, and so the total time of the new algorithm is a single exponential relative to the size of the final output.

3.3.6 CETA library

The tree automata techniques developed in this paper are not only for theoretical use. The emptiness checking techniques for associative and AC symbols explained in this section have been implemented in the CETA library [69]. This library is a complex C++ library with approximately 10 thousand lines of code. Emptiness checking is performed by the subset construction algorithm extended with support for associative and commutativity axioms described previously. The reason that CETA is so large is that the subset construction algorithm relies on quite complex algorithms on context free grammars, semilinear sets, and finite automata. Although the theoretical worst-case complexity is quite high, we have found that CETA performs quite well in the sufficient completeness checker described in Chapter 5. The reason seems to be that most data structures such as lists, trees, and multisets already have a highly regular structure, and the number of sets of states constructed by our algorithm does not grow very large. Most sufficient completeness problems can be verified in seconds.

This software provides the function for emptiness checking with not only associativity and commutativity axioms, but identity axioms as well. The identity axiom for a function symbol + with a unit symbol 0 is the equations of the forms $0 + x = x$ and $x + 0 = x$. In CETA, identity axioms in a propositional tree automaton are converted into the rewrite rules $x + 0 \to x$ and $0 + x \to x$ in conjunction with a specialized Knuth-Bendix style completion procedure modulo associativity and commutativity that preserves the set of reachable states for each term.

Though still a prototype, CETA has additionally been integrated to work with the reachability analysis tool ACTAS [126], as well as the next generation sufficient completeness tool for Maude which is described later in Chapter 5. In a future project, we plan to apply the new ACTAS to tree automata-based

verification of infinite state systems, including network protocols.

3.4 AC intersection free propositional emptiness

Having shown that intersection emptiness is undecidable in general for equational tree automata over a theory $\mathcal{E}$ with AC and ACI symbols earlier in Section 3.2, in this section we search for a restricted subclass of equational tree automata over $\mathcal{E}$ for which not only is intersection emptiness decidable, but so is the propositional emptiness problem. Our search for this class began by trying to eliminate the main culprit that led to the undecidability result in Cor. 3.2.2 — the ability of clauses with ACI symbols to simulate the intersection clauses of an alternating AC-tree automata.

The solution we have found is to subject each ACI symbol $\circ$ in $\mathcal{E}$ to one of two constraints: (1) either the clauses in the automaton where $\circ$ appears must satisfy certain syntactic restrictions explained below; or (2) the idempotence equation $x \circ x = x$ in $\mathcal{E}$ must be treated as a rewrite rule $x \circ x \to x$ as in the *tree automata with normalization* framework of [124]. We first define the syntactic restrictions:

Definition 3.4.1. *Let $\mathcal{E}$ be an equational theory in which each symbol is AC, ACI, or free. A regular $\mathcal{E}$-tree automaton $\mathcal{A}$ is* AC intersection free *iff for each clause in $\mathcal{A}$ with the form $p(x_1 \circ x_2) \Leftarrow p_1(x_1), p_2(x_2)$ where $\circ \in \Sigma$ is an ACI symbol, it is the case that for all $q_1, q_2 \in Q$, and AC or ACI symbols $+ \neq \circ$,*

$$p_1(x_1 + x_2) \Leftarrow q_1(x_1), q_2(x_2) \in \mathcal{A} \implies p(x_1 + x_2) \Leftarrow q_1(x_1), q_2(x_2) \in \mathcal{A}.$$

The intuition behind this definition is that if an intersection clause $p(x) \Leftarrow p_1(x), p_2(x)$ is entailed by a clause $p(x_1 \circ x_2) \Leftarrow p_1(x_1), p_2(x_2)$ with an ACI symbol $\circ$, then we can disregard it in considering terms whose root symbol is an AC or ACI symbol $+ \neq \circ$. In the next section, see Lemma 3.4.11 to see how our definition is used technically.

One important observation is that AC intersection free automata are closed under disjoint unions — that is given two AC intersection free $\mathcal{E}$-tree automata $\mathcal{A}$ and $\mathcal{B}$ such that the states have been renamed so that the states in $\mathcal{A}$ and $\mathcal{B}$ are disjoint, the union $\mathcal{E}$-tree automaton $\mathcal{C} = \mathcal{A} \cup \mathcal{B}$ is also AC intersection free. Moreover, $\mathcal{L}_p(\mathcal{A}/\mathcal{E}) = \mathcal{L}_p(\mathcal{C}/\mathcal{E})$ for each state p in $\mathcal{A}$, and $\mathcal{L}_q(\mathcal{B}/\mathcal{E}) = \mathcal{L}_q(\mathcal{C}/\mathcal{E})$ for each state q in $\mathcal{A}$. Since we will soon show that the propositional emptiness problem is decidable for AC intersection free automata, it follows that the emptiness of an arbitrary Boolean combination of AC intersection free tree languages is decidable even if the languages are defined in different automata.

This syntactic restriction may be too strong in some applications, and so we also study a different approach to handling idempotence equations that is suggested by the tree automata with normalization framework of [124]. A *tree*

automaton with normalization (TAN) $\mathcal{A}$ is equipped with a rewrite system $\mathcal{R}$ that is confluent and terminating modulo an equational theory $\mathcal{E}$. A term t is accepted by TAN $\mathcal{A}$ if its normal form $[t \downarrow_{\mathcal{R}/\mathcal{E}}]$ is in the underlying equational tree language $\mathcal{L}(\mathcal{A}/\mathcal{E})$. This framework borrows the fundamental idea in term rewriting, namely that some of the equations in a theory $\mathcal{E}'$ are best handled by orienting them as rewrite rules in a rewrite system $\mathcal{R}$ in a way so that $\mathcal{R}$ is confluent and terminating modulo the remaining equations $\mathcal{E} \subseteq \mathcal{E}'$. As $\mathcal{R}$ is terminating and confluent modulo $\mathcal{E}$, the language is closed with respect to both the equations in $\mathcal{E}$ and the equations obtained from the rules in $\mathcal{R}$.

Our interest in the TAN framework stems from the fact that if $\mathcal{R}_{\mathrm{I}}$ is a rewrite system containing idempotence rules $f(x, x) \to x$ for some of the AC symbols in a theory $\mathcal{E}$ with free, AC, and ACI symbols, then $\mathcal{R}_{\mathrm{I}}$ is confluent and terminating modulo $\mathcal{E}$. This suggests that as an alternative to the restrictions in Def. 3.4.1, we can treat some of the idempotence equations as rules, and still have a class of tree automata closed modulo both the equations in $\mathcal{E}$ and the underlying equations in $\mathcal{R}_{\mathrm{I}}$. By handling the idempotence equations as rules, we avoid the problem of simulating intersection clauses, because that simulation relies on applying idempotence in the direction $x \to x + x$.

By requiring that each ACI symbol either satisfies the syntactic constraints in the definition of AC intersection free automata, or treats the idempotence equation as a rule as in the tree automata with normalization approach, we describe an algorithm in the next section whose correctness implies the following:

Theorem 3.4.2. *Let $\mathcal{E}$ be a theory with free, AC, and ACI symbols, and let $\mathcal{R}_{\mathrm{I}}$ be a set of rewrite rules which may contain an idempotence rule for any of the AC symbols in $\mathcal{E}$.*

For each AC intersection free $\mathcal{E}$-tree automaton $\mathcal{A}$, and propositional formula ϕ over the states in $\mathcal{A}$, the following problem is decidable:

$$\mathcal{L}_{\phi}(\mathcal{A}/\mathcal{E}) \cap \mathrm{Can}_{\mathcal{R}_{\mathrm{I}}/\mathcal{E}} = \varnothing.$$

In other words, we can decide whether the language $\mathcal{L}_{\phi}(\mathcal{A}/\mathcal{E})$ contains an $\mathcal{R}_{\mathrm{I}}/\mathcal{E}$-irreducible equivalence class $[t] \in \mathrm{Can}_{\mathcal{R}_{\mathrm{I}}/\mathcal{E}}$. This theorem simultaneously settles two open questions:

The first open question is the emptiness problem for tree automata with normalization over an equational theory $\mathcal{E}_{\mathrm{AC}}$ with free and AC symbols and a rewrite system $\mathcal{R}_{\mathrm{I}}$ containing idempotence equations for the AC symbols in $\mathcal{E}_{\mathrm{AC}}$. Specifically, we want to decide whether $\mathrm{Can}_{\mathcal{R}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}} \cap L_p(\mathcal{A}/\mathcal{E}_{\mathrm{AC}}) = \varnothing$ for each $\mathcal{E}_{\mathrm{AC}}$-tree automaton $\mathcal{A}$ and state p in $\mathcal{A}$. The problem was mentioned in [124], but left unsolved. Theorem 3.4.2 solves this problem, because $\mathcal{E}_{\mathrm{AC}}$ contains no ACI symbols and thus every $\mathcal{E}_{\mathrm{AC}}$-tree automaton is AC intersection free. One observation made in [124] is that the decidability of the emptiness problem for tree automata with normalization only depends on the left hand

sides of the rules in $\mathcal{R}$. It follows that if the emptiness problem is decidable when $\mathcal{R}$ contains idempotence rules $x + x \to x$, it is also decidable when $\mathcal{R}$ contains nilpotence rules $x + x \to 0$.

The second open question settled by Theorem 3.4.2 is the problem of deciding the propositional emptiness of equational tree automata over a theory $\mathcal{E}_{\text{ACI}}$ with a single ACI symbol and free symbols. This problem is interesting, because equational tree automata over $\mathcal{E}_{\text{ACI}}$ are not closed under complementation [137], and so the propositional emptiness problem is not reducible to the regular emptiness problem in this theory. Theorem 3.4.2 solves this problem, because $\mathcal{E}_{\text{ACI}}$ contains only a single ACI symbol, and thus every $\mathcal{E}_{\text{ACI}}$-tree automaton is AC intersection free. Solving the propositional emptiness problem also shows that both subsumption ($\mathcal{L}_p(\mathcal{A}/\mathcal{E}_{\text{ACI}}) \subseteq \mathcal{L}_q(\mathcal{B}/\mathcal{E}_{\text{ACI}})$) and universality ($\mathcal{L}_p(\mathcal{A}/\mathcal{E}_{\text{ACI}}) = T_{\mathcal{E}_{\text{ACI}}}$) are decidable. Both problems appear to be open. Additionally, since intersection emptiness is undecidable for equational tree automata over $\mathcal{E}_{\text{AC}} \cup \mathcal{E}_{\text{ACI}}$ due to Cor. 3.2.2, it follows that propositional emptiness over $\mathcal{E}_{\text{AC}} \cup \mathcal{E}_{\text{ACI}}$ is undecidable as well. However, propositional emptiness is decidable for $\mathcal{E}_{\text{AC}}$ [125] and implied to be decidable for $\mathcal{E}_{\text{ACI}}$ by Theorem 3.4.2. It follows that propositional emptiness is also a *non-modular property* for the combination of disjoint theories.

3.4.1 Profile graphs

In this section, we define an algorithm that solves the decision problem posed in Theorem 3.4.2. We begin with a discussion of our overall approach, and how any solution to check the *emptiness* of a regular equational tree language over a theory containing idempotence axioms appears to also require being able to compute the *size* of a language. We then present results about terms whose root is a free symbol in Section 3.4.2, and present results abouts terms whose root is an AC or ACI symbol in Section 3.4.3. In Section 3.4.4, we present our function for estimating the number of distinct equivalence classes that reach a particular profile. Finally, in Section 3.4.5, we present the algorithm itself, and verify its correctness.

For this section, $\mathcal{E} = (F, E)$ denotes a theory in which each symbol is AC, ACI, or free, $\mathcal{R}_{\text{I}}$ denotes a rewrite system where the only axioms are idempotence rules of the form $x+x \to x$ for an AC symbol $+ \in \Sigma$, and $\mathcal{A}$ denotes a regular AC intersection free $\mathcal{E}$-tree automaton with states Q. It is sometimes useful to treat all of the idempotence equations as rules. We let $\mathcal{E}_{\text{AC}} \subseteq \mathcal{E}$ denote the theory containing only the AC equations in $\mathcal{E}$, and we let $\hat{\mathcal{R}}_{\text{I}}$ denote the rewrite system containing the rules in $\mathcal{R}_{\text{I}}$ as well as a rule $x+x \to x$ for each equation $x+x = x$ in $\mathcal{E}$. $\hat{\mathcal{R}}_{\text{I}}$ is terminating and confluent modulo $\mathcal{E}_{\text{AC}}$, so for all $\mathcal{R}_{\text{I}}/\mathcal{E}$-irreducible terms $t, u \in T_\Sigma$, $t =_{\mathcal{E}} u$ iff $t \downarrow_{\hat{\mathcal{R}}_{\text{I}}/\mathcal{E}_{\text{AC}}} =_{\mathcal{E}_{\text{AC}}} u \downarrow_{\hat{\mathcal{R}}_{\text{I}}/\mathcal{E}_{\text{AC}}}$. For all $[t], [u] \in \text{Can}_{\mathcal{R}_{\text{I}}/\mathcal{E}}$, we say that $[t]$ is a *flattened subterm* of $[u]$, denoted $[t] \trianglelefteq_{\text{flat}} [u]$, if either:

- $u \downarrow_{\hat{\mathcal{R}}_{\text{I}}/\mathcal{E}_{\text{AC}}} =_{\mathcal{E}_{\text{AC}}} f(u_1, \ldots, u_n)$ with f a free symbol and $t \downarrow_{\hat{\mathcal{R}}_{\text{I}}/\mathcal{E}_{\text{AC}}} =_{\mathcal{E}_{\text{AC}}} u_i$

for some $i \in [1,n]$, or

- $u \downarrow_{\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}} =_{\mathcal{E}_{\mathrm{AC}}} u_1 + \cdots + u_n$ with $+$ an AC or ACI symbol, $n \geq 2$, $\mathrm{root}(u_i) \neq +$ for all $i \in [1,n]$, and $t \downarrow_{\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}} =_{\mathcal{E}_{\mathrm{AC}}} u_j$ for some $j \in [1,n]$.

Our algorithm is similar to the subset construction algorithm in [33] for determinizing a regular tree automata. For each equivalence class $[t] \in T_{\mathcal{E}}$, the *profile* of $[t]$, denoted $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t])$, is a pair that contains all the information about $[t]$ relevant to the algorithm.

Definition 3.4.3. *Let* $\mathrm{profile}_{\mathcal{A}/\mathcal{E}} : T_{\mathcal{E}} \to \Sigma \times \mathcal{P}(Q)$ *be the function such that:*

$$\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t]) = (\mathrm{root}(t \downarrow_{\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}}), \mathrm{states}_{\mathcal{A}/\mathcal{E}}([t])).$$

where $\mathrm{states}_{\mathcal{A}/\mathcal{E}}([t]) = \{\, p \in Q \mid \mathcal{A}/\mathcal{E} \vdash p(t) \,\}$.

Note that $\mathrm{root}(t \downarrow_{\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}})$ is uniquely determined as $\mathcal{E}_{\mathrm{AC}}$ only contains associativity and commutativity axioms which do not change the root symbol of a term. We will later show in Section 3.4.2 (Lemma 3.4.7) and Section 3.4.3 (Lemma 3.4.14) how to compute $\mathrm{states}_{\mathcal{A}/\mathcal{E}}([t])$ and $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t])$ when $\mathcal{A}$ is intersection free and $\mathcal{E}$ contains AC, ACI, and free symbols.

Given an automaton $\mathcal{B}$ over a theory $\mathcal{E}'$ with associative and AC symbols Σ', the subset algorithm in Section 3.3 constructed the set

$$D_{\mathcal{B}} = \{\, (f,P) \in \Sigma' \times \mathcal{P}(Q') \mid (\exists [t] \in T_{\mathcal{E}'})\, \mathrm{root}([t]) = f \wedge \mathrm{states}_{\mathcal{B}/\mathcal{E}'}([t]) = P \,\}.$$

By computing this set, we can decide if $\mathcal{L}_{\phi}(\mathcal{B}/\mathcal{E}') \neq \varnothing$ by checking for a profile $(f,P) \in \det(\mathcal{B})$ such that $P \models \phi$ where $P \models \phi$ is defined inductively:

$$P \models \phi_1 \wedge \phi_2 \text{ iff } P \models \phi_1 \text{ and } P \models \phi_2 \qquad P \models \neg\phi \text{ iff } P \not\models \phi \qquad P \models p \text{ iff } p \in P$$

For solving the problem in Theorem 3.4.2, this approach is inadequate for two reasons: (1) We want to decide whether $\mathrm{Can}_{\mathcal{R}_{\mathrm{I}}/\mathcal{E}} \cap \mathcal{L}_{\phi}(\mathcal{A}/\mathcal{E}) = \varnothing$ rather than deciding whether $\mathcal{L}_{\phi}(\mathcal{A}/\mathcal{E}) = \varnothing$. (2) Both $\mathcal{E}$ and $\mathcal{R}$ may contain idempotence axioms, and idempotence appears to require constructing a structure which in addition to enable checking if there *exists* a term with a particular profile, also enables checking *how many* distinct terms have that profile. We illustrate this with an example. Let $\mathcal{E}_{\mathrm{ACI}}$ be the theory containing an ACI symbol $\circ$ and constants a, b, and c, and let $\mathcal{B}$ be the $\mathcal{E}_{\mathrm{ACI}}$-tree automaton with the rules:

$$p_1(a) \quad p_1(b) \quad p_2(x_1 \circ x_2) \Leftarrow p_1(x_1), p_1(x_2) \quad p_3(x_1 \circ x_2) \Leftarrow p_1(x_1), p_2(x_2).$$

In this automaton, one can observe that

$$\mathcal{L}_{p_2}(\mathcal{B}/\mathcal{E}_{\mathrm{ACI}}) = \mathcal{L}_{p_3}(\mathcal{B}/\mathcal{E}_{\mathrm{ACI}}) = \{\, [a], [b], [a \circ b] \,\},$$

and consequently $\mathcal{L}_{p_3 \wedge \neg p_2}(\mathcal{B}/\mathcal{E}_{\mathrm{ACI}}) = \varnothing$. Now consider the automaton $\mathcal{B}'$ con-

taining the clauses in $\mathcal{B}$ and the additional clause $p_1(c)$. One can observe that $\mathcal{L}_{p_3 \wedge \neg p_2}(\mathcal{B}'/\mathcal{E}_{\text{ACI}}) = \{\, [a \circ b \circ c] \,\}$. The language $\mathcal{L}_{p_3 \wedge \neg p_2}(\mathcal{B}'/\mathcal{E}_{\text{ACI}})$ is not empty, because there are 3 distinct elements in $\mathcal{L}_{p_1}(\mathcal{B}'/\mathcal{E}_{\text{ACI}})$, whereas $\mathcal{L}_{p_1}(\mathcal{B}/\mathcal{E}_{\text{ACI}})$ only contains 2 elements. If we generalize this idea, it is not difficult to show that for any positive integer $n \in \mathbb{N}$ and tree automaton $\mathcal{B}$ over $\mathcal{E}$ with a state p, we can construct a tree automaton $\mathcal{B}'_n$ over the theory $\mathcal{E}'$ containing $\mathcal{E}$ as well as a fresh ACI symbol $\circ$ and a formula ϕ_n over the states in $\mathcal{B}'_n$ such that

$$\mathcal{L}_{\phi_n}(\mathcal{B}'_n/\mathcal{E}') \neq \varnothing \iff |\mathcal{L}_p(\mathcal{B}/\mathcal{E})| \geq n.$$

Since a language may contain a (countably) infinite number of elements, for reasoning about the size of the language, it is helpful to extended basic arithmetic operators to ω. Specifically, we extend addition to ω so that it is still commutative, and satisfies the equations

$$\omega + \omega = \omega, \quad \text{and} \quad n + \omega = \omega,$$

and we extend multiplication to ω so that it is still commutative, and satisfies the equations

$$\omega \times \omega = \omega, \qquad 0 \times \omega = 0, \quad \text{and} \quad n \times \omega = \omega \text{ if } n > 0.$$

In this chapter, we construct the directed graph $(D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$, called the *profile graph*, where

$$D_{\mathcal{A}} = \{\, d \in \Sigma \times \mathcal{P}(Q) \mid (\exists [t] \in \text{Can}_{\mathcal{R}_1/\mathcal{E}})\ \text{profile}_{\mathcal{A}/\mathcal{E}}([t]) = d \,\},$$

and $\trianglelefteq_{\mathcal{A}}$ contains an edge $d_1 \trianglelefteq_{\mathcal{A}} d_2$ iff there are $[t], [u] \in \text{Can}_{\mathcal{R}_1/\mathcal{E}}$ such that $\text{profile}_{\mathcal{A}/\mathcal{E}}([t]) = d_1$, $\text{profile}_{\mathcal{A}/\mathcal{E}}([u]) = d_2$, and $[t] \trianglelefteq_{\text{flat}} [u]$. The edge relation $\trianglelefteq_{\mathcal{A}}$ is used in counting the number of equivalence classes with a given profile. To given an example of the directed graph, in the automaton $\mathcal{B}'$ described above:

$$D_{\mathcal{B}'} = \{\, (a, \{\, p_1, p_2, p_3 \,\}), (b, \{\, p_1, p_2, p_3 \,\}), (c, \{\, p_1, p_2, p_3 \,\}), (\circ, \{\, p_2, p_3 \,\}), (\circ, \{\, p_3 \,\}) \,\},$$

and $\trianglelefteq_{\mathcal{B}'}$ contains the following edges:

$$\begin{aligned}
&(a, \{\, p_1, p_2, p_3 \,\}) \trianglelefteq_{\mathcal{B}'} (\circ, \{\, p_2, p_3 \,\}) \qquad (a, \{\, p_1, p_2, p_3 \,\}) \trianglelefteq_{\mathcal{B}'} (\circ, \{\, p_3 \,\}) \\
&(b, \{\, p_1, p_2, p_3 \,\}) \trianglelefteq_{\mathcal{B}'} (\circ, \{\, p_2, p_3 \,\}) \qquad (b, \{\, p_1, p_2, p_3 \,\}) \trianglelefteq_{\mathcal{B}'} (\circ, \{\, p_3 \,\}) \\
&(c, \{\, p_1, p_2, p_3 \,\}) \trianglelefteq_{\mathcal{B}'} (\circ, \{\, p_2, p_3 \,\}) \qquad (c, \{\, p_1, p_2, p_3 \,\}) \trianglelefteq_{\mathcal{B}'} (\circ, \{\, p_3 \,\})
\end{aligned}$$

Our approach is to incrementally construct $(D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$. We start with the empty graph $(D_0, \trianglelefteq_0) = (\varnothing, \varnothing)$ and apply inference rules to form increasing larger subgraphs $(D_1, \trianglelefteq_1) \subseteq (D_2, \trianglelefteq_2) \subseteq \cdots \subseteq (D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$ until saturation. This process terminates with a unique final graph as the size of $D_{\mathcal{A}}$ is at most $|\Sigma| \times 2^{|Q|}$, and the construction process is monotonic. Each profile graph $(D, \trianglelefteq) \subseteq$

$(D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$ can be viewed as representing the possibly infinite subset of $\mathrm{Can}_{\mathcal{R}_{\mathrm{I}}/\mathcal{E}}$ that is already explored:

Definition 3.4.4. *For each graph $(D, \trianglelefteq) \subseteq (D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$, let $\mathrm{Can}_{D,\trianglelefteq}$ denote the smallest set containing each $[t] \in \mathrm{Can}_{\mathcal{R}_{\mathrm{I}}/\mathcal{E}}$ if $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t]_{\mathcal{E}}) \in D$ and for all $[u] \in \mathrm{Can}_{\mathcal{R}_{\mathrm{I}}/\mathcal{E}}$,*

$$[u] \trianglelefteq_{\mathrm{flat}} [t] \implies [u] \in \mathrm{Can}_{D,\trianglelefteq} \wedge \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([u]) \trianglelefteq \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t]).$$

Furthermore, for each $d \in D$, we let $\mathrm{profile}^{-1}_{D,\trianglelefteq}(d)$ denote the elements in $\mathrm{Can}_{D,\trianglelefteq}$ with profile d, i.e., $\mathrm{profile}^{-1}_{D,\trianglelefteq}(d) = \{\, [t] \in \mathrm{Can}_{D,\trianglelefteq} \mid \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t]) = d \,\}$.

For the automaton $\mathcal{B}'$ described previously, if we let $(D, \trianglelefteq)$ denote the complete subgraph of $(D_{\mathcal{B}'}, \trianglelefteq_{\mathcal{B}'})$ containing the nodes

$$D = \big\{\, (a, \{\, p_1, p_2, p_3 \,\}), (b, \{\, p_1, p_2, p_3 \,\}), (\circ, \{\, p_2, p_3 \,\}), (\circ, \{\, p_3 \,\}) \,\big\},$$

then $\mathrm{Can}_{D,\trianglelefteq} = \{\, [a], [b], [a \circ b] \,\}$. The graph $(D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$ can be viewed as the graph where every $\mathcal{R}_{\mathrm{I}}/\mathcal{E}$-irreducible term has been explored.

Lemma 3.4.5. *For all $(D, \trianglelefteq) \subseteq (D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$,*

$$(D, \trianglelefteq) = (D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}}) \iff \mathrm{Can}_{D,\trianglelefteq} = \mathrm{Can}_{\mathcal{R}_{\mathrm{I}}/\mathcal{E}}.$$

Proof. Under the assumption $(D, \trianglelefteq) = (D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$, we first show that $\mathrm{Can}_{D,\trianglelefteq} = \mathrm{Can}_{\mathcal{R}_{\mathrm{I}}/\mathcal{E}}$. As $\hat{\mathcal{R}}_{\mathrm{I}}$ is confluent and terminating modulo $\mathcal{E}_{\mathrm{AC}}$, it is sufficient to show that for each $\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}$-irreducible term $t \in T_{\Sigma}$, $[t] \in \mathrm{Can}_{D_{\mathcal{A}},\trianglelefteq_{\mathcal{A}}}$. We prove this by structural induction on t. By definition, $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t]) \in D_{\mathcal{A}}$, and so we only need to prove that for all $\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}$-irreducible terms $u \in T_{\Sigma}$, if $[u] \trianglelefteq_{\mathrm{flat}} [t]$, then $[u] \in \mathrm{Can}_{D,\trianglelefteq}$ and $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([u]) \trianglelefteq_{\mathcal{A}} \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t])$. There are two cases to consider:

- If $t = f(t_1, \ldots, t_n)$ where f is a free symbol. In this case, if $u \in T_{\Sigma}$ is a $\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}$-irreducible term such that $[u] \trianglelefteq_{\mathrm{flat}} [t]$ then there is an $i \in [1, n]$ such that $u =_{\mathcal{E}_{\mathrm{AC}}} t_i$. Since t_i is a subterm of t, by induction we know that $[t_i] \in \mathrm{Can}_{D_{\mathcal{A}},\trianglelefteq_{\mathcal{A}}}$, and $[t_i] \trianglelefteq_{\mathrm{flat}} [t]$ by definition. Therefore, $[t] \in \mathrm{Can}_{D_{\mathcal{A}},\trianglelefteq_{\mathcal{A}}}$.
- Otherwise $t = t_1 + \cdots + t_n$ for some AC or ACI symbol $+$ where $n \geq 2$, $\mathrm{root}(t_i) \neq +$ for all $i \in [1, n]$. In this case, if $u \in T_{\Sigma}$ is a $\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}$-irreducible term such that $[u] \trianglelefteq_{\mathrm{flat}} [t]$, then there is an $i \in [1, n]$ such that $u =_{\mathcal{E}_{\mathrm{AC}}} t_i$. As t_i is a subterm of t, by induction we know that $[t_i] \in \mathrm{Can}_{D_{\mathcal{A}},\trianglelefteq_{\mathcal{A}}}$, and by definition $[t_i] \trianglelefteq_{\mathrm{flat}} [t]$. Therefore, $[t] \in \mathrm{Can}_{D_{\mathcal{A}},\trianglelefteq_{\mathcal{A}}}$.

On the other hand, if $\mathrm{Can}_{D,\trianglelefteq} = \mathrm{Can}_{\mathcal{R}_{\mathrm{I}}/\mathcal{E}}$, then for each $[t] \in \mathrm{Can}_{\mathcal{R}_{\mathrm{I}}/\mathcal{E}}$, we know that $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t]) \in D$ and thus $D = D_{\mathcal{A}}$. Additionally, for all

$[t], [u] \in \text{Can}_{\mathcal{R}_1/\mathcal{E}}$, if $[t] \trianglelefteq_{\text{flat}} [u]$, then we know $\text{profile}_{\mathcal{A}/\mathcal{E}}([t]) \trianglelefteq \text{profile}_{\mathcal{A}/\mathcal{E}}([u])$. Consequently, $\trianglelefteq = \trianglelefteq_{\mathcal{A}}$.

□

3.4.2 Free symbols

For each free symbol $f \in \Sigma$, we define a function states_f which computes the states of a term $f(t_1, \ldots, t_n)$ when the states for each term t_i are already known:

Definition 3.4.6. *Given a free symbol $f \in \Sigma$ with arity n, we define the function $\text{states}_f : \mathcal{P}(Q)^n \to \mathcal{P}(Q)$ such that for sets of states $P_1, \ldots, P_n \subseteq Q$, $\text{states}_f(P_1, \ldots, P_n) \subseteq Q$ is the smallest set containing a state $p \in Q$ if either:*

- *$\mathcal{A}$ contains the clause $p(f(x_1, \ldots, x_n)) \Leftarrow p_1(x_1), \ldots, p_n(x_n)$ with $p_i \in P_i$ for $i \in [1, n]$,*
- *or $\mathcal{A}$ contains $p(x_1 \circ x_2) \Leftarrow p_1(x_1), p_2(x_2)$ with $\circ$ an ACI-symbol in $\mathcal{E}$ and $p_1, p_2 \in \text{states}_f(P_1, \ldots, P_n)$.*

Let $f(t_1, \ldots, t_n)$ be a term whose root symbol f is free. In the definition above, the two cases mimic the two possible forms that a term $u =_{\mathcal{E}} f(t_1, \ldots, t_n)$ may have. As f is free, u must either have the form: (1) $u = f(u_1, \ldots, u_n)$ with $u_i =_{\mathcal{E}} t_i$ for all $i \in [1, n]$; or (2) $u = u_1 \circ u_2$ with $u_1 =_{\mathcal{E}} u_2$ and $\circ$ an ACI symbol.

Just as in Lemma 3.3.4, we can relate $\text{states}_{\mathcal{A}/\mathcal{E}}$ and states_f for theories with ACI symbols as follows:

Lemma 3.4.7. *For each term $t = f(t_1, \ldots, t_n) \in T_\Sigma$ with f free in $\mathcal{E}$,*

$$\text{states}_{\mathcal{A}/\mathcal{E}}([t]) = \text{states}_f(\text{states}_{\mathcal{A}/\mathcal{E}}([t_1]), \ldots, \text{states}_{\mathcal{A}/\mathcal{E}}([t_n])).$$

Proof. For all states $p \in Q$, we know by Theorem 3.1.1 that $p \in \text{states}_{\mathcal{A}/\mathcal{E}}([t])$ iff there is a term $u \in [t]$ such that $\mathcal{A} \vdash p(u)$. Since u is equivalent modulo $\mathcal{E}$ to a term whose root symbol is the free symbol f, u may only have two possible forms:

1. $u = f(u_1, \ldots, u_n)$ with $u_i =_{\mathcal{E}} t_i$ for $i \in [1, n]$. In this case, $\mathcal{A}$ must contain a clause $p(f(x_1, \ldots, x_n)) \Leftarrow p_1(x_1), \ldots, p_n(x_n)$ such that $\mathcal{A} \vdash p_i(u_i)$ for $i \in [1, n]$. It follows that $p_i \in \text{states}_{\mathcal{A}/\mathcal{E}}([t_i])$ for each $i \in [1, n]$.
2. $u = u_1 \circ u_2$ with $\circ$ is an ACI symbol $u_1 =_{\mathcal{E}} u_2$. In this case, as $\mathcal{A} \vdash p(u)$, $\mathcal{A}$ must contain a clause $p(x_1 \circ x_2) \Leftarrow p_1(x_1), p_2(x_2)$ such that $\mathcal{A} \vdash p_1(u_1)$ and $\mathcal{A} \vdash p_2(u_2)$. Furthermore, both u_1 and u_2 are smaller terms equivalent modulo $\mathcal{E}$ to u and t, so $p_1, p_2 \in \text{states}_{\mathcal{A}/\mathcal{E}}([t])$.

For $i \in [1, n]$, let $P_i = \text{states}_{\mathcal{A}/\mathcal{E}}([t_i])$. These two cases mirror the two rules used in the definition of states_f, and so it can be shown that for all $u =_{\mathcal{E}} t$, $\mathcal{A} \vdash p(u)$ implies $p \in \text{states}_f(P_1, \ldots, P_n)$ by structural induction on the proof $\mathcal{A} \vdash$

$p(u)$. It is also straightforward to show that $p \in \text{states}_f(P_1, \ldots, P_n) \implies \mathcal{A}/\mathcal{E} \vdash p(t)$ by induction on the inference steps used to construct $\text{states}_f(P_1, \ldots, P_n)$. □

Given a graph $(D, \trianglelefteq) \subseteq (D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$, the following lemma is useful for determining the number of distinct equivalence classes in $\text{Can}_{D,\trianglelefteq}$ with profile $(f, P) \in D_{\mathcal{A}}$ where f is a free symbol.

Lemma 3.4.8. *For each graph $(D, \trianglelefteq) \subseteq (D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$, and profile $(f, P) \in D$ where f is a free symbol,*

$$\left|\text{profile}^{-1}_{D,\trianglelefteq}(f, P)\right| = \sum_{\substack{(f_1,P_1),\ldots,(f_n,P_n) \in D \\ (\forall i \in [1,n])\ (f_i,P_i) \trianglelefteq (f,P) \\ \text{states}_f(P_1,\ldots,P_n) = P}} \prod_{i=1}^{n} \left|\text{profile}^{-1}_{D,\trianglelefteq}(f_i, P_i)\right|. \tag{3.2}$$

Proof. For each equivalence class $[t] \in \text{profile}^{-1}_{D,\trianglelefteq}(f, P)$, we may assume without loss of generality that t is $\hat{\mathcal{R}}_{\text{I}}/\mathcal{E}_{\text{AC}}$-irreducible. It follows that t has the form $f(t_1, \ldots, t_n)$ with $[t_i] \in \text{Can}_{D,\trianglelefteq}$ for $i \in [1, n]$. Let $P_i = \text{states}_{\mathcal{A}/\mathcal{E}}([t_i])$ for $i \in [1, n]$. As $[t]$ is in $\text{Can}_{D,\trianglelefteq}$, we know that $P_i \trianglelefteq (f, P)$. By Lemma 3.4.7, we know that $\text{states}_{\mathcal{A}/\mathcal{E}}([f(t_1, \ldots, t_n)]) = P$ if and only if $\text{states}_f(P_1, \ldots, P_n) = P$. For each tuple $(d_1, \ldots, d_n) \in D^n$, we define the set $\text{Can}_f(d_1, \ldots, d_n) \subseteq \text{Can}_{D,\trianglelefteq}$ as follows:

$$\text{Can}_f(d_1, \ldots, d_n) = \\ \{\, [f(t_1, \ldots, t_n)] \in \text{Can}_{D,\trianglelefteq} \mid \text{profile}_{\mathcal{A}/\mathcal{E}}([t_i]) = d_i \text{ for } i \in [1, n] \,\}.$$

For distinct tuples $\overline{d}_1, \overline{d}_2 \in D^n$, it is not difficult to observe that the sets $\text{Can}_f(\overline{d}_1)$ and $\text{Can}_f(\overline{d}_2)$ are disjoint. From these observations, we can conclude that:

$$\left|\text{Can}_{D,\trianglelefteq}(f, P)\right| = \sum_{\substack{(f_1,P_1),\ldots,(f_n,P_n) \in D \\ (\forall i \in [1,n])\ (f_i,P_i) \trianglelefteq (f,P) \\ \text{states}_f(P_1,\ldots,P_n) = P}} \left|\text{Can}_f((f_1, P_1), \ldots, (f_n, P_n))\right|. \tag{3.3}$$

Moreover, for each tuple $(d_1, \ldots, d_n) \in D^n$, it is not difficult to show that

$$\left|\text{Can}_f(d_1, \ldots, d_n)\right| = \left|\text{profile}^{-1}_{D,\trianglelefteq}(d_1)\right| \times \cdots \times \left|\text{profile}^{-1}_{D,\trianglelefteq}(d_n)\right|. \tag{3.4}$$

Equation (3.2) follows immediately from (3.3) and (3.4). □

3.4.3 AC and ACI symbols

Our results for AC and ACI symbols in this section are very similar to our results for A and AC symbols in Section 3.3. Many of the results involve the

grammar $G(+)$ given in Def. 3.3.5. We reprint this definition below in order to make this section more self contained.

Definition 3.4.9. *For an associative symbol $+ \in \Sigma$, $G(+)$ is the context free grammar with terminals $D(+) = (F - \{+\}) \times \mathcal{P}(Q)$, nonterminals Q, and production rules*

$$G(+) = \{\, p := p_1 p_2 \mid p(x_1 + x_2) \Leftarrow p_1(x_1), p_2(x_2) \in \mathcal{A} \,\}$$
$$\cup \{\, p := (f, P) \mid (f, P) \in D(+) \wedge p \in P \,\}.$$

For each nonterminal $p \in Q$, the Parikh image $\mathcal{S}_p(G(+))$ defined in Section 3.3.2 is definable by an existential Presburger formula $\psi_{G(+),p}(\overline{x})$ with free variables $\overline{x} = \{\, x_d \,\}_{d \in D(+)}$ whose models $M(\psi_{G(+),p}) = \mathcal{S}_p(G(+))$ [139]. We first show that parse trees in $G(+)$ corresponds to proofs in $\mathcal{A}/\mathcal{E}$:

Lemma 3.4.10. *For each term $t = t_1 + \cdots + t_n \in T_\Sigma$ where $+$ is an AC or ACI symbol, and $\mathrm{root}(t_i \downarrow_{\hat{\mathcal{R}}_1/\mathcal{E}_{\mathrm{AC}}}) \neq +$ for $i \in [1, n]$,*

$$\#(\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_1]), \ldots, \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_n])) \in M(\psi_{G(+),p}) \implies \mathcal{A}/\mathcal{E} \vdash p(t).$$

Proof. We show this statement by induction on $n \geq 1$. There are two cases to consider:

- If $n = 1$, $G(+)$ must contain the rule $p := \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_1])$. This implies $p \in \mathrm{states}_{\mathcal{A}/\mathcal{E}}([t_1])$, and thus $\mathcal{A}/\mathcal{E} \vdash p(t_1)$.
- Otherwise $n \geq 2$, and $G(+)$ contains a rule $p := p_1 p_2$ which can be viewed as partitioning $t_1, \ldots, t_n$ into two sequences:

  1. a sequence $u_1, \ldots, u_m$ with $1 < m < n$ and

     $$\#(\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([u_1]), \ldots, \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([u_m])) \in M(\psi_{G(+),p_1}), \text{ and}$$

  2. a sequence $v_1, \ldots, v_{n-m}$ with

     $$\#(\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([v_1]), \ldots, \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([v_{n-m}])) \in M(\psi_{G(+),p_2}).$$

  Let $u = u_1 + \cdots + u_m$, and let $v = v_1 + \cdots + v_{n_m}$. As $+$ is associative in $\mathcal{E}$, we know that $t =_{\mathcal{E}} u + v$. As both m and $n - m$ are less than n, by induction we know that $\mathcal{A}/\mathcal{E} \vdash p_1(u)$ and $\mathcal{A}/\mathcal{E} \vdash p_2(v)$. By the definition of $G(+)$, we know that $\mathcal{A}$ contains the clause $p(x_1 + x_2) \Leftarrow p_1(x_1), p_2(x_2)$, and consequently $\mathcal{A}/\mathcal{E} \vdash p(t)$.

□

Due to the possibility of idempotence equations in $\mathcal{E}$, it is more complex to show how a proof that $\mathcal{A}/\mathcal{E} \vdash p(t)$ corresponds to a parse tree using the production rules in $G(+)$. We first show the following lemma, which relies on our assumption that $\mathcal{A}$ is AC intersection free.

Lemma 3.4.11. *For each AC or ACI symbol $+ \in \Sigma$, if $\mathcal{A}/\mathcal{E} \vdash p(t)$, then there is a term $t_1 + \cdots + t_n =_{\mathcal{E}} t$ such that $\#(\text{profile}_{\mathcal{A}/\mathcal{E}}([t_1]), \ldots, \text{profile}_{\mathcal{A}/\mathcal{E}}([t_n])) \in M(\psi_{G(+),p})$ and $\text{root}(t_i) \neq +$ for $i \in [1, n]$.*

Proof. The term $t_1 + \cdots + t_n =_{\mathcal{E}} t$ can be inductively constructed from the proof used to show $\mathcal{A}/\mathcal{E} \vdash p(t)$. Equivalence steps in that proof are trivial as the inductive hypothesis immediately implies a suitable term can be constructed. For membership steps, there are three cases to consider:

- We first consider the case where $\text{root}(t \downarrow_{\hat{\mathcal{R}}_{\text{I}}/\mathcal{E}_{\text{AC}}}) \neq +$. By Theorem 3.1.1, there is a term $u =_{\mathcal{E}} t$ such that $\mathcal{A} \vdash p(u)$. We know that $u \downarrow_{\hat{\mathcal{R}}_{\text{I}}/\mathcal{E}_{\text{AC}}} =_{\mathcal{E}_{\text{AC}}} t \downarrow_{\hat{\mathcal{R}}_{\text{I}}/\mathcal{E}_{\text{AC}}}$, and thus $\text{root}(u \downarrow_{\hat{\mathcal{R}}_{\text{I}}/\mathcal{E}_{\text{AC}}}) \neq +$. Consequently, $G(+)$ contains the rule $p := \text{profile}_{\mathcal{A}/\mathcal{E}}([u])$, and so $\#(\text{profile}_{\mathcal{A}/\mathcal{E}}([u])) \in M(\psi_{G(+),p})$. The term $u \downarrow_{\hat{\mathcal{R}}_{\text{I}}/\mathcal{E}_{\text{AC}}} =_{\mathcal{E}} t$ is exactly the term we are looking for.

- We next consider the case where $\text{root}(t) = +$, and we let $t = u + v$. The membership must have the form:

$$\frac{\mathcal{A}/\mathcal{E} \vdash p_1(u) \qquad \mathcal{A}/\mathcal{E} \vdash p_2(v)}{\mathcal{A}/\mathcal{E} \vdash p(u + v)}.$$

  By our induction hypothesis, there are terms $u_1 + \cdots + u_m =_{\mathcal{E}} u$ and $v_1 + \cdots + v_n =_{\mathcal{E}} v$ satisfying the conditions in the lemma. Let $\overline{u}, \overline{v} \in \mathbb{N}^{F \times \mathcal{P}(Q)}$ be the vectors:

$$\overline{u} = \#(\text{profile}_{\mathcal{A}/\mathcal{E}}([u_1]), \ldots, \text{profile}_{\mathcal{A}/\mathcal{E}}([u_m]))$$
$$\overline{v} = \#(\text{profile}_{\mathcal{A}/\mathcal{E}}([v_1]), \ldots, \text{profile}_{\mathcal{A}/\mathcal{E}}([v_n])).$$

  We know $\mathcal{A}$ contains the clause $p(x_1 + x_2) \Leftarrow p_1(x_1), p_2(x_2)$, and so $G(+)$ contains the rule $p := p_1 p_2$. By induction, we know that $\overline{u} \in M(\psi_{G(+),p_1})$ and $\overline{v} \in M(\psi_{G(+),p_2})$, and so $\overline{u} + \overline{v} \in M(\psi_{G(+),p})$. It follows that the term $(u_1 + \cdots + u_m) + (v_1 + \cdots + v_n) \in [1]_{\mathcal{E}}$ satisfies the required conditions.

- Otherwise, we know that $\text{root}(t \downarrow_{\hat{\mathcal{R}}_{\text{I}}/\mathcal{E}_{\text{AC}}}) = +$ while $\text{root}(t) \neq +$. It follows that t must have the form $t = u \circ v$ for some ACI symbol $\circ$, and the membership must have the form:

$$\frac{\mathcal{A}/\mathcal{E} \vdash p_1(u) \qquad \mathcal{A}/\mathcal{E} \vdash p_2(v)}{\mathcal{A}/\mathcal{E} \vdash p(u \circ v)}$$

  where $u =_{\mathcal{E}} v =_{\mathcal{E}} t$. We construct $u_1 + \cdots + u_n \in [u]_{\mathcal{E}}$ by induction. We know that $n \geq 2$, as $u_1 + \cdots + u_n =_{\mathcal{E}} t$ implies $\text{root}(u_1 + \cdots + u_n \downarrow_{\hat{\mathcal{R}}_{\text{I}}/\mathcal{E}_{\text{AC}}}) = +$. Let $\overline{u} = \#(\text{profile}_{\mathcal{A}/\mathcal{E}}([u_1]), \ldots, \text{profile}_{\mathcal{A}/\mathcal{E}}([u_n]))$. By induction we

know that $\overline{u} \in M(\psi_{G(+),p_1})$, and thus $G(+)$ contains a rule of the form $p_1 := q_1 q_2$ as $n \geq 2$. It follows that $\mathcal{A}$ contains the rule $p_1(x_1 + x_2) \Leftarrow q_1(x_1), q_2(x_2)$. As $\mathcal{A}$ is AC intersection free, it must also contain must contain the rule $p(x_1 + x_2) \Leftarrow q_1(x_1), q_2(x_2)$. Thus $G(+)$ contains the $p := q_1 q_2$, and if we swap this rule in for the rule $p_1 := q_1 q_2$ used to show $\overline{u} \in M(\psi_{G(+),p_1})$, it follows that $\overline{u} \in M(\psi_{G(+),p})$. It follows that the term $u_1 + \cdots + u_1 \in [t]_{\mathcal{E}}$ satisfies the required conditions.

□

For each AC symbol $+$, we can show:

Lemma 3.4.12. *For each $\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}$-irreducible term $t = t_1 + \cdots + t_n \in T_\Sigma$ where $+$ is an AC symbol and $\mathrm{root}(t_i) \neq +$ for $i \in [1, n]$,*

$$\mathcal{A}/\mathcal{E} \vdash p(t) \iff \#(\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_1]), \ldots, \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_n])) \in M(\psi_{G(+),p}).$$

Proof. Lemma 3.4.10 shows that

$$\#(\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_1]), \ldots, \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_n])) \in M(\psi_{G(+),p}) \implies \mathcal{A}/\mathcal{E} \vdash p(t).$$

On the other hand, if $\mathcal{A}/\mathcal{E} \vdash p(t)$ then by Lemma 3.4.11, there is a term $u_1 + \cdots + u_m =_{\mathcal{E}} t$ such that $\#(\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([u_1]), \ldots, \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([u_m])) \in M(\psi_{G(+),p})$ and $\mathrm{root}(u_i) \neq +$ for $i \in [1, m]$. As $t_1 + \cdots + t_n =_{\mathcal{E}} u_1 + \cdots + u_m$, and $+$ only appears in associativity and commutativity equations, it follows that $m = n$ and

$$\#(\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([u_1]), \ldots, \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([u_m])) = \#(\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_1]), \ldots, \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_n])).$$

Consequently,

$$\#(\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_1]), \ldots, \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_n])) \in M(\psi_{G(+),p}).$$

□

For each ACI symbol $\circ$, we can show:

Lemma 3.4.13. *For each $\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}$-irreducible term $t = t_1 \circ \cdots \circ t_n \in T_\Sigma$ where $\circ$ is an ACI symbol and $\mathrm{root}(t_i) \neq \circ$ for $i \in [1, n]$,*

$$\mathcal{A}/\mathcal{E} \vdash p(t) \iff \#(\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_1]), \ldots, \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_n])) \in M((\exists \overline{y})\, \overline{x} \sqsubseteq \overline{y} \wedge \psi_{G(\circ),p}(\overline{y}))$$

where $\overline{x} \sqsubseteq \overline{y}$ is the formula $\bigwedge_{d \in D(\circ)} x_d \leq y_d \wedge ((y_d > 0) \Rightarrow (x_d > 0))$.

Proof. We let $\overline{x} = \#(\text{profile}_{\mathcal{A}/\mathcal{E}}([t_1]), \ldots, \text{profile}_{\mathcal{A}/\mathcal{E}}([t_n]))$ We first show that if there is a $\overline{y} \in \mathbb{N}^{F \times \mathcal{P}(Q)}$ such that $\overline{x} \sqsubseteq \overline{y}$ and $\psi_{G(\circ),p}(\overline{y})$ holds, then $\mathcal{A}/\mathcal{E} \vdash p(t)$. Since $\overline{x} \sqsubseteq \overline{y}$, we know that for each $d \in D(\circ)$, if $y_d > x_d$, then $x_d > 0$. Consequently, there is a term $t_d \in T_\Sigma$ such that $t_d = t_i$ for some $i \in [1, n]$. We let $u \in T_\Sigma$, denote the term $u = t \circ u_1 \circ \cdots \circ u_m$ where for each $d \in D$ with $y_d > x_d$, there are exactly $y_d - x_d$ distinct indices $d(1), \ldots, d(y_d - x_d) \in [1, m]$ such that $u_{d(i)} = t_d$. It is not difficult to show that $u =_{\mathcal{E}} t$ as $\circ$ is ACI, and moreover $\overline{x} + \#(\text{profile}_{\mathcal{A}/\mathcal{E}}(u_1), \ldots, \text{profile}_{\mathcal{A}/\mathcal{E}}(u_m)) = \overline{y}$. It follows by Lemma 3.4.10 that $\mathcal{A}/\mathcal{E} \vdash p(u)$, and thus $\mathcal{A}/\mathcal{E} \vdash p(t)$.

We now show that if $\mathcal{A}/\mathcal{E} \vdash p(t)$, then there is a vector $\overline{y} \in \mathbb{N}^{D(\circ)}$ such that $\#(\text{profile}_{\mathcal{A}/\mathcal{E}}([t_1]), \ldots, \text{profile}_{\mathcal{A}/\mathcal{E}}([t_n])) \sqsubseteq \overline{y}$ and $\overline{y} \in M(\psi_{G(\circ),p})$. In this case, by Lemma 3.4.11, there is a term $u_1 \circ \cdots \circ u_m =_{\mathcal{E}} t$ such that

$$\#(\text{profile}_{\mathcal{A}/\mathcal{E}}([u_1]), \ldots, \text{profile}_{\mathcal{A}/\mathcal{E}}([u_m])) \in M(\psi_{G(\circ),p}).$$

and $\text{root}(u_i \downarrow_{\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}}) \neq \circ$ for $i \in [1, m]$. As t is $\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}$-irreducible, there must be a surjective function $h : [1, m] \to [1, n]$ such that $u_i =_{\mathcal{E}} t_{h(i)}$ for $i \in [1, m]$. Let $\overline{y} = \#(\text{profile}_{\mathcal{A}/\mathcal{E}}([u_1]), \ldots, \text{profile}_{\mathcal{A}/\mathcal{E}}([u_m]))$. For each $d \in D$, the existence of h implies $x_d \leq y_d$ and the surjectivity of h implies that $y_d > 0 \implies x_d > 0$. □

We use $\psi_{G(+),p}$ to define the formula $\psi_{G(+),P}$ which identifies terms whose profile is $(+, P)$. For each AC symbol $+ \in \Sigma$ and each symbol $\circ \in \Sigma$ idempotent in $\mathcal{E}$ or $\mathcal{R}$,

$$\psi_{+,P}(\overline{x}) = \bigwedge_{p \in P} \psi_{G(+),p}(\overline{x}) \wedge \bigwedge_{p \in Q \setminus P} \neg\psi_{G(+),p}(\overline{x}) \wedge \sum_{x_d \in \overline{x}} x_d \geq 2$$
$$\psi_{\circ,P}(\overline{x}) = \bigwedge_{p \in P} (\exists \overline{y})\, \overline{x} \sqsubseteq \overline{y} \wedge \psi_{G(\circ),p}(\overline{y}) \wedge \bigwedge_{p \in Q \setminus P} \neg(\exists \overline{y})\, \overline{x} \sqsubseteq \overline{y} \wedge \psi_{G(\circ),p}(\overline{y}) \wedge \sum_{x_d \in \overline{x}} x_d \geq 2.$$

The following lemma describes precisely how the models in $M(\psi_{+,P})$ correspond to $\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}$-irreducible terms with a particular profile.

Lemma 3.4.14. *For each $\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}$-irreducible term $t = t_1 + \cdots + t_n \in T_\Sigma$ where $+ \in \Sigma$ is an AC or ACI symbol and $\text{root}(t_i) \neq +$ for $i \in [1, n]$,*

$$\text{profile}_{\mathcal{A}/\mathcal{E}}([t]) = (+, P) \iff \#(\text{profile}_{\mathcal{A}/\mathcal{E}}([t_1]), \ldots, \text{profile}_{\mathcal{A}/\mathcal{E}}([t_n])) \in M(\psi_{+,P}).$$

Proof. Since t is $\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}$ irreducible, we know that $\text{profile}_{\mathcal{A}/\mathcal{E}}([t]) = (+, P)$ iff $n \geq 2$ and $\mathcal{A}/\mathcal{E} \vdash p(t) \iff p \in P$ for all states $p \in Q$. Let $\overline{x} = \#(\text{profile}_{\mathcal{A}/\mathcal{E}}(t_1), \ldots, \text{profile}_{\mathcal{A}/\mathcal{E}}(t_n))$.

- If $+$ is an AC symbol, then by Lemma 3.4.12, $\mathcal{A}/\mathcal{E} \vdash p(t) \iff \overline{x} \in$

$M(\psi_{G(+),p})$. It follows that $p \in P \iff \in M(\psi_{G(+),p})$, and consequently

$$\text{profile}_{\mathcal{A}/\mathcal{E}}([t]) = (+, P) \iff \overline{x} \in M(\psi_{+,P}).$$

- Otherwise $+$ is an ACI symbol, and by Lemma 3.4.13, $\mathcal{A}/\mathcal{E} \vdash p(t)$ iff there is a $\overline{y} \in \mathbb{N}^{D(+)}$ such that $\overline{x} \sqsubseteq \overline{y}$ and $\overline{y} \in M(\psi_{G(+),p}(\overline{y}))$. It follows that $p \in P \iff \overline{x} \in M((\exists \overline{y})\, \overline{x} \sqsubseteq \overline{y} \wedge \psi_{G(+),p}(\overline{y}))$, and consequently

$$\text{profile}_{\mathcal{A}/\mathcal{E}}([t]) = (+, P) \iff \overline{x} \in M(\psi_{+,P}).$$

□

We now turn our attention to the problem of counting the number of distinct elements in $\text{Can}_{D,\trianglelefteq}$ with profile $(+, P) \in D_{\mathcal{A}}$ where $+$ is an AC or ACI symbol. For doing this, the classical *choose function* $\text{C} : (\mathbb{N} \cup \{\omega\}) \times \mathbb{N} \to \mathbb{N} \cup \{\omega\}$ which has been partially extended to ω becomes quite useful.

$$\text{C}(n, k) = n!/(k!(n-k)!), \quad \text{C}(\omega, 0) = 1, \quad \text{and} \quad \text{C}(\omega, k) = \omega \text{ if } k > 0.$$

For a symbol $\circ \in \Sigma$ that is ACI in $\mathcal{E}$ or AC in $\mathcal{E}$ and $\mathcal{R}_{\text{I}}$ contains the rule $x \circ x \to x$ appears in $\mathcal{R}$, each equivalence class $[t_1 \circ \cdots \circ t_n] \in \text{Can}_{\mathcal{R}_{\text{I}}/\mathcal{E}}$ can be viewed as a set $\{[t_1], \ldots, [t_n]\} \subseteq \text{Can}_{\mathcal{R}_{\text{I}}/\mathcal{E}}$. For these symbols, the following classical result about C becomes quite useful:

Proposition 3.4.15. *Given a finite or countably infinite set A and natural number $k \leq |A|$, the total number of distinct subsets of A with size k equals* $\text{C}(|A|, k)$.

Proof. If $k = 0$, then there the empty set is the only set with size 0, and clearly $\text{C}(|A|, 0) = 0$. When A is a countably infinite set and $k > 0$, then there are an infinite number of subsets with size k. If A is finite with size $n > 0$, then there are two cases to consider. If $k = n$, then there is only one subset of A, namely A itself. This leaves us with the final case $0 < k < |A|$. In this case, for each element $a_0 \in A$, by induction on n we know that there are $\text{C}(n-1, k)$ subsets of A with size k not containing a_0, and $\text{C}(n-1, k-1)$ subsets of A with size k containing a_0. It is straightforward to show that

$$\text{C}(n, k) = \text{C}(n-1, k) + \text{C}(n-1, k-1).$$

□

For an AC symbols $+ \in \Sigma$ where $\mathcal{R}$ does not contain an idempotence rule, each equivalence class $[t_1 + \cdots + t_n] \in \text{Can}_{\mathcal{R}_{\text{I}}/\mathcal{E}}$ can be viewed as a multiset $\{\!\{[t_1], \ldots, [t_n]\}\!\} \in \mathbb{N}^{\text{Can}_{\mathcal{R}_{\text{I}}/\mathcal{E}}}$. For these symbols, the following classical result about C becomes quite useful:

Proposition 3.4.16. *Given a non-empty finite or countably infinite set A and natural number $k \in \mathbb{N}$, the total number of distinct multisets of A is given by the formula* $\mathrm{C}(|A| + k - 1, k)$.

Proof. If $k = 0$, then the empty multiset is the only set with size 0, and clearly $\mathrm{C}(|A| - 1, 0) = 0$. When A is a countably infinite set and $k > 0$, then there are an infinite number of distinct multisets with size k. Finally if $k > 0$ and A is finite with size $n > 0$, then for each element $a_0 \in A$, by induction there are $\mathrm{C}(n + k - 2, k - 1)$ multisets with size k containing at least one a_0 and $\mathrm{C}(n + k - 2, k)$ multisets with size k not containing a_0. It is straightforward to show that

$$\mathrm{C}(n + k - 1, k) = \mathrm{C}(n + k - 2, k - 1) + \mathrm{C}(n + k - 2, k).$$

□

For a symbol $\circ$ is idempotent in $\mathcal{E}$ or $\mathcal{R}$, we need the following result about the size of $\mathrm{profile}^{-1}_{D,\trianglelefteq}(\circ, P)$:

Lemma 3.4.17. *For each profile graph $(D, \trianglelefteq) \subseteq (D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$, and profile $(\circ, P) \in D$ where $\circ$ is a symbol that is idempotent in $\mathcal{E}$ or $\mathcal{R}_{\mathrm{I}}$,*

$$\left|\mathrm{profile}^{-1}_{D,\trianglelefteq}(\circ, P)\right| = \sum_{\overline{u} \in M(\psi_{\circ,P,\mathrm{Can}_{D,\trianglelefteq}})} \prod_{d \trianglelefteq (\circ,P)} \mathrm{C}(\left|\mathrm{profile}^{-1}_{D,\trianglelefteq}(d)\right|, u_d) \tag{3.5}$$

where $\psi_{\circ,P,\mathrm{Can}_{D,\trianglelefteq}}(\overline{x}) = \psi_{\circ,P}(\overline{x}) \wedge \bigwedge_{d \trianglelefteq (\circ,P)} x_d \leq \left|\mathrm{profile}^{-1}_{D,\trianglelefteq}(d)\right| \wedge \bigwedge_{d \not\trianglelefteq (\circ,P)} x_d = 0.$

Proof. For each $\overline{v} \in \mathbb{N}^{D(\circ)}$, let $\mathrm{profile}^{-1}_{D,\trianglelefteq,\circ}(\overline{v}) \subseteq \mathrm{Can}^{*}_{D,\trianglelefteq}$ denote the set

$$\begin{aligned}\mathrm{profile}^{-1}_{D,\trianglelefteq,\circ}(\overline{v}) = \{\, \{[t_1], \ldots, [t_n]\} \subseteq \mathrm{Can}_{D,\trianglelefteq} \mid (\forall i, j \in [1,n])\, i \neq j \implies t_i \neq_{\mathcal{E}} t_j \\ \wedge\ \#(\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_1]), \ldots, \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_n])) = \overline{v}\,\}.\end{aligned}$$

For each $\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}$-irreducible term $t \in T_\Sigma$ such that $[t]_{\mathcal{E}} \in \mathrm{profile}^{-1}_{D,\trianglelefteq}(\circ, P)$, we know that t must have the form $t = t_1 \circ \cdots \circ t_n$ where $n \geq 2$. Moreover, we can assume that each term t_i is distinct with $\mathrm{root}(t_i) \neq \circ$ and $[t_i]_{\mathcal{E}} \in \mathrm{Can}_{D,\trianglelefteq}$ for $i \in [1, n]$. Let $\overline{x} = \#(\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_1]), \ldots, \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_n]))$. By Lemma 3.4.14, we know that $\overline{t} \in M(\psi_{\circ,P})$. As $[t_i] \in \mathrm{Can}_{D,\trianglelefteq}$ for $i \in [1, n]$, we know that $x_d \leq \left|\mathrm{profile}^{-1}_{D,\trianglelefteq}(d)\right|$ for $d \in D$. It follows that $\overline{x} \in M(\psi_{\circ,P})$. For distinct vectors $\overline{u}, \overline{v} \in \mathbb{N}^{D(\circ)}$, we know that $\mathrm{profile}^{-1}_{D,\trianglelefteq,\circ}(\overline{u})$ and $\mathrm{profile}^{-1}_{D,\trianglelefteq,\circ}(\overline{v})$ are disjoint sets, and consequently,

$$\left|\mathrm{profile}^{-1}_{D,\trianglelefteq}(\circ, P)\right| = \sum_{\overline{u} \in M(\psi_{\circ,P,\mathrm{Can}_{D,\trianglelefteq}})} \left|\mathrm{profile}^{-1}_{D,\trianglelefteq,\circ}(\overline{u})\right|. \tag{3.6}$$

Moreover, if we partition equivalence classes in each set in $\mathrm{profile}^{-1}_{D,\trianglelefteq,\circ}(\overline{u})$ by

their profile, it can be observed that:

$$\left|\text{profile}^{-1}_{D,\trianglelefteq,\circ}(\overline{u})\right| = \prod_{d \trianglelefteq (\circ,P)} \left|\{\, P \subseteq \text{profile}^{-1}_{D,\trianglelefteq}(d) \mid |P| = u_d \,\}\right|. \tag{3.7}$$

Finally, by Prop. 3.4.15, it follows that for each $k \leq \left|\text{profile}^{-1}_{D,\trianglelefteq}(d)\right|$,

$$\left|\{\, P \subseteq \text{profile}^{-1}_{D,\trianglelefteq}(d) \mid |P| = k \,\}\right| = \text{C}(\left|\text{profile}^{-1}_{D,\trianglelefteq}(d)\right|, k). \tag{3.8}$$

Equation (3.5) follows immediately from (3.6), (3.7), and (3.8). □

For an AC symbol $+$ is not idempotent in $\mathcal{R}$, we need the following result about the size of $\text{profile}^{-1}_{D,\trianglelefteq}(+,P)$:

Lemma 3.4.18. *For each graph $(D,\trianglelefteq) \subseteq (D_{\mathcal{A}},\trianglelefteq_{\mathcal{A}})$, and profile $(+,P) \in D$ where $+$ is an AC symbol in $\mathcal{E}$ that is not idempotent in $\mathcal{R}_{\text{I}}$,*

$$\left|\text{profile}^{-1}_{D,\trianglelefteq}(+,P)\right| = \sum_{\overline{u} \in M(\psi_{+,P,\text{Can}_{D,\trianglelefteq}})} \prod_{\substack{d \trianglelefteq (+,P) \\ \left|\text{profile}^{-1}_{D,\trianglelefteq}(d)\right| > 0}} \text{C}(\left|\text{profile}^{-1}_{D,\trianglelefteq}(d)\right| + u_d - 1, u_d) \tag{3.9}$$

where $\psi_{+,P,\text{Can}_{D,\trianglelefteq}}(\overline{x}) = \psi_{+,P}(\overline{x}) \wedge \bigwedge_{\substack{d \trianglelefteq (+,P) \\ \left|\text{profile}^{-1}_{D,\trianglelefteq}(d)\right| = 0}} x_d = 0 \bigwedge_{d \not\trianglelefteq (+,P)} x_d = 0.$

Proof. For each $\overline{v} \in \mathbb{N}^{D(+)}$, let $\text{profile}^{-1}_{D,\trianglelefteq,+}(\overline{v}) \subseteq \text{Can}^{*}_{D,\trianglelefteq}$ denote the set

$$\{\, \{\!\{[t_1],\ldots,[t_n]\}\!\} \in \mathbb{N}^{\text{Can}_{D,\trianglelefteq}} \mid \#(\text{profile}_{\mathcal{A}/\mathcal{E}}([t_1]),\ldots,\text{profile}_{\mathcal{A}/\mathcal{E}}([t_n])) = \overline{v} \,\}.$$

For each $[t]_{\mathcal{E}} \in \text{profile}^{-1}_{D,\trianglelefteq}(+,P)$, we can assume that t is $\hat{\mathcal{R}}_{\text{I}}/\mathcal{E}_{\text{AC}}$-irreducible and $t = t_1 + \cdots + t_n$ where $n \geq 2$, $\text{root}(t_i) \neq +$, and $[t_i]_{\mathcal{E}} \in \text{Can}_{D,\trianglelefteq}$ for $i \in [1,n]$. Let $\bar{t} = \#(\text{profile}_{\mathcal{A}/\mathcal{E}}([t_1]),\ldots,\text{profile}_{\mathcal{A}/\mathcal{E}}([t_n]))$. By Lemma 3.4.14, we know that $\bar{t} \in M(\psi_{+,P})$. By the definition of $\text{Can}_{D,\trianglelefteq}$ we know that for $i \in [1,n]$, $\text{profile}_{\mathcal{A}/\mathcal{E}}([t_i]) \trianglelefteq (+,P)$. For $i \in [1,n]$, if t_i has a profile d, then we know that $t_i \in \text{profile}^{-1}_{D,\trianglelefteq,+}(d)$ and consequently $\left|\text{profile}^{-1}_{D,\trianglelefteq,+}(d)\right| > 0$. It follows that $\bar{t} \in M(\psi_{+,P})$. For distinct vectors $\overline{u},\overline{v} \in \mathbb{N}^{D(+)}$, we know that $\text{profile}^{-1}_{D,\trianglelefteq}(\overline{u})$ and $\text{profile}^{-1}_{D,\trianglelefteq}(\overline{v})$ are disjoint sets. By putting the last two observations together, we can conclude that:

$$\left|\text{profile}^{-1}_{D,\trianglelefteq}(+,P)\right| = \sum_{\overline{u} \in M(\psi_{+,P,\text{Can}_{D,\trianglelefteq}})} \left|\text{profile}^{-1}_{D,\trianglelefteq}(\overline{u})\right|. \tag{3.10}$$

Moreover, if we partition the elements of each multiset in $\text{profile}^{-1}_{D,\trianglelefteq}(\overline{u})$ by their profile d, it is not difficult to show that

$$\left|\text{profile}^{-1}_{D,\trianglelefteq}(\overline{u})\right| = \prod_{d \trianglelefteq (+,P)} \left|\{\, \overline{x} \in \mathbb{N}^{\text{profile}^{-1}_{D,\trianglelefteq}(d)} \mid |\overline{x}| = u_d \,\}\right|. \tag{3.11}$$

Finally, by Prop. 3.4.16, it follows that for each $k \in N$ and $d \in D$ where

$\text{profile}^{-1}_{D,\trianglelefteq}(d)$ is non-empty, [

$$\left|\{\,\overline{x} \in \mathbb{N}^{\text{profile}^{-1}_{D,\trianglelefteq}(d)} \mid |\overline{x}| = k\,\}\right| = \mathrm{C}(\left|\text{profile}^{-1}_{D,\trianglelefteq}(d)\right| + k - 1, k). \tag{3.12}$$

Equation (3.9) follows immediately from (3.10), (3.11), and (3.12). □

3.4.4 Computing the size of a language

We next introduce a function $\text{cnt}_{D,\trianglelefteq} : D \to \mathbb{N} \cup \{\,\omega\,\}$ which for each graph $(D, \trianglelefteq) \subseteq (D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$ and profile $d \in D$, returns an estimate of the number of elements in $\text{Can}_{\mathcal{R}_{\mathrm{I}}/\mathcal{E}}$ with the profile d. We show below that for each $d \in D$,

$$\left|\text{profile}^{-1}_{D,\trianglelefteq}(d)\right| \leq \text{cnt}_{D,\trianglelefteq}(d) \leq \left|\text{profile}^{-1}_{D_{\mathcal{A}},\trianglelefteq_{\mathcal{A}}}(d)\right|. \tag{3.13}$$

For correctness purposes, any value in the range is sufficient. The proof that our procedure always shows *emptiness* only requires that $\text{cnt}_{D,\trianglelefteq}(d)$ is at most the total number of elements in $\text{Can}_{D_{\mathcal{A}},\trianglelefteq_{\mathcal{A}}}$ with a profile d, while the proof that our procedure always shows *non-emptiness* only requires that $\text{cnt}_{D,\trianglelefteq}(d)$ is at least the number of explored elements in $\text{Can}_{D,\trianglelefteq}$ with a profile d.

Before showing (3.13), we first must define $\text{cnt}_{D,\trianglelefteq}$. In the definition, we use the *choose function* $\mathrm{C} : (\mathbb{N} \cup \{\,\omega\,\}) \times \mathbb{N} \to \mathbb{N} \cup \{\,\omega\,\}$ which is partially extended to ω, i.e,

$$\mathrm{C}(n,k) = n!/k!(n-k)!, \qquad \mathrm{C}(\omega, 0) = 1, \qquad \text{and} \qquad \mathrm{C}(\omega, k) = \omega \text{ if } k > 0.$$

Definition 3.4.19. *For each $(D, \trianglelefteq) \subseteq (D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$, let $\text{cnt}_{D,\trianglelefteq} : D \to \mathbb{N} \cup \{\,\omega\,\}$ be the function such that $\text{cnt}_{D,\trianglelefteq}(d) = \omega$ if $d \trianglelefteq^{+} d$ and otherwise*

- *For each constant $c \in \Sigma$, $\text{cnt}_{D,\trianglelefteq}(c, P) = 1$.*
- *For each free symbol $f \in \Sigma$ with arity $n > 0$,*

$$\text{cnt}_{D,\trianglelefteq}(f, P) = \sum_{\substack{(f_1,P_1),\ldots,(f_n,P_n) \in D \\ (\forall i \in [1,n])\ (f_i,P_i) \trianglelefteq (f,P) \\ \text{states}_f(P_1,\ldots,P_n) = P}} \prod_{i=1}^{n} \text{cnt}_{D,\trianglelefteq}(f_i, P_i).$$

- *For an AC symbol $\circ \in \Sigma$ that is idempotent in $\mathcal{E}$ or $\mathcal{R}$, $\text{cnt}_{D,\trianglelefteq}(\circ, P) = \omega$ if $|M(\psi_{\circ,P,\mathrm{I}})| = \omega$, and otherwise,*

$$\text{cnt}_{D,\trianglelefteq}(\circ, P) = \sum_{\overline{u} \in M(\psi_{\circ,P,\mathrm{I}})} \prod_{d \trianglelefteq (\circ,P)} \mathrm{C}(\text{cnt}_{D,\trianglelefteq}(d), u_d)$$

where $\psi_{\circ,P,\mathrm{I}}(\overline{x}) = \psi_{\circ,P}(\overline{x}) \wedge \bigwedge_{d \trianglelefteq (\circ,P)} x_d \leq \text{cnt}_{D,\trianglelefteq}(d) \wedge \bigwedge_{d \not\trianglelefteq (\circ,P)} x_d = 0$.

- *For an AC symbol $+ \in \Sigma$ that is not idempotent in $\mathcal{E}$ or $\mathcal{R}$, we let*

$\mathrm{cnt}_{D,\trianglelefteq}(+,P) = \omega$ *if* $|M(\psi_{+,P,\mathrm{AC}})| = \omega$*, and otherwise,*

$$\mathrm{cnt}_{D,\trianglelefteq}(+,P) = \sum_{\overline{u} \in M(\psi_{+,P,\mathrm{AC}})} \prod_{\substack{d \trianglelefteq (+,P) \\ \mathrm{cnt}_{D,\trianglelefteq}(d) > 0}} \mathrm{C}(\mathrm{cnt}_{D,\trianglelefteq}(d) + u_d - 1, u_d)$$

where $\psi_{+,P,\mathrm{AC}}(\overline{x}) = \psi_{+,P}(\overline{x}) \wedge \bigwedge_{\substack{d \trianglelefteq (+,P) \\ \mathrm{cnt}_{D,\trianglelefteq}(d) = 0}} x_d = 0 \wedge \bigwedge_{d \not\trianglelefteq (+,P)} x_d = 0.$

For proving the computability and correctness of $\mathrm{cnt}_{D,\trianglelefteq}$, we define the binary relation $\triangleleft \subseteq \trianglelefteq$ as follows:

$$d_1 \triangleleft d_2 \iff d_1 \trianglelefteq d_2 \wedge \neg(d_2 \trianglelefteq^+ d_2).$$

The relation $\triangleleft^+$ is irreflexive. Every irreflexive and transitive relation over a finite set is well-founded, and so it follows that $\triangleleft^+$ is well-founded as well.

Lemma 3.4.20. *The function* $\mathrm{cnt}_{D,\trianglelefteq}$ *is computable.*

Proof. To show this, observe that if in evaluating $\mathrm{cnt}_{D,\trianglelefteq}(d)$, we recursively call $\mathrm{cnt}_{D,\trianglelefteq}(d')$ for some $d' \in D$, then $d' \triangleleft d$. Since $\triangleleft$ is well-founded, it follows that the chain of recursive calls is finite. Most of the other operations are straightforward to implement. For representing elements of $\mathbb{N} \cup \{\omega\}$, an abstract data type should be used that can represent any natural number as well as the constant ω. Each of the formulas ψ appearing an expression $M(\psi)$ are formulas in Presburger arithmetic, and thus $M(\psi)$ is effectively a semilinear set [62]. It follows that one can easily decide whether $|M(\psi)| = \omega$ and enumerate the vectors if $M(\psi)$ is finite.

□

Before we can prove the claim made in equation (3.13), we need to show how the edge relation $\trianglelefteq_{\mathcal{A}}$ can be used to detect when $\mathrm{Can}_{\mathcal{R}_\mathrm{I}/\mathcal{E}}$ contains an infinite number of equivalence classes with a given profile. To show this, we first define the *size* of a term $t \in T_\Sigma$, denoted $\mathrm{size}(t)$ to be the number of symbols in t. Since the associativity and commutativity equations in $\mathcal{E}_{\mathrm{AC}}$ preserve the size of a term, one can observe that if $t =_{\mathcal{E}_{\mathrm{AC}}} u$, then $\mathrm{size}(t) = \mathrm{size}(u)$.

Lemma 3.4.21. *If* $d_1 \trianglelefteq^+_{\mathcal{A}} d_2$ *for* $d_1, d_2 \in D_{\mathcal{A}}$*, then for all* $[t_1] \in \mathrm{Can}_{\mathcal{R}_\mathrm{I}/\mathcal{E}}$ *such that* $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_1]_\mathcal{E}) = d_1$*, there* $\exists [t_2] \in \mathrm{Can}_{\mathcal{R}_\mathrm{I}/\mathcal{E}}$ *such that* $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_2]_\mathcal{E}) = d_2$ *and* $\mathrm{size}(t_2 \downarrow_{\hat{\mathcal{R}}_\mathrm{I}/\mathcal{E}_{\mathrm{AC}}}) > \mathrm{size}(t_1 \downarrow_{\hat{\mathcal{R}}_\mathrm{I}/\mathcal{E}_{\mathrm{AC}}})$.

Proof. We prove this by induction on the length of the chain of inferences used to show $d_1 \trianglelefteq^+_{\mathcal{A}} d_2$.

The inductive case is easier, and so we prove it first. In this case, we know there is a profile $d' \in D_{\mathcal{A}}$ such that $d_1 \trianglelefteq^+_{\mathcal{A}} d' \trianglelefteq^+_{\mathcal{A}} d_2$. By our first induction hypothesis we know that there is an equivalence class $[t'] \in \mathrm{Can}_{\hat{\mathcal{R}}_\mathrm{I}/\mathcal{E}_{\mathrm{AC}}}$ such that $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t']_\mathcal{E}) = d$ and $\mathrm{size}(t_1) < \mathrm{size}(t')$. Our second induction hypothesis

then implies the existence of $[t_2] \in \mathrm{Can}_{\hat{\mathcal{R}}_\mathrm{I}/\mathcal{E}_\mathrm{AC}}$ such that $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_2]_\mathcal{E}) = d_2$ and $\mathrm{size}(t_1) < \mathrm{size}(t') < \mathrm{size}(t_2)$.

In the base case, we know that $d_1 \trianglelefteq_\mathcal{A} d_2$. By the definition of $\trianglelefteq_\mathcal{A}$, there must be equivalence classes $[u], [v] \in \mathrm{Can}_{\mathcal{R}_\mathrm{I}/\mathcal{E}}$ such that $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([u]) = d_1$, $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([v]) = d_2$, and $[u] \trianglelefteq_\mathrm{flat} [v]$. Assuming $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_1]) = d_1$, we construct t_2 by analyzing why $[u] \trianglelefteq_\mathrm{flat} [v]$. There are two cases to consider:

- If $v \downarrow_{\hat{\mathcal{R}}_\mathrm{I}/\mathcal{E}_\mathrm{AC}} =_{\mathcal{E}_\mathrm{AC}} f(v_1, \ldots, v_n)$ where f is a free symbol and $u =_{\mathcal{E}_\mathrm{AC}} v_i$ for some $i \in [1, n]$, then we let

  $$t_2 = f(v_1, \ldots, v_{i-1}, t_1, v_{i+1}, \ldots, v_n).$$

  Clearly, $[t_1] \trianglelefteq_\mathrm{flat} [t_2]$. We know that $\mathrm{states}_{\mathcal{A}/\mathcal{E}}([t_1]) = \mathrm{states}_{\mathcal{A}/\mathcal{E}}([u])$, and consequently $\mathrm{states}_{\mathcal{A}/\mathcal{E}}([t_2]) = \mathrm{states}_{\mathcal{A}/\mathcal{E}}([v])$ by Lemma 3.4.7. As $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([v]) = d_2 = (f, \mathrm{states}_{\mathcal{A}/\mathcal{E}}([v]))$, it follows that $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_2]) = d_2$. Finally,

  $$\mathrm{size}(t_2 \downarrow_{\hat{\mathcal{R}}_\mathrm{I}/\mathcal{E}_\mathrm{AC}}) = 1 + \sum_{j \in [1,n] \setminus \{i\}} \mathrm{size}(v_j) + \mathrm{size}(t_1 \downarrow_{\hat{\mathcal{R}}_\mathrm{I}/\mathcal{E}_\mathrm{AC}}),$$

  and thus $\mathrm{size}(t_2 \downarrow_{\hat{\mathcal{R}}_\mathrm{I}/\mathcal{E}_\mathrm{AC}}) > \mathrm{size}(t_1 \downarrow_{\hat{\mathcal{R}}_\mathrm{I}/\mathcal{E}_\mathrm{AC}})$.

- Otherwise, $v \downarrow_{\hat{\mathcal{R}}_\mathrm{I}/\mathcal{E}_\mathrm{AC}} =_{\mathcal{E}_\mathrm{AC}} v_1 + \cdots + v_n$ with $+$ an AC or ACI symbol, $n \geq 2$, $\mathrm{root}(v_i) \neq +$ for all $i \in [1, n]$, and $u =_{\mathcal{E}_\mathrm{AC}} v_i$ for some $i \in [1, n]$. If $t_1 \downarrow_{\hat{\mathcal{R}}_\mathrm{I}/\mathcal{E}_\mathrm{AC}} =_{\mathcal{E}_\mathrm{AC}} v_j$ for some $j \in [1, n]$, then we let $t_2 = u$ and it trivially follows that $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_2]) = d_2$ and $\mathrm{size}(t_2 \downarrow_{\hat{\mathcal{R}}_\mathrm{I}/\mathcal{E}_\mathrm{AC}}) > \mathrm{size}(t_1 \downarrow_{\hat{\mathcal{R}}_\mathrm{I}/\mathcal{E}_\mathrm{AC}})$. Otherwise, we let

  $$t_2 = v_1 + \cdots + v_{i-1} + t_1 \downarrow_{\hat{\mathcal{R}}_\mathrm{I}/\mathcal{E}_\mathrm{AC}} + v_{i+1} + \cdots + v_n.$$

  We know that $t_1 \downarrow_{\hat{\mathcal{R}}_\mathrm{I}/\mathcal{E}_\mathrm{AC}} \neq_{\mathcal{E}_\mathrm{AC}} v_i$ for all $i \in [1, n]$, and thus t_2 is $\hat{\mathcal{R}}_\mathrm{I}/\mathcal{E}_\mathrm{AC}$-irreducible. It follows that $\mathrm{size}(t_2 \downarrow_{\hat{\mathcal{R}}_\mathrm{I}/\mathcal{E}_\mathrm{AC}}) > \mathrm{size}(t_1 \downarrow_{\hat{\mathcal{R}}_\mathrm{I}/\mathcal{E}_\mathrm{AC}})$. Since $n > 2$, $\mathrm{root}(u) = +$, and thus $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([u]) = d_1 = (+, P)$ for some $P \subseteq Q$. It follows by Lemma 3.4.14 that

  $$\#(\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([u_1]), \ldots, \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([u_n])) \in M(\psi_{+,P}).$$

  As $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_1]) = \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([u_1])$, it follows $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_2]) = d_2$.

□

We can use the previous lemma to make the following observation:

Corollary 3.4.22. *For all $d \in D_\mathcal{A}$, if $d \trianglelefteq_\mathcal{A}^+ d$, then $\left|\mathrm{profile}^{-1}_{D_\mathcal{A}, \trianglelefteq_\mathcal{A}}(d)\right| = \omega$.*

Proof. For all $d \in D_\mathcal{A}$, there is a $[t] \in \mathrm{Can}_{\mathcal{R}_\mathrm{I}/\mathcal{E}}$ such that $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t]) = d$. If $d \trianglelefteq_\mathcal{A}^+ d$, then we can use Lemma 3.4.21 to construct an infinite sequence $[t_1], [t_2], \cdots \in \mathrm{Can}_{\mathcal{R}_\mathrm{I}/\mathcal{E}}$ of equivalence classes each with profile d and where

$\text{size}(t_i \downarrow_{\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}}) < \text{size}(t_j \downarrow_{\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}})$ for $i < j$. It follows that for all distinct $i, j \in \mathbb{N}$, $t_i \downarrow_{\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}} \neq_{\mathcal{E}_{\mathrm{AC}}} t_j \downarrow_{\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}}$, and thus are also distinct modulo $\mathcal{E}$. Consequently, $\text{Can}_{\mathcal{R}_{\mathrm{I}}/\mathcal{E}}$ contains an infinite number of equivalence classes with profile d. □

We are now ready to prove our previous claim in equation (3.13).

Lemma 3.4.23. *For all profile graphs $(D, \trianglelefteq) \subseteq (D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$ and $d \in D$,*

$$\left|\text{profile}^{-1}_{D,\trianglelefteq}(d)\right| \leq \text{cnt}_{D,\trianglelefteq}(d) \leq \left|\text{profile}^{-1}_{D_{\mathcal{A}},\trianglelefteq_{\mathcal{A}}}(d)\right|. \tag{3.13}$$

Proof. We prove (3.13) for all $d \in D$ by induction on d with respect to the well-founded relation $\triangleleft$. In our inductive proof, there are four cases to consider:

- If $d \trianglelefteq^{+} d$, then $\text{cnt}_{D,\trianglelefteq}(d) = \omega$, and thus $\left|\text{profile}^{-1}_{D,\trianglelefteq}(d)\right| \leq \text{cnt}_{D,\trianglelefteq}(d)$. On the other hand, as $\trianglelefteq$ is a subset of $\trianglelefteq_{\mathcal{A}}$, we know that $d \trianglelefteq^{+}_{\mathcal{A}} d$. By Cor. 3.4.22, it follows that $\left|\text{profile}^{-1}_{D_{\mathcal{A}},\trianglelefteq_{\mathcal{A}}}(d)\right| = \omega$.

- If $d = (c, P)$ with c a constant, then because $D \subseteq D_{\mathcal{A}}$, we know there is an equivalence class $[t] \in \text{Can}_{\mathcal{R}_{\mathrm{I}}/\mathcal{E}}$ such that $\text{root}(t \downarrow_{\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}}) = c$ and $\text{states}_{\mathcal{A}/\mathcal{E}}([t]) = P$. However, $\text{root}(t \downarrow_{\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}}) = c$ implies that $t =_{\mathcal{E}} c$. As there is only one equivalence class containing c, it follows that $\left|\text{profile}^{-1}_{D,\trianglelefteq}(d)\right| = 1$ and $\left|\text{profile}^{-1}_{D_{\mathcal{A}},\trianglelefteq_{\mathcal{A}}}(d)\right| = 1$.

- If $d = (f, P)$ with f a free symbol with arity $n > 0$, then by using Lemma 3.4.8 with both $(D, \trianglelefteq)$ and $(D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$, we can reduce (3.13) for d to the problem of showing (3.13) for all $d' \trianglelefteq d$. However, this follows trivially by our induction hypothesis as $d \not\trianglelefteq^{+} d$ and $d' \trianglelefteq d$ implies $d' \triangleleft d$.

- If $d = (\circ, P)$ with $\circ \in \Sigma$ idempotent in $\mathcal{E}$ or $\mathcal{R}$, then we first note that for all $d' \in D$, $d' \trianglelefteq d \implies d' \triangleleft d$ as $d \not\trianglelefteq^{+} d$. It follows that we may assume that equation (3.13) holds for each $d' \trianglelefteq d$. This implies that $M(\psi_{\circ,P,\text{Can}_{D,\trianglelefteq}}) \subseteq M(\psi_{\circ,P,\mathrm{I}}) \subseteq M(\psi_{\circ,P,\text{Can}_{D_{\mathcal{A}},\trianglelefteq_{\mathcal{A}}}})$, and consequently by using Lemma 3.4.17, we can reduce the problem of showing (3.13) for all $d \in D$ to two problems: (1) for all $d' \trianglelefteq d$ and $k \leq \left|\text{profile}^{-1}_{D,\trianglelefteq}(d')\right|$,

  $$\text{C}(\left|\text{profile}^{-1}_{D,\trianglelefteq}(d')\right|, k) \leq \text{C}(\text{cnt}_{D,\trianglelefteq}(d'), k),$$

  and (2) for all for all $d' \trianglelefteq d$ and $k \leq \text{cnt}_{D,\trianglelefteq}(d')$,

  $$\text{C}(\text{cnt}_{D,\trianglelefteq}(d'), k) \leq \text{C}(\left|\text{profile}^{-1}_{D_{\mathcal{A}},\trianglelefteq_{\mathcal{A}}}(d')\right|, k).$$

  Both of these problems follow easily from our induction hypothesis and the definition of C.

Starting with the empty graph $(D_0, \trianglelefteq_0) = (\varnothing, \varnothing)$, we freely apply either of the rules below to construct $(D_{i+1}, \trianglelefteq_{i+1})$ from $(D_i, \trianglelefteq_i)$ subject to the condition that a rule may only be applied if the resulting graph $(D_{i+1}, \trianglelefteq_{i+1})$ is distinct from $(D_i, \trianglelefteq_i)$. The rules are applied until completion to obtain the graph $(D_*, \trianglelefteq_*)$.

$$\frac{\text{choose free symbol } f \in \Sigma \text{ and } (f_1, P_1), \ldots, (f_n, P_n) \in D_i}{\begin{array}{l} D_{i+1} := D_i \cup \{\, (f, \mathrm{states}_f(P_1, \ldots, P_n)) \,\} \\ \trianglelefteq_{i+1} := \trianglelefteq_i \cup \{\, ((f_j, P_j), (f, \mathrm{states}_f(P_1, \ldots, P_n))) \mid j \in [1, n] \,\} \end{array}}$$

$$\frac{\text{choose AC or ACI symbol } + \in \Sigma \text{ and } P \subseteq Q \text{ s.t. } (\exists \overline{x})\, \psi_{+,P,D_i,\trianglelefteq_i}(\overline{x})}{\begin{array}{l} D_{i+1} := D_i \cup \{\, (f, P) \,\} \\ \trianglelefteq_{i+1} := \trianglelefteq_i \cup \{\, (d, (f, P)) \mid d \in D_i \wedge (\exists \overline{x})\, \psi_{+,P,D_i,\trianglelefteq_i}(\overline{x}) \wedge x_d > 0 \} \end{array}}$$

where for each AC symbol $\circ \in \Sigma$ that is idempotent in $\mathcal{E}$ or $\mathcal{R}$ and each AC symbol $+ \in \Sigma$ that is not idempotent in $\mathcal{E}$ or $\mathcal{R}$, we let

$$\psi_{\circ,P,D_i,\trianglelefteq_i}(\overline{x}) = \psi_{\circ,P}(\overline{x}) \wedge \bigwedge_{d \in D(\circ) \setminus D_i} x_d = 0 \wedge \bigwedge_{d \in D(\circ) \cap D_i} x_d \leq \mathrm{cnt}_{D_i,\trianglelefteq_i}(d)$$

$$\psi_{+,P,D_i,\trianglelefteq_i}(\overline{x}) = \psi_{+,P}(\overline{x}) \wedge \bigwedge_{d \in D(+) \setminus D_i} x_d = 0 \wedge \bigwedge_{\substack{d \in D(+) \cap D_i \\ \mathrm{cnt}_{D_i,\trianglelefteq_i}(d) = 0}} x_d = 0.$$

Figure 3.4: Inference System for Constructing $(D_*, \trianglelefteq_*)$

- Otherwise $d = (+, P)$ with $+ \in \Sigma$ an AC symbol that is not idempotent in $\mathcal{R}$. We first note that for all $d' \trianglelefteq d$, $d' \triangleleft d$ as $d \not\trianglelefteq^+ d$. It follows that we may assume (3.13) for each $d' \trianglelefteq d$. This implies that $M(\psi_{+,P,\mathrm{Can}_{D,\trianglelefteq}}) \subseteq M(\psi_{+,P,\mathrm{AC}}) \subseteq M(\psi_{+,P,\mathrm{Can}_{D_\mathcal{A},\trianglelefteq_\mathcal{A}}})$, and consequently by using Lemma 3.4.18, we can reduce the problem of showing (3.13) for all $d \in D$ to two problems: (1) for all $d' \trianglelefteq d$ and $k \in \mathbb{N}$ where $\left|\mathrm{profile}^{-1}_{D,\trianglelefteq}(d')\right| > 0$,

  $$\mathrm{C}(\mathrm{cnt}_{D,\trianglelefteq}(d') + k - 1, k) \leq \mathrm{C}(\left|\mathrm{profile}^{-1}_{D_\mathcal{A},\trianglelefteq_\mathcal{A}}(d')\right| + k - 1, k),$$

  and (2) for all $d' \trianglelefteq d$ and $k \in \mathbb{N}$ where $\mathrm{cnt}_{D,\trianglelefteq}(d') > 0$,

  $$\mathrm{C}(\mathrm{cnt}_{D,\trianglelefteq}(d') + k - 1, k) \leq \mathrm{C}(\left|\mathrm{profile}^{-1}_{D_\mathcal{A},\trianglelefteq_\mathcal{A}}(d')\right| + k - 1, k).$$

  Both of these problems follow easily from our induction hypothesis and the definition of the choose function C.

□

3.4.5 Constructing the profile graph

The algorithm for constructing the profile graph $(D_*, \trianglelefteq_*)$ is given Figure 3.4. We show that $(D_*, \trianglelefteq_*) = (D_\mathcal{A}, \trianglelefteq_\mathcal{A})$ in two steps. First, we show that $(D_*, \trianglelefteq_*) \subseteq (D_\mathcal{A}, \trianglelefteq_\mathcal{A})$ by showing that if $(D_i, \trianglelefteq_i) \subseteq (D_\mathcal{A}, \trianglelefteq_\mathcal{A})$, then any graph $(D_{i+1}, \trianglelefteq_{i+1})$

obtained by applying one of the inference rules in Figure 3.4 is a subgraph of $(D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$. Since the initial graph $(D_0, \trianglelefteq) = (\varnothing, \varnothing) \subseteq (D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$, this implies that $(D_*, \trianglelefteq_*) \subseteq (D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$. Second, we prove that $\mathrm{Can}_{D_*, \trianglelefteq_*} = \mathrm{Can}_{\mathcal{R}_{\mathrm{I}}/\mathcal{E}}$, which by Lemma 3.4.5 implies that $(D_*, \trianglelefteq_*) = (D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$.

The following lemma is essential to showing that $(D_*, \trianglelefteq_*) \subseteq (D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$:

Lemma 3.4.24. *For all $(D_i, \trianglelefteq_i) \subseteq (D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$, if $(D_{i+1}, \trianglelefteq_{i+1})$ is obtained from $(D_i, \trianglelefteq_i)$ by an inference step using the rules in Figure 3.4, then $(D_{i+1}, \trianglelefteq_{i+1}) \subseteq (D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$.*

Proof. We consider three different cases separately:

- In the first case, suppose $(D_{i+1}, \trianglelefteq_{i+1})$ is obtained by applying the first rule after choosing the free symbol $f \in \Sigma$ and profiles $(f_1, P_1), \ldots, (f_n, P_n) \in D_i$. Let $P = \mathrm{states}_f(P_1, \ldots, P_n)$. We must show that $(f, P) \in D_{\mathcal{A}}$, and $(f_j, P_j) \trianglelefteq_{\mathcal{A}} (f, P)$ for all $j \in [1, n]$. As $D_i \subseteq D_{\mathcal{A}}$, we know that for each $j \in [1, n]$, there is an equivalence class $[t_j] \in \mathrm{Can}_{\mathcal{R}_{\mathrm{I}}/\mathcal{E}}$ such that $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_j]) = (f_j, P_j)$. Let $t = f(t_1, \ldots, t_n)$. As f is free, we know that $[t] \in \mathrm{Can}_{\mathcal{R}_{\mathrm{I}}/\mathcal{E}}$, and therefore $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t]) \in D_{\mathcal{A}}$ Observe that $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t]) = (f, P)$ by Lemma 3.4.7, and thus $(f, P) \in D_{\mathcal{A}}$. For $j \in [1, n]$, observe that $[t_j] \trianglelefteq_{\mathrm{flat}} [t]$, and thus $(f_j, P_j) \trianglelefteq_{\mathcal{A}} (f, P)$.

- In the second case, suppose $(D_{i+1}, \trianglelefteq_{i+1})$ is obtained by applying the second rule after choosing the symbol $\circ \in \Sigma$ that is idempotent in $\mathcal{E}$ or $\mathcal{R}$ and choosing a set $P \subseteq Q$. It is sufficient to show that $(\circ, P) \in D_{\mathcal{A}}$, and for each $\overline{x} \in M(\psi_{\circ, P, D_i, \trianglelefteq_i})$, if $x_d > 0$, then $d \trianglelefteq_{\mathcal{A}} (\circ, P)$. We know that there is at least one $\overline{x} \in M(\psi_{\circ, P, D_i, \trianglelefteq_i})$. For each $d \in D(\circ) \cap D_i$, we know that $x_d \leq \mathrm{cnt}_{D_i, \trianglelefteq_i}(d)$. By Lemma 3.4.23, it follows that there are at least x_d distinct equivalence classes $[t_{d(1)}], \ldots, [t_{d(x_d)}] \in \mathrm{profile}^{-1}_{D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}}}(D)$. Without loss of generality, we may assume that each term $t_{d(i)}$ is $\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}$-irreducible. Let $t = t_1 \circ \cdots \circ t_n$ be a term where each term t_j corresponds to a unique term $t_{d(k)}$ for some $d \in D_i$ and $k \in [1, x_d]$. The term t is $\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}$-irreducible, and $\#(\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_1]), \ldots, \mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t_n])) = \overline{x}$. It follows that $\mathrm{profile}_{\mathcal{A}/\mathcal{E}}([t]) = (\circ, P)$ by Lemma 3.4.14, and so $(\circ, P) \in D_{\mathcal{A}}$. For each $d \in D(\circ)$, if $x_d > 0$, then $[t_{d(1)}] \trianglelefteq_{\mathrm{flat}} [t]$, and consequently $d \trianglelefteq_{\mathcal{A}} (\circ, P)$.

- In the third case, suppose $(D_{i+1}, \trianglelefteq_{i+1})$ is obtained by applying the second rule after choosing the symbol $+ \in \Sigma$ that is AC in $\mathcal{E}$ and not idempotent in $\mathcal{R}$ and choosing a set $P \subseteq Q$. It is enough to show that $(+, P) \in D_{\mathcal{A}}$, and for each $\overline{x} \in M(\psi_{+, P, D_i, \trianglelefteq_i})$, if $x_d > 0$, then $d \trianglelefteq_{\mathcal{A}} (+, P)$. We know that there is at least one $\overline{x} \in M(\psi_{+, P, D_i, \trianglelefteq_i})$. For each $d \in D(+)$, if $x_d > 0$, then we know $d \in D_i$ and $\mathrm{cnt}_{D_i, \trianglelefteq_i}(d) > 0$. It follows by Lemma 3.4.23 that there is an equivalence class $[t_d] \in \mathrm{Can}_{D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}}}$ with profile d. Without loss of generality, we may assume that t_d is $\hat{\mathcal{R}}_{\mathrm{I}}/\mathcal{E}_{\mathrm{AC}}$-irreducible. Let $t = t_1 + \cdots + t_n$ be a term in which for each $d \in D(+)$, there are exactly x_d distinct indices $d(1), \ldots, d(x_d) \in [1, n]$ such that $t_{d(i)} = t_d$.

It follows that $\#(\text{profile}_{\mathcal{A}/\mathcal{E}}([t_1]), \ldots, \text{profile}_{\mathcal{A}/\mathcal{E}}([t_n])) = \overline{x}$, and consequently $\text{profile}_{\mathcal{A}/\mathcal{E}}([t]) = (+, P)$ by Lemma 3.4.14, and so $(+, P)$. For each $d \in D(+)$ if $x_d > 0$, then $[t_d] \trianglelefteq_{\text{flat}} [t]$, and thus $d \trianglelefteq_{\mathcal{A}} (+, P)$.

□

The previous lemma implies that $(D_n, \trianglelefteq_n) \subseteq (D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$ for all $n \in N$. Since $(D_*, \trianglelefteq_*) = (D_n, \trianglelefteq_n)$ for some $n \in \mathbb{N}$, it follows that $(D_*, \trianglelefteq *) \subseteq (D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$.

Corollary 3.4.25.

$$(D_*, \trianglelefteq_*) \subseteq (D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$$

□

We now show that $(D_*, \trianglelefteq_*)$ has explored all the elements in $\text{Can}_{\mathcal{R}_{\text{I}}/\mathcal{E}}$.

Lemma 3.4.26.

$$\text{Can}_{D_*, \trianglelefteq_*} = \text{Can}_{\mathcal{R}_{\text{I}}/\mathcal{E}}.$$

Proof. As $\hat{\mathcal{R}}_{\text{I}}$ is confluent and terminating, it is enough to show by structural induction that for each $\hat{\mathcal{R}}_{\text{I}}/\mathcal{E}_{\text{AC}}$-irreducible term t, the equivalence class $[t]_{\mathcal{E}} \in \text{Can}_{D_*, \trianglelefteq_*}$ There are three cases to consider:

- In the first case, suppose $t = f(t_1, \ldots, t_n)$ with f a free symbol. By induction $t_i \in \text{Can}_{D_*, \trianglelefteq_*}$ for $i \in [1, n]$, and consequently $\text{profile}_{\mathcal{A}/\mathcal{E}}(t_i) \in D_*$. Let $\text{profile}_{\mathcal{A}/\mathcal{E}}(t_i) = (f_i, P_i)$, and let $\text{states}_{\mathcal{A}/\mathcal{E}}([t]) = P$. By Lemma 3.4.7, we know that $\text{states}_f(P_1, \ldots, P_n) = P$. According to the the first rule in Figure 3.4, we know that $\text{profile}_{\mathcal{A}/\mathcal{E}}([t]) \in D_*$ and for all $i \in [1, n]$, $\text{profile}_{\mathcal{A}/\mathcal{E}}([t_i]) \trianglelefteq_* \text{profile}_{\mathcal{A}/\mathcal{E}}([t])$. Consequently, $[t] \in \text{Can}_{D_*, \trianglelefteq_*}$.

- In the second case, suppose $t = t_1 \circ \cdots \circ t_n$ where $\circ$ an symbol that is idempotent in $\mathcal{E}$ or $\mathcal{R}$ and $\text{root}(t_i) \neq \circ$ for $i \in [1, n]$. By induction we know that $t_i \in \text{Can}_{D_*, \trianglelefteq_*}$ for $i \in [1, n]$. Let $P = \text{states}_{\mathcal{A}/\mathcal{E}}([t])$ and let $\overline{x} = \#(\text{profile}_{\mathcal{A}/\mathcal{E}}([t_1]), \ldots, \text{profile}_{\mathcal{A}/\mathcal{E}}([t_n]))$. By Lemma 3.4.14 we know that $\overline{x} \in M(\psi_{\circ, P})$. As all of the subterms $t_1, \ldots, t_n$ are distinct and also in $\text{Can}_{D_*, \trianglelefteq_*}$, we know by Lemma 3.4.23 that $x_d \leq \text{cnt}_{D_i, \trianglelefteq_i}(d)$. It follows that $\overline{x} \in M(\psi_{\circ, P, D_i, \trianglelefteq_i})$, and by the second rule in Figure 3.4, we know that $\text{profile}_{\mathcal{A}/\mathcal{E}}([t]) \in D_*$ and for all $i \in [1, n]$, $\text{profile}_{\mathcal{A}/\mathcal{E}}([t_i]) \trianglelefteq \text{profile}_{\mathcal{A}/\mathcal{E}}([t])$. Consequently, $[t] \in \text{Can}_{D_*, \trianglelefteq_*}$.

- In the final case, suppose $t = t_1 + \cdots + t_n$ with $+$ an AC symbol in $\mathcal{E}$ that is not idempotent in $\mathcal{R}$ and $\text{root}(t_i) \neq +$ for $i \in [1, n]$. By induction we know that $t_i \in \text{Can}_{D_*, \trianglelefteq_*}$ for $i \in [1, n]$. Let $P = \text{states}_{\mathcal{A}/\mathcal{E}}([t])$ and let $\overline{x} = \#(\text{profile}_{\mathcal{A}/\mathcal{E}}([t_1]), \ldots, \text{profile}_{\mathcal{A}/\mathcal{E}}([t_n]))$. By Lemma 3.4.14 we know that $\overline{x} \in M(\psi_{+, P})$. As all of the subterms $t_1, \ldots, t_n$ are in $\text{Can}_{D_*, \trianglelefteq_*}$, we know by Lemma 3.4.23 that if $x_d > 0$, then $\text{cnt}_{D_i, \trianglelefteq_i}(d) > 0$. It follows that $\overline{x} \in M(\psi_{+, P, D_i, \trianglelefteq_i})$, and according to the second rule in Figure 3.4,

we know that $\text{profile}_{\mathcal{A}/\mathcal{E}}([t]) \in D_*$ and for all $i \in [1,n]$, $\text{profile}_{\mathcal{A}/\mathcal{E}}([t_i]) \trianglelefteq \text{profile}_{\mathcal{A}/\mathcal{E}}([t])$. Consequently, $[t] \in \text{Can}_{D_*,\trianglelefteq_*}$.

□

We are now able to prove the main result of this section.

Theorem 3.4.27. *The graph $(D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$ is effectively constructable.*

Proof. We know by Lemma 3.4.26 that $\text{Can}_{D_*,\trianglelefteq_*} = \text{Can}_{\mathcal{R}_\text{I}/\mathcal{E}}$. As $(D_*, \trianglelefteq_*)$ is a subgraph of $(D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$ by Cor. 3.4.25, it follows by Lemma 3.4.5 that $(D_*, \trianglelefteq_*) = (D_{\mathcal{A}}, \trianglelefteq_{\mathcal{A}})$. However, $(D_*, \trianglelefteq_*)$ can be constructed by applying each inference rule in Figure 3.4 a finite number of times. It is decidable whether an inference rule can be applied, because each choice ranges over a finite set, the function $\text{cnt}_{D,\trianglelefteq}$ is computable by Lemma 3.4.20 and each formula $\psi_{\circ,P,D_i,\trianglelefteq_i}$ is expressible in Presburger arithmetic after the value for $\text{cnt}_{D_i,\trianglelefteq_i}(d)$ has been replaced with its computed value.

□

Theorem 3.4.2 can be obtained as a corollary of Theorem 3.4.27.

Theorem 3.4.2. *Let $\mathcal{E}$ be a theory with free, AC, and ACI symbols, and let $\mathcal{R}_\text{I}$ be a set of rewrite rules which may contain an idempotence rule for any of the AC symbols in $\mathcal{E}$.*

Given an AC intersection free $\mathcal{E}$-tree automaton $\mathcal{A}$, and propositional formula ϕ over the states in $\mathcal{A}$, the question of whether $\mathcal{L}_\phi(\mathcal{A}/\mathcal{E}) \cap \text{Can}_{\mathcal{R}_\text{I}/\mathcal{E}} = \varnothing$ is decidable.

Proof. By structural induction on ϕ, it is easy to show that

$$\mathcal{L}_\phi(\mathcal{A}/\mathcal{E}) = \{\, [t] \in T_{\mathcal{E}} \mid \text{states}_{\mathcal{A}/\mathcal{E}}([t]) \models \phi \,\}.$$

It follows that

$$\begin{aligned}\text{Can}_{\mathcal{R}_\text{I}/\mathcal{E}} \cap \mathcal{L}_\phi(\mathcal{A}/\mathcal{E}) \neq \varnothing \iff (\exists [t] \in \text{Can}_{\mathcal{R}_\text{I}/\mathcal{E}})\, \text{states}_{\mathcal{A}/\mathcal{E}}([t]) \models \phi \\ \iff (\exists (f,P) \in D_{\mathcal{A}})\, P \models \phi.\end{aligned}$$

Since $D_{\mathcal{A}}$ is finite and effectively constructable by Theorem 3.4.27, it follows that the question of whether $\text{Can}_{\mathcal{R}_\text{I}/\mathcal{E}} \cap \mathcal{L}_\phi(\mathcal{A}/\mathcal{E}) = \varnothing$ is decidable. □

3.5 Related work and conclusions

Our main contributions in this chapter are: (1) We have shown that each alternating equational tree language can be expressed as the intersection of two regular equational tree languages by adding a fresh ACI symbol to the theory. This implies that intersection emptiness is undecidable for regular equational

tree automata over a theory with both AC and ACI symbols. (2) We studied modularity in equational tree automata and have shown that both intersection emptiness and propositional emptiness are non-modular properties even for disjoint theories. (3) We presented a practical semi-algorithm for checking propositional emptiness of theories with A and AC symbols that has been implemented in the CETA tree automaton library. (4) We presented a subclass of regular equational tree automata over theories with AC and ACI symbols and have shown that propositional emptiness is decidable for that subclass.

One of our goals was to obtain decidability results over non-linear theories. In this direction there are numerous papers on extending tree automata techniques to better handle non-linearity in adding constraints to the automata rules [33, Chapter 4] as well as extending that idea to handle some equational theories [80, 106]. The problem of deciding whether a non-equational tree language accepts an irreducible term for any set of linear or non-linear rules was shown in [32], however the approach used here is quite different. The technique of counting the number of distinct terms was influenced by similar issues in deciding the emptiness of multitree automata [105], and our realization that Presburger arithmetic is useful in the ACI case was inspired by the generalization of Parikh's theorem to arbitrary Kleene algebras in [78].

Although we have solved two open problems, our work suggests additional questions that are worth exploring. One interesting direction would be to investigate imposing stronger conditions on the theories such as linearity or collapse-freeness to attempt finding a positive modularity result. A second direction would be to combine the semi-decision procedure for the associative case in Section 3.3 with the algorithm for AC-intersection free automata to handle a large class of automata over theories with any combination of associativity, commutativity, and idempotence.

Chapter 4

Order-sorted equational unification

This chapter introduces an approach to performing order-sorted equational unification – a fundamental operation used in many different applications. The main idea behind unification is to generate a set of solutions which represent all the solutions to a system of equations. In the context of unification, the solutions are substitutions called *unifiers* and the system of equations is called a *unification problem*. In this section, we focus on unification problems in the context of *order-sorted* theories $\mathcal{E} = (F, E)$.

The main application that we have been using unification for is to solve reachability problems using narrowing. In narrowing, unification is a fundamental operation that is called many times to unify terms representing reachable states against the left-hand sides of rewrite rules. This process is computationally expensive and often generates a large number of different terms — many of which may represent states that do not correspond to legal states. In order to avoid this problem tools such as the Maude-NRL protocol analyzer [48, 49] use order-sorted algebras and rely on the sorts to only consider well-formed terms.

In this chapter, we present an algorithm which can use a procedure for *unsorted* E-unification to perform *order-sorted* E-unification under conditions general enough to cover many practical applications. This algorithm solves a key challenge faced by the Maude-NRL protocol analyzer — most existing unification tools only support unsorted unification and ignore the sort information. Since equational unification procedures are often quite complex, it requires significantly less work to use an existing unification tool rather that writing an order-sorted equational unification procedure from scratch.

The order-sorted unification algorithm we present in this work can be naturally described by a terminating and confluent set of rewrite rules which compute order-sorted unifiers $\theta_1, \ldots, \theta_n$ for each unsorted unifier $\overline{\theta}$ returned by the unsorted unification procedure. The algorithm is implemented in Maude, and uses CiME [36] to perform unsorted equational unification. Our experimental results so far have shown that, although technically there may be many order-sorted unifiers for each unsorted unifier, this is rarely the case in practice. In practice, there are usually *fewer* order-sorted unifiers than unsorted unifiers, and the use of order-sorted unification is essential for both the correctness and performance of the Maude-NRL protocol analyzer.

The results in this chapter first appeared in [71]. The basic idea however did not originate in that paper, and appeared in [112] and more recently without a proof of correctness in [49]. However, after implementing these ideas in the Maude-NRL protocol analyzer, we decided to study the topic in more detail for several reasons:

- Our experience with the Maude-NRL protocol analyzer so far has suggested that for theories with AC operators, the use of sort information is essential for practical protocol verification tools based on narrowing. However, most existing unification procedures only perform unsorted $\mathcal{E}$-unification and do not support sorts and subsorts. By using the techniques described in this work, one can obtain an order-sorted $\mathcal{E}$-unification procedure from an unsorted one with very little effort for many equational theories.

- The algorithm in [112] was buried in a function's definition appearing in the proof of Theorem 34 in [112]. In this chapter, we present a simple rule-based algorithm which is almost directly implementable in Maude. The algorithm only consists of three confluent and terminating rewrite rules, and it should be easily possible to compose these rules with inference steps in a modular way in other reasoning tools using unification.

- Perhaps most important from a technical perspective, the correctness results in [112] imposed unnecessarily strong technical conditions which excluded the majority of $\mathcal{E}$-unification problems when $\mathcal{E}$ contains collapsing equations like idempotence $x + x = x$ and identity $x + 0 = x$. As identity was important for the Maude-NRL protocol analyzer and idempotence is a common axiom in many $\mathcal{E}$-unification algorithms, in this chapter we prove the correctness results under weaker assumptions about the equational theory and some technical assumptions about the unification engine. The assumptions about the unification engine should be satisfied in practice. Additionally, we show specifically how the algorithm can be used in Maude for equational theories with any combination of free, commutative, AC, and ACU symbols.

This chapter is organized as follows. In Section 4.1, we review basic definitions of order-sorted unification and present our algorithm to compute order-sorted unifiers from unsorted unifiers. In Section 4.2, we illustrate how it can be used for AC and ACU order-sorted unification in Maude and, in Section 4.3 we prove its correctness. Finally, in Section 4.4, we discuss related work and suggest directions for future research.

4.1 General order-sorted equational unification

Our main goal in this chapter is to develop a clear rule-based algorithm for solving order-sorted $\mathcal{E}$-unification problems using an unsorted $\overline{\mathcal{E}}$-unification procedure. In order to show that the rule-based algorithm returns a complete set of most-general unifiers, there are some technical requirements placed on the order-sorted theory $\mathcal{E}$ as well as on the most-general unifiers $\overline{U}$ returned by the unsorted $\overline{\mathcal{E}}$-unification procedure. The basic techniques behind our algorithm were described in [112]. However the correctness shown in [112] imposed conditions that are *too strong* when the theory $\mathcal{E}$ contains collapsing equations like identity or idempotence axioms.

Our approach to find suitable requirements is then to relax the requirements on $\mathcal{E}$ while making requirements on the unsorted unification procedure in relation to the theory $\mathcal{E}$. At first this appears to be less general than the approach in [112], since that paper did not make assumptions about the unsorted unification procedure. However, as we will discuss later, the theories we are interested in are such that every practical unification procedure will satisfy the requirements. Most importantly for our purposes, this includes theories with identity axioms.

We let $\mathrm{rvars}(\theta)$ denote the variables occurring in terms in the codomain of θ, i.e., $\mathrm{rvars}(\theta) = \bigcup_{x \in Y} \mathrm{vars}(x\theta)$. Given substitutions $\theta_1, \theta_2 : Y \to T_\Sigma(X)$, we write $\theta_1 =_\mathcal{E} \theta_2$ if $x\theta_1 =_\mathcal{E} x\theta_2$ for all $x \in Y$, and we write $\theta_1 \geq_\mathcal{E} \theta_2$ if there is a substitution $\psi : \mathrm{rvars}(\theta_1) \to T_\Sigma(X)$ such that $\theta_1\psi =_\mathcal{E} \theta_2$.

For a fixed order-sorted theory $\mathcal{E}$ over a signature $\Sigma = (S, F, \leq)$, we define an order-sorted *unification problem* to be a finite conjunctive set Γ of Σ-equations $t = u$ where t and u are terms in $T_\Sigma(X)$ whose sorts belong to the same connected component in $(S, \leq)$. An *$\mathcal{E}$-unifier* for Γ is an order-sorted substitution $\theta : \mathrm{vars}(\Gamma) \to T_\Sigma(X)$ such that $t\theta =_\mathcal{E} u\theta$ for each equation $t = u \in \Gamma$. We denote the set of $\mathcal{E}$-unifiers for Γ by $\mathrm{Un}_\mathcal{E}(\Gamma)$, and we let $\mathrm{Un}_\Sigma(\Gamma)$ denote the syntactic unifiers for Γ, i.e., $\mathrm{Un}_\Sigma(\Gamma) = \mathrm{Un}_\varnothing(\Gamma)$. A set $S \subseteq \mathrm{Un}_\mathcal{E}(\Gamma)$ of $\mathcal{E}$-unifiers of Γ is *complete* if for all unifiers $\psi \in \mathrm{Un}_\mathcal{E}(\Gamma)$, there is a unifier $\theta \in S$ such that $\theta \geq_\mathcal{E} \psi$. A set of $\mathcal{E}$-unifiers S is *most-general* if for distinct substitutions $\theta_1, \theta_2 \in S$, $\theta_1 \not\geq_\mathcal{E} \theta_2$. A given theory $\mathcal{E}$ has a *finitary* unification problem if there is a complete finite set of $\mathcal{E}$-unifiers S for each unification problem Γ.

In this section, we assume the following conditions on an order-sorted theory $\mathcal{E}$ over a signature Σ and the unsorted unification procedure for $\overline{\mathcal{E}}$.

(i) Σ is *preregular* [64], that is for each well-sorted term $t \in T_\Sigma(X)$ there is a *least sort* $\mathrm{ls}(t) \in S$ such that $t \in T_\Sigma(X)_{\mathrm{ls}(t)}$ and for all sorts $s \in S$,

$$t \in T_\Sigma(X)_s \implies \mathrm{ls}(t) \leq s.$$

(ii) $\mathcal{E}$ is *sort-independent* (see Def. 2.4.1) which means that for all order-sorted

terms $t, u \in T_\Sigma(X)$,

$$t =_{\overline{\mathcal{E}}} u \Rightarrow t =_{\mathcal{E}} u.$$

(iii) For each unification problem Γ, the unsorted unification procedure generates a complete finite set of most-general unifiers $\overline{U}$ which is *sort preserving*. This means that each order-sorted unifier $\psi \in \mathrm{Un}_{\mathcal{E}}(\Gamma)$ can be decomposed into an unsorted unifier $\overline{\theta} \in \overline{U}$ and an unsorted substitution $\overline{\phi} : \mathrm{rvars}(\overline{\theta}) \to T_\Sigma(X)$ such that: (1) $\psi =_{\overline{\mathcal{E}}} \overline{\theta\phi}$, and (2) $\overline{\theta\phi}$ is an order-sorted substitution.

These three requirements are rather technical, and we will later show in Section 4.2 how these three properties are satisfied for theories used by the Maude-NRL protocol analyzer. Before doing that, we prefer to focus on how these requirements can be used to simplify order-sorted unification problems.

If the equational theory $\mathcal{E}$ and unsorted $\overline{\mathcal{E}}$-unification procedure satisfy the previous requirements, as we show below, the unsorted $\overline{\mathcal{E}}$-unification procedure can be used to solve order-sorted $\mathcal{E}$-unification problems. We can split the process of solving an order-sorted unification problem $\Gamma = t_1 =_{\mathcal{E}} u_1 \wedge \cdots \wedge t_n =_{\mathcal{E}} u_n$ into two phases: an *unsorted unification* phase and a *sort propagation* phase.
Unsorted Unification. First, we call the unsorted $\overline{\mathcal{E}}$-unification procedure on the unsorted $\overline{\mathcal{E}}$-unification problem $\overline{\Gamma} = t_1 =_{\overline{\mathcal{E}}} u_1 \wedge \cdots \wedge t_n =_{\overline{\mathcal{E}}} u_n$ to obtain a finite complete set of most-general sort-preserving unifiers $\overline{U}$ for $\overline{\Gamma}$.
Sort Propagation. In the second phase, for each unsorted unifier $\overline{\theta} \in \overline{U}$, we use the *membership propagation* algorithm described below to generate a set of *variable renamings*. In this context, a variable renaming is an injective function $\rho : \mathrm{rvars}(\overline{\theta}) \to X$. For each variable renaming ρ generated for an unsorted unifier $\overline{\theta} \in \overline{U}$, our procedure returns $\overline{\theta}\rho$ as one element in the complete set of most-general unifiers.

The membership propagation algorithm is described by a set of rules that maintain a *disjunctive* set D of *membership constraints*. Each membership constraint $M \in D$ is a conjunctive formula of the form $M = t_1 : s_1 \wedge \cdots \wedge t_n : s_n$, and D is a finite set $D = \{ M_1, \ldots, M_p \}$ of membership constraints. A membership constraint M captures the sort constraints for each assignment $\overline{\theta}(x_s)$ to be a well-sorted term with sort s.

For each unsorted unifier $\overline{\theta} \in \overline{U}$, we initially generate a singleton set $D(\overline{\theta})$ reflecting the sort constraints on the variables appearing in the original unification problem Γ.

$$D(\overline{\theta}) = \{ \bigwedge_{x_s \in \mathrm{vars}(\Gamma)} x_s\overline{\theta} : s \}.$$

We then apply the three rewrite rules in Figure 4.1 to $D(\overline{\theta})$ until termination. The **Intersection** rule exploits the preregularity assumption to simplify multiple membership constraints $t : s_1$ and $t : s_2$ on the same term t. The **Propa-**

Given an initial set of membership constraints D, we freely apply the rules below to obtain a final set of constraints D^*.

Intersection $\{ t : s_1 \wedge t : s_2 \wedge M \} \rightarrow \bigcup_{s \in \mathrm{glb}_\Sigma(s_1,s_2)} \{ t : s \wedge M \}$

Propagation $\{ f(t_1,\ldots,t_n) : s \wedge M \} \rightarrow \bigcup_{s_1 \ldots s_n \in \mathrm{ar}_\Sigma(f,s,n)} \{ t_1 : s_1 \wedge \cdots \wedge t_n : s_n \wedge M \}$

Subsumption $\{ M_1, M_2 \} \rightarrow \{ M_1 \}$ if $M_1 \geq M_2$

where $\mathrm{glb}_\Sigma(s_1,s_2) = \sup_{\leq}(\{ s \in S \mid s \leq s_1 \wedge s \leq s_2 \})$,
$\mathrm{ar}_\Sigma(f,s,n) = \sup_{\leq^n}(\{ w \in S^n \mid (\exists s' \in S)\, f \in F_{w,s'} \wedge s' \leq s \})$ and
$M_1 \geq M_2 \iff (\forall\, t : s \in M_1)(\exists s' \leq s)$ s.t. $t : s' \in M_2$.

Figure 4.1: Sort Propagation Algorithm

gation rule simplifies constraints on terms $f(t_1,\ldots,t_n) : s$ to the smaller terms $t_1,\ldots,t_n$. Finally, the **Subsumption** rule is used to eliminate membership constraints that are subsumed by other more-general membership constraints. We let D^* denote the unique normal form obtained by rewriting D until completion.

Upon termination of the rules, each membership constraint $M \in D^*$ will have the form $x_1 : s_1 \wedge \cdots \wedge x_n : s_n$ with $x_i \neq x_j$ for $i \neq j$. We call membership constraints with this form *reduced*. A reduced membership constraint can be viewed as a function $\mathrm{sort}_M : \mathrm{rvars}(\overline{\theta}) \rightarrow S$ that maps each variable $x_i \in \mathrm{rvars}(\overline{\theta})$ to the sort $s_i \in S$. Furthermore, for each reduced membership constraint M, we let $\rho_M : \mathrm{rvars}(\overline{\theta}) \rightarrow X$ be a variable renaming which maps each variable $x \in \mathrm{rvars}(\overline{\theta})$ to a fresh variable $x\rho_M$ with sort $\mathrm{sort}_M(x)$.

For the set of unsorted sort-preserving unifiers $\overline{U} \subseteq \mathrm{Un}_{\overline{\mathcal{E}}}(\Gamma)$, we define the set

$$\mathrm{OS}(\overline{U}) = \{ \overline{\theta}\rho_M \mid \overline{\theta} \in \overline{U} \wedge M \in D(\overline{\theta})^* \}.$$

As an example, consider the unification problem $x_{NzNat} = y_{Nat} + z_{Nat}$ over an order-sorted theory $\mathcal{E} = (F, E)$ where $+$ contains the following operator declarations:

$$+ : \mathit{Nat}\ \mathit{Nat} \rightarrow \mathit{Nat} \quad + : \mathit{Nat}\ \mathit{NzNat} \rightarrow \mathit{NzNat} \quad + : \mathit{NzNat}\ \mathit{Nat} \rightarrow \mathit{NzNat}$$

where the declaration $f : s_1\ s_2 \rightarrow s$ means that $f \in F_{s_1 s_2, s}$. In this case, the unsorted unification engine can return a single unifier

$$\overline{\theta} = \{ x_{NzNat} \mapsto y_{Nat} + z_{Nat}, y_{Nat} \mapsto y_{Nat}, z_{Nat} \mapsto z_{Nat} \}.$$

However, $\overline{\theta}$ is not an order-sorted unifier, because that would require that $\overline{\theta}(x_{NzNat}) = y_{Nat} + z_{Nat}$ to be in $T_\Sigma(X)_{NzNat}$, and in order for that to be the case either y_{Nat} or z_{Nat} would have to have sort NzNat.

To obtain variables with the appropriate sort constraints, we pass $\overline{\theta}$ to the sort propagation algorithm, which generates the initial set of membership constraints

$$D(\overline{\theta}) = \{\, y + z : NzNat \wedge y : Nat \wedge z : Nat \,\}.$$

For this simple example, a single application of **Propagation** yields the membership constraints:

$$\begin{aligned} D(\overline{\theta})' = \{\, & (y : Nat \wedge z : NzNat \wedge y : Nat \wedge z : Nat), \\ & (y : NzNat \wedge z : Nat \wedge y : Nat \wedge z : Nat) \,\}. \end{aligned}$$

From $D(\overline{\theta})'$, we only need to apply **Intersection** several times to yield the final set of membership constraints:

$$D(\overline{\theta})^* = \{\, (y : Nat \wedge z : NzNat), (y : NzNat \wedge z : Nat) \,\}.$$

From $D(\overline{\theta})^*$, we can extract two variables renamings. When applied to the initial unsorted unifier $\overline{\theta}$, this yields the final complete set of order-sorted unifiers:

$$\begin{aligned} \mathrm{OS}(\overline{U}) = \{\, & \{\, x \mapsto u_{Nat} + v_{NzNat}, y \mapsto u_{Nat}, z \mapsto v_{NzNat} \,\}, \\ & \{\, x \mapsto u_{NzNat} + v_{Nat}, y \mapsto u_{NzNat}, z \mapsto v_{Nat} \,\} \,\}. \end{aligned}$$

In Section 4.3, we prove the following result to show that our algorithm is correct.

Theorem 4.3.9. *Let $\mathcal{E}$ denote a preregular and sort-independent order-sorted Σ-theory.*

Given a unification problem Γ with a complete set of most-general sort-preserving unsorted unifiers $\overline{U}$, $\mathrm{OS}(\overline{U})$ is a complete set of most-general order-sorted unifiers for Γ.

4.2 Order-sorted AC + ACU unification

As the requirements on $\mathcal{E}$ and $\overline{U}$ seem rather technical, to give the reader a more intuitive feel for them, we show how the requirements are satisfied by many order-sorted equational theories specified as Maude modules having free, commutative, AC, and ACU symbols. Essentially, each such Maude module can be viewed as an order-sorted theory $\mathcal{E}$ over a signature $\Sigma = (S, F, \leq)$ such that:

(a) Each equivalence class $[s] \in S/\equiv_{\leq}$ contains a maximal element k_s called the *kind* of s where $\equiv_{\leq}$ denotes the equivalence relation generated by $\leq$. Moreover, for each operator declaration $f \in F_{s_1 \ldots s_n, s}$, there is also a declaration $f \in F_{k_{s_1} \ldots k_{s_n}, k_s}$.

(b) E contains axioms of the following forms:

$$\underset{\text{associativity}}{f(f(x,y),z) = f(x,f(y,z))} \quad \underset{\text{commutativity}}{f(x,y) = f(y,x)} \quad \underset{\text{unit}}{f(c,x) = x}$$

where the sorts of x, y, z are maximal sorts, that is, sorts of the form k_s for some $s \in S$, and for each binary symbol $f \in F$, either f does not appear in E, or E contains commutativity (C), associativity and commutativity (AC), or associativity, commutativity, and unit (ACU) axioms for f.

(c) Σ is preregular.

(d) Each axiom $l = r \in E$ is *sort-preserving*, that is for each variable mapping $\rho : X \to X$, $\mathrm{ls}(l\rho) = \mathrm{ls}(r\rho)$.

The assumptions (a)–(d) are quite reasonable for order-sorted Maude specifications with free, commutative, AC, and ACU operators. Maude will automatically introduce additional top-most sorts k_s, and requires that associativity, commutativity, and unit axioms satisfy the requirements in (b). Maude does allow associative symbols that are not commutative, however unification for such theories may be infinitary [130] and is not considered here. The preregularity requirement is checked automatically by Maude when the module is entered. The sort-preservation requirement (d) is essential as the sort-propagation algorithm described in the previous section operates syntactically on terms, and disregards the possibility that applying an equation may change the sort of a term. It is guaranteed by a three-pronged approach:

- For each associativity axiom $f(f(x,y),z) = f(x,f(y,z))$, Maude checks that it is sort-preserving by considering possible variable mappings.
- For each commutativity axiom $f(x,y) = f(y,x)$ and each operator declaration $f : s_1\ s_2 \to s$, Maude completes the theory by adding the declaration $f : s_2\ s_1 \to s$.
- For each pair of identity axioms $f(x,c) = x$ and $f(c,x) = x$, our unification procedure completes the theory by introducing a fresh sort s_c together with: (1) an operator declaration $c : \to s_c$, (2) a subsort declaration $s_c < \mathrm{ls}(c)$, and (3) for each sort $s \equiv_{\leq} \mathrm{ls}(c)$, operator declarations $f : s\ s_c \to s$ and $f : s_c\ s \to s$.

We now focus on the relationship between the assumptions (a)–(d) and the earlier requirements (i)–(iii). The first preregularity requirement follows from the preregularity assumption. The sort-independence requirements follows form the assumptions (a) and (b).

Theorem 4.2.1. *If $\mathcal{E}$ is an order-sorted Σ-theory satisfying assumptions (a) and (b) above, then $\mathcal{E}$ is sort-independent.*

Proof. Showing that $\mathcal{E}$ is sort-independent requires showing that for all well-sorted terms $t, u \in T_\Sigma(X)$, $t =_{\overline{\mathcal{E}}} u$ implies $t =_\mathcal{E} u$.

We first partition E into disjoint sets $E = R \uplus A_x$ where A_x contains the associativity and commutativity equations in E and the identity equations $f(c, x) = x$ in E are interpreted as rules $f(c, x) \to x$ in R. It is not difficult to see that the rules R modulo A_x are terminating and confluent, and therefore $t =_{\overline{\mathcal{E}}} u$ iff $t \downarrow_{\overline{R}/\overline{A}_x} =_{\overline{A}_x} u \downarrow_{\overline{R}/\overline{A}_x}$.

As A_x only contains associativity and commutativity axioms, if $t \in T_\Sigma(X)_{k_s}$ for some maximal sort k_s and $t =_{\overline{A}_x} v$, then it easily follows that $v \in T_\Sigma(X)_{k_s}$ and $t =_{A_x} v$ by the requirement (a). It also easily follows that if $t \to^*_{\overline{R}/\overline{A}_x} v$, then $v \in T_\Sigma(X)_{k_s}$ and $t \to^*_{R/A_x} v$. From this, we can conclude that

$$t \downarrow_{\overline{R}/\overline{A}_x} =_{\overline{A}_x} u \downarrow_{\overline{R}/\overline{A}_x} \implies t \downarrow_{R/A_x} =_{A_x} u \downarrow_{R/A_x} .$$

It easily follows that $t =_\mathcal{E} u$, and thus $\mathcal{E}$ is sort-independent. □

In general, the requirement that the unification procedure is sort-preserving *does not* follow from the assumptions given above. For an example, consider the theory $\mathcal{E}$ with two unrelated top-most sorts *Nat* and *Cns* where *Nat* contains the ACU symbol + with the identity element 0, and *Cns* contains the constant a. Given the unification problem $x_{Cns} = a$, it would be permissible for the unsorted unification procedure to return the unifiers

$$\overline{U} = \{ \{ x \mapsto a + 0 \} \}.$$

This is a complete set of unsorted unifiers due to the identity axiom, but unsuitable for our sort propagation algorithm as $a + 0$ is not a legal term. This counterexample illustrates why the earlier work [112] imposed significant restrictions on theories with collapsing equations like identity.

These stronger restrictions appear unnecessary in practice — in our experience, the procedure will not introduce extra symbols, and in this case return the simpler unifier $x \mapsto a$. The reason that unsorted AC and ACU unification procedures satisfy this assumption is that the unifiers are computed from the terms appearing in equations $l = r \in \Gamma$. When those subterms are well-typed with the same top-most sort k, substitutions generated by the unsorted unification procedure should be well-typed as well. Provided that the sorts of fresh variables in the right hand side of a variable are given the appropriate top sort k, due to our assumption (a), we have found it is safe to assume the following:

(e) For each unifier $\overline{\theta}$ in the set of unifiers $\overline{U}$ returned by the unsorted unification procedure for the order-sorted unification problem Γ, and for each variable $x_s \in \text{vars}(\Gamma)$, $x_s\overline{\theta} \in T_\Sigma(X)_{k_s}$.

To validate these ideas and test this assumption, we have extended an alpha version of Maude so that it may communicate with CiME [35, 36] by passing un-

sorted unification problems as strings, and parsing the unsorted unifiers returned from CiME back into Maude terms. As an additional safeguard, the parsing process checks the substitutions returned by CiME to verify that assumption (e) is satisfied. These checks have always been satisfied in our experience using the procedure so far. We then apply the sort propagation algorithm described in the previous section to generate order-sorted $\mathcal{E}$-unifiers. The order-sorted unification procedure is used to analyze cryptographic protocols with algebraic properties of associativity and commutativity using the Maude-NRL protocol analyzer [49].

4.3 Correctness

The goal of this section is to show the correctness of our approach to order-sorted equational unification. Before we can show this, we need several intermediate lemmas. The first lemma shows how preregularity is used.

Lemma 4.3.1. *If $\Sigma = (S, F, \leq)$ is preregular, then for all sorts $s_1, s_2 \in S$ and terms $t \in T_{\overline{\Sigma}}(\overline{X})$,*

$$t \in T_\Sigma(X)_{s_1} \cap T_\Sigma(X)_{s_2} \iff (\exists s \in \mathrm{glb}_\Sigma(s_1, s_2))\, t \in T_\Sigma(X)_s$$

where $\mathrm{glb}_\Sigma(s_1, s_2) = \sup_{\leq}(\{\, s \in S \mid s \leq s_1 \wedge s \leq s_2 \,\})$.

Proof. If there is a sort $s \in \mathrm{glb}(s_1, s_2)$ such that $t \in T_\Sigma(X)_s$, then $t \in T_\Sigma(X)_{s_1} \cap T_\Sigma(X)_{s_2}$ as $s \leq s_1$ and $s \leq s_2$. We still must show that $t \in T_\Sigma(X)_{s_1} \cap T_\Sigma(X)_{s_2}$ implies that there is a sort $s \in \mathrm{glb}(s_1, s_2)$ such that $t \in T_\Sigma(X)_s$. However, this follows immediately as t must have a least sort $s' \in S$. It must be the case that $s' \leq s_1$ and $s' \leq s_2$. Therefore, there is an $s \in \mathrm{glb}(s_1, s_2)$ such that $s' \leq s$. As $T_\Sigma(X)_{s'} \subseteq T_\Sigma(X)_s$, it follows that $t \in T_\Sigma(X)_s$. □

Lemma 4.3.2. *For all terms $f(t_1, \ldots, t_n) \in T_{\overline{\Sigma}}(\mathrm{X})$ and sorts $s \in S$,*

$$f(t_1, \ldots, t_n) \in T_\Sigma(X)_s \iff (\exists s_1 \ldots s_n \in \mathrm{ar}_\Sigma(f, s, n))\, t_1 \in T_\Sigma(X)_{s_1} \wedge \cdots \wedge t_n \in T_\Sigma(X)_{s_n}$$

where $\mathrm{ar}_\Sigma(f, s, n) = \sup_{\leq^n}(\{\, w \in S^n \mid (\exists s' \in S)\, f \in F_{w,s'} \wedge s' \leq s \,\})$.

Proof. If there are sorts $s_1 \ldots s_n \in \mathrm{ar}_\Sigma(f, s, n)$ such that $t_i \in T_\Sigma(X)_{s_i}$ for $i \in [1, n]$, then there must be a sort $s' \leq s$ such that $f \in F_{s_1 \ldots s_n, s'}$. It follows that $f(t_1, \ldots, t_n) \in T_\Sigma(X)_{s'}$, and thus $f(t_1, \ldots, t_n) \in T_\Sigma(X)_s$.

On the other hand, if $f(t_1, \ldots, t_n) \in T_\Sigma(X)_s$, then there is some $s' \leq s$ such that $f \in F_{s'_1 \ldots s'_n, s'}$ and $t_i \in T_\Sigma(X)_{s'_i}$ for $i \in [1, n]$. It follows that there are sorts $s_1 \ldots s_n \in \mathrm{ar}_\Sigma(f, s, n)$ such that $s'_i \leq s_i$ for $i \in [1, n]$. Consequently, $t_i \in T_\Sigma(X)_{s_i}$ for $i \in [1, n]$. □

For a membership constraint M, we define the *unifiers* for M, denoted $\mathrm{Un}_{\overline{\Sigma}}(M)$ to be the set of unsorted substitutions $\theta : \overline{X} \to T_{\overline{\Sigma}}(\overline{X})$ such that for each membership $t : s \in M$, $t\theta \in T_\Sigma(X)_s$.

Lemma 4.3.3. *For each order-sorted signature $\Sigma = (S, F, \leq)$ and pair of membership constraints M_1 and M_2,*

$$M_1 \geq M_2 \implies \mathrm{Un}_\Sigma(M_1) \supseteq \mathrm{Un}_\Sigma(M_2).$$

Proof. To show that $\mathrm{Un}_\Sigma(M_1) \supseteq \mathrm{Un}_\Sigma(M_2)$, we must show for each substitution $\theta \in \mathrm{Un}_\Sigma(M_2)$ and membership $t : s \in M_1$, we have $t\theta \in T_\Sigma(X)_s$. However, since $M_1 \geq M_2$, for each $t : s \in M_1$, there is a membership $t : s' \in M_2$ such that $s' \leq s$. By definition $t\theta \in T_\Sigma(X)_{s'}$, and therefore $t\theta \in T_\Sigma(X)_s$. □

When the membership constraints M_1 and M_2 are reduced, the previous implication holds in the other direction.

Lemma 4.3.4. *For each order-sorted signature $\Sigma = (S, F, \leq)$ and pair of reduced membership constraints M_1 and M_2 such that* $\mathrm{vars}(M_1) = \mathrm{vars}(M_2)$,

$$\mathrm{Un}_\Sigma(M_1) \supseteq \mathrm{Un}_\Sigma(M_2) \implies M_1 \geq M_2$$

Proof. Since both M_1 and M_2 are reduced and $\mathrm{vars}(M_1) = \mathrm{vars}(M_2)$, to show that $M_1 \geq M_2$, it is sufficient to show that for each $x \in \mathrm{vars}(M_1)$, $\mathrm{sort}_{M_1}(x) \geq \mathrm{sort}_{M_2}(x)$. Since M_2 is reduced, there is a substitution $\rho_{M_2} : \mathrm{vars}(M_2) \to T_\Sigma(X)$ which maps each variable $x \in \mathrm{vars}(M_2)$ to the a fresh variable x' with sort $\mathrm{sort}_{M_2}(x) \in S$. Clearly $\rho_{M_2} \in \mathrm{Un}_\Sigma(M_2)$, and so $\rho_{M_2} \in \mathrm{Un}_\Sigma(M_1)$ by assumption. since $x\rho_{M_2}$ is a variable with sort $\mathrm{sort}_{M_2}(x)$ and $x\rho_{M_2} \in \mathrm{Un}_\Sigma(M_2)$, it follows that $\mathrm{sort}_{M_1}(x) \geq \mathrm{sort}_{M_2}(x)$ for each $x \in \mathrm{vars}(M_2)$. □

For a disjunctive set of membership constraints D, we let $\mathrm{Un}_{\overline{\Sigma}}(D)$ denote the set of unsorted substitutions that are unifiers for a set of membership constraints $M \in D$, i.e.,

$$\mathrm{Un}_{\overline{\Sigma}}(D) = \bigcup_{M \in D} \mathrm{Un}_{\overline{\Sigma}}(M).$$

The key correctness property of the inference rules in Figure 4.1 is captured by the following lemma.

Lemma 4.3.5. *For a preregular order-sorted signature Σ, if $D_1 \to^* D_2$ using the inference rules in Figure 4.1, then* $\mathrm{Un}_\Sigma(D_1) = \mathrm{Un}_\Sigma(D_2)$.

Proof. To show this it is enough to show the single step case that $D_1 \to D_2$ implies $\mathrm{Un}_\Sigma(D_1) = \mathrm{Un}_\Sigma(D_2)$. The full lemma follows easily by induction on the number of rules used to show $D_1 \to^* D_2$. To show the single step case, we must consider three separate cases, one for each of the inference rules in Figure 4.1:

1. If **Intersection** is used, then this lemma follows from Lemma 4.3.1.

2. If **Propagation** is used, then this follows from Lemma 4.3.2.

3. If **Subsumption** is used, then this follows from 4.3.3.

□

In order to preserve the set of substitutions, we also need to show that the inference rules do not discard variables or introduce new ones:

Lemma 4.3.6. *If $D_1 \rightarrow^* D_2$ using the inference rules in Figure 4.1 and each set of membership constraints $M_1 \in D_1$ has the same variables* $\text{vars}(M_1) = X$*, then for all $M_2 \in D_2$,* $\text{vars}(M_2) = X$*.*

Proof. This is a straightforward induction over the number of rewrites used to show $D_1 \rightarrow^* D_2$ and considering each rule separately. □

The following lemma is useful to show that the inference rules terminate with a unique set of membership constraints.

Lemma 4.3.7. *If D_1 and D_2 are both disjunctive sets of membership constraints that are irreducible by the inference rules in Figure 4.1, then* $\text{Un}_\Sigma(D_1) = \text{Un}_\Sigma(D_2)$ *implies $D_1 = D_2$.*

Proof. We show this by showing that $D_1 \neq D_2$ implies $\text{Un}(D_1) \neq \text{Un}(D_2)$. If $D_1 \neq D_2$, then there must be a conjunction of membership constraints M_1 which is in $D_1 \setminus D_2$ or $D_2 \setminus D_1$. We assume that the M_1 is in D_1 as the other case is symmetric. Since the rules in Figure 4.1 cannot be applied to D_1, the membership constraints M_1 must be reduced, and hence has the form $M_1 = x_1 : s_1 \wedge \cdots \wedge x_n : s_n$ with $x_i \neq x_j$ for each $i \neq j$. Let ρ_{M_1} denote the substitution mapping each variable $x \in \text{vars}(M_1)$ to a fresh variable $x\rho_{M_1}$ with sort $\text{sort}_{M_1}(x) \in S$. By definition $\rho_{M_1} \in \text{Un}_\Sigma(M_1)$ and therefore $\rho_{M_1} \in \text{Un}(D_1)$. If $\rho_{M_1} \notin \text{Un}(D_2)$, then $\text{Un}(D_1) \neq \text{Un}(D_2)$, and consequently we are done. Otherwise, ρ_{M_1} is in $\text{Un}(D_2)$, and so there must be a membership $M_2 \in \text{Un}(D_2)$ such that for each membership $x : s \in M_2$ there is a membership $x : s' \in M_1$ with $s' \leq s$. It follows that $M_2 \geq M_1$. Since D_1 is fully reduced by the rules in Figure 4.1, it follows that the substitution ρ_{M_2} is not in $\text{Un}_\Sigma(D_1)$, since this would imply that there is a mapping $M \geq M_2 \geq M_1$ in D_1. This is impossible since D_1 has been fully reduced by the rules in Figure 4.1. □

Using the previous lemmas, it is not difficult to show the following Termination Theorem which shows that the inference rules terminate with a unique set of membership constraints.

Theorem 4.3.8 (Termination Theorem)**.** *For each disjunctive set of membership constraints D, there is a unique set of membership constraints D^* such that $D \rightarrow^! D^*$ using the inference rules in Figure 4.1.*

Proof. Showing this requires proving that: (1) the rules in Figure 4.1 are terminating and (2) if $D \to^! D_1$ and $D \to^! D_2$, then $D_1 = D_2$. The rules in Figure 4.1 are terminating, because each rewrite either reduces the size of a term in a membership, or preserves the terms while reducing the total number of memberships. To show (2), observe that if $D \to^! D_1$ and $D \to^! D_2$, then $\mathrm{Un}_\Sigma(D_1) = \mathrm{Un}_\Sigma(D) = \mathrm{Un}_\Sigma(D_2)$ by Lemma 4.3.5. Therefore, $D_1 = D_2$ by Lemma 4.3.7. □

We now conclude with a proof of the main theorem of this chapter.

Theorem 4.3.9. *Let $\mathcal{E}$ denote a preregular and sort-independent order-sorted Σ-theory.*

Given a unification problem Γ with a complete set of most-general sort-preserving unsorted unifiers $\overline{U}$, $\mathrm{OS}(\overline{U})$ is a complete set of most-general order-sorted unifiers for Γ.

Proof. Proving the above theorem requires showing three things: (1) each element of $\mathrm{OS}(\overline{U})$ is an order-sorted unifier for Γ; (2) the set of unifiers $\mathrm{OS}(\overline{U})$ is complete; (3) the set of unifiers $\mathrm{OS}(\overline{U})$ is most-general. We show each of these facts separately.

- For each element $\theta \in \mathrm{OS}(\overline{U})$, there is an unsorted unifier $\overline{\theta} \in \overline{U}$ and reduced membership constraints $M \in D(\overline{\theta})^*$ such that $\theta = \overline{\theta}\rho_M$. We first show that $x_s\theta \in T_\Sigma(X)_s$ for each variable $x_s \in \mathrm{vars}(\Gamma)$. To see this, observe that by definition $\rho_M \in \mathrm{Un}_\Sigma(D(\overline{\theta})^*)$, and so by Lemma 4.3.5, $\rho_M \in \mathrm{Un}_\Sigma(D(\overline{\theta}))$. Furthermore, by Lemma 4.3.6, we know that $\mathrm{vars}(M) = \mathrm{vars}(D(\overline{\theta})) = \mathrm{vars}(\Gamma)$. It follows by definition that for each variable $x_s \in \mathrm{vars}(\Gamma)$, $x\theta = x\overline{\theta}\rho_M$ is in $T_\Sigma(X)_s$. For each equation $t = u$ in Γ both t and u are well-sorted terms belonging to the same component. It follows that $t\theta$ and $u\theta$ are well-sorted terms as well. By definition $t\theta =_{\overline{\mathcal{E}}} u\theta$, and as $\mathcal{E}$ is sort-independent it follows that $t\theta =_{\mathcal{E}} u\theta$.

- To show that $\mathrm{OS}(\overline{U})$ is complete, we must show for each order-sorted unifier $\psi \in \mathrm{Un}_{\mathcal{E}}(\Gamma)$, there is a unifier $\theta \in \mathrm{OS}(\overline{U})$ and order-sorted substitution $\phi : \mathrm{rvars}(\theta) \to T_\Sigma(X)$ such that $x\psi =_{\mathcal{E}} x\theta\phi$ for each $x \in \mathrm{vars}(\Gamma)$. Let ψ be a unifier in $\mathrm{Un}_{\mathcal{E}}(\Gamma)$. As $\overline{U}$ is a complete set of sort-preserving unifiers, there is an unifier $\overline{\theta} \in \overline{U}$ such that $\psi =_{\overline{\mathcal{E}}} \overline{\theta\phi}$ for some unsorted substitution $\overline{\phi} : Y \to T_\Sigma(X)$ with $Y = \mathrm{rvars}(\overline{\theta})$. Moreover, since $x_s\psi \in T_\Sigma(X)_s$ for each variable $x_s \in \mathrm{vars}(\Gamma)$, we can assume that $x_s\overline{\theta\phi} \in T_\Sigma(X)_s$ since $\overline{U}$ is sort-preserving.

  It follows that $\overline{\phi} \in \mathrm{Un}_\Sigma(D(\overline{\theta}))$. By Lemma 4.3.5 and Theorem 4.3.8, there must be a reduced set of membership constraints $M \in D(\overline{\theta})^*$ such that

$$(\forall x \in Y)\; x\overline{\phi} \in T_\Sigma(X)_{\mathrm{sort}_M(x)}. \tag{4.1}$$

Since M is reduced, there is a variable renaming ρ_M with maps each variable $x \in Y$ to a fresh variable x' with sort $\mathrm{sort}_M(x)$. Let ρ_M^{-1} denote the inverse of that renaming. By using (4.1), it should be clear that $\rho_M^{-1}; \overline{\phi}$ is an order-sorted substitution. Moreover, as $\rho_M \in \mathrm{Un}_\Sigma(M)$ and therefore in $\mathrm{Un}_\Sigma(D(\overline{\theta}))$ by Lemma 4.3.5, $\overline{\theta}; \rho_M$ must be an order-sorted substitution. Since $\overline{\theta}; \rho_M \in \mathrm{OS}(\overline{U})$ and $\psi = (\overline{\theta}; \rho_M); (\rho_M^{-1}; \overline{\phi})$, it follows that $\mathrm{OS}(\overline{U})$ is a complete set of unifiers.

- To show that $\mathrm{OS}(\overline{U})$ is a most-general set of unifiers, we must show for all distinct substitutions $\theta_1, \theta_2 \in \mathrm{OS}(\overline{U})$, we have $\theta_1 \not\geq \theta_2$. We prove this by contradiction. Assume that there are substitutions $\theta_1, \theta_2 \in \mathrm{OS}(\overline{U})$ and a substitution $\psi : Y \to T_\Sigma(X)$ such that $\theta_1 = \theta_2; \psi$, where Y denotes the variables in the right-hand side of θ_2. Since both θ_1 and θ_2 are in $\mathrm{OS}(\overline{U})$, they must have the form $\theta_1 = \overline{\theta_1}; \rho_{M_1}$ and $\theta_2 = \overline{\theta_2}; \rho_{M_2}$ with $\overline{\theta_1}, \overline{\theta_2} \in \overline{U}$, $M_1 \in D(\overline{\theta_1})^*$, and $M_2 \in D(\overline{\theta_2})^*$. Our assumption $\theta_1 = \theta_2; \psi$ implies that $\overline{\theta_1} = \overline{\theta_2}; (\rho_{M_2}; \psi; \rho_{M_1}^{-1})$. Since $\overline{U}$ is most-general, this can only be the case if $\overline{\theta_1} = \overline{\theta_2}$. As we assumed that $\theta_1 = \theta_2; \psi$, it is not difficult to show that $\psi = \rho_{M_2}^{-1} \rho_{M_1}$. For each variable $x \in Y$, the sort of the variable $x\rho_{M_2}^{-1}\rho_{M_1}$ is $\mathrm{sort}_{M_1}(x\rho_{M_2}^{-1})$, and $\mathrm{sort}_{M_1}(x\rho_{M_2}^{-1}) \leq \mathrm{sort}_{M_2}(x\rho_{M_2}^{-1})$ since ψ is an order-sorted substitution. It follows that $M_2 \geq M_1$ which is impossible since both $M_1, M_2 \in D(\overline{\theta_1})^*$ and $D(\overline{\theta_1})^*$ have been fully normalized using the inference rules in Figure 4.1.

□

4.4 Related work and conclusions

There is a considerable amount of research already in unification in theories with sorts and subsorts (e.g., [16, 54, 93, 112, 141, 142]) due to the improved expressiveness of order-sorted algebras and ability to simplify automated reasoning. The use of rule-based algorithms in describing unification has a long history as well with the most well-known example being Martelli and Montanari's algorithm for syntactic unification [107]. Our use of a rule-based approach to order-sorted unification is not particularly novel; however the relaxed conditions of our soundness proof are necessary to ensure the correctness of the algorithm when used by the Maude-NRL protocol analyzer.

Due to our experience with the order-sorted unification in the Maude-NRL analyzer, an order-sorted unification engine is planned to be included in the next Maude release. This unification engine will make similar assumptions to our own ones about the supported theories, however it should have better performance as it will no longer need to parse unsorted unifiers back as strings, and can more tightly integrate the order-sorted constraints into the core unification routines. In fact, a prototype BDD-based approach to solving the sort constraints is currently being developed by Steven Eker. This BDD-based approach has the

advantage that the subsumption checks can be handled automatically by the BDD generation-algorithms.

Our aim in this chapter is more general than the Maude-based applications of our algorithm. Our aim is one of *modularity*, so that different formal tool building efforts needing equational order-sorted unification procedures may be able to modularly decompose such a procedure into its *unsorted part* where several existing tools may be used and the rule-based *sort propagation algorithm* that we have presented and proved correct. Furthermore, our algorithm may be useful for Maude applications even after the built-in implementation of order-sorted unification for theories with C and AC symbols. This is because our algorithm can perform order-sorted unification for theories other than C or AC provided an unsorted unification procedure is available.

Chapter 5

Sufficient completeness

The symbols in a signature are often used in two different ways: they can be viewed as *constructors* that create data, or as *defined operations* that compute functions of interest. One property that is easy to overlook, but nevertheless crucial for correctness, is the sufficient completeness property. A specification is *sufficiently complete* if each function returns a well-defined value when called on all relevant inputs.

Sufficient completeness is an important property to check, both to debug and formally reason about specifications and equational programs. For example, many inductive theorem proving techniques are based on the constructors building up the data and crucially depend on the specification being sufficiently complete. Failure to be sufficiently complete is a common error in specifications. The problem is that each operation typically requires several equations — each matching terms with a different topmost constructor. It is easy to forget one of the equations when defining a function, or to forget to add additional equations for all defined functions when adding a new constructor symbol.

Sufficient completeness was first defined in Guttag's thesis [66] (but see [67] for a more accessible treatment). An unsorted theory $\mathcal{E}$ over a signature Σ is said to be *sufficiently complete* relative to a constructor subsignature $\Omega \subseteq \Sigma$ iff for all ground terms $t \in T_\Sigma$, there is a ground constructor term $u \in T_\Omega$ such that $t =_\mathcal{E} u$. For a given choice of constructors Ω, the terms in T_Ω are said to be *constructor terms*.

Sufficient completeness is in general undecidable, even for unconditional specifications [66, 67]. Approaches to check sufficient completeness are obtained by treating the equations as oriented rules, and casting sufficient completeness as a *ground reducibility* problem. An early algorithm for handling unconditional linear specifications is due to Nipkow and Weikum [120]. For a good review of the literature up to the late 80s, as well as some important decidability/undecidability and complexity results, see [89, 90]. A more recent development is the casting of sufficient completeness as a tree automata (TA) decision problem: see Chapter 4 of [33] and references there. In particular, tree automata techniques were used to show that for the unsorted and unconditional case, sufficient completeness for a terminating and confluent specification is EXPTIME-complete [32]. Tools for checking sufficient completeness include

Spike [11], the RRL [88] theorem prover, and two new tools which we introduce in this chapter.

In practice, there is a need to have expressive equational specification formalisms that match the needs of real applications and a corresponding need to extend sufficient completeness methods to handle such formalisms. This chapter presents new contributions extending sufficient completeness methods in several useful directions, namely: (1) to handle partiality; (2) to handle operator overloading; (3) to allow conditional specifications; and (4) to support equational reasoning *modulo* axioms. These extensions are needed in practice because: (1) functions defined on data types are often *partial*; (2) overloading is an essential feature of sorted logics like order-sorted and membership equational logic; (3) many languages support conditional specifications; and (4) functions often assume algebraic properties of their underlying data. For example, functions on sets or multisets are much more simply specified using the fact that set and multiset union are associative and commutative.

Of course, there is tension between expressiveness of specifications and decidability of sufficient completeness. Furthermore, when the specification is given in a logic with operator overloading and partiality such as MEL, the definition of sufficient completeness must be generalized to take two additional factors into account: (1) partial functions may intentionally not be defined for all inputs, but only on a particular subset of inputs; (2) the same symbol may be a constructor on a small domain, but is a defined operation on another domain. For example, *successor* is a constructor when applied to natural numbers, and a defined function on negative integers.

Sufficient completeness of MEL specifications was first studied in [19]. The definition and methods presented in this chapter substantially extend and generalize those in [73], which in turn had generalized those in [19], allowing a much wider class of MEL specifications to be checked. For checking sufficient completeness, our work can be seen as a generalization of two different approaches to the membership equation logic case: techniques based on the incremental constructor-based *narrowing* of patterns; (2) techniques that cast sufficient completeness as decision problem for tree automata.

In our view, the best way to deal with this tension is *not to give up* on the expressive specifications that are often needed in practice and for which sufficient completeness is undecidable. We advocate a two-pronged approach. First, the sufficient completeness problem should be studied for increasingly more expressive formalisms, and the set of decidable specifications should likewise be expanded as much as possible. Second, sufficient completeness checking algorithms should be coupled with inductive theorem proving techniques that can discharge proof obligations generated when the input specification falls outside of the decidable classes. We refer the reader to [73, 83] for ideas on coupling sufficient completeness and inductive theorem proving.

In this chapter, we focus on advancing the first prong in several ways. Our

first contribution is to characterize sufficient completeness in a more general setting to support the extensions mentioned above. For this purpose, we use membership equational logic (MEL) to allow conditional specification of partial functions (see [111] for a survey of partial specification formalisms and the use of MEL in this context). In MEL atomic sentences are either equations $t = t'$, or memberships $t : s$ stating that a term t has a sort s, where t having a sort is equivalent to t being *defined*.

The key idea is that a partial function's domain is axiomatized by conditional membership axioms. We precisely define the sufficient completeness property for conditional MEL specifications which can have extra variables in their conditions and can be defined modulo a set A of unconditional equations. We define sufficient completeness for both MEL theories of this kind and their corresponding conditional rewrite theories when the equations are used as rewrite rules. We also characterize specifications for which both notions coincide. Finally for a large class of specifications, we give conditions equivalent to sufficient completeness which can be the basis of a checking algorithm. These theoretical developments directly apply to the analysis of functional modules in the Maude language [28], which are MEL specifications supporting deduction modulo axioms such as associativity, commutativity, and identity.

Our second contribution is to present two new tools for checking sufficient completeness: (1) a constructor-narrowing based tool for conditional MEL specifications which operates by generating proof obligations for showing sufficient completeness which may be discharged by an inductive theorem prover, and (2) a tree automata based checker which casts sufficient completeness for left-linear, unconditional, order-sorted specifications with rewriting modulo axioms as a propositional emptiness problem for equational tree automata. Although sufficient completeness as a tree automata decision problem has been studied before in the unsorted free case, this has not been done before in studying sufficient completeness with rewriting modulo axioms. This decision problem may still not be decidable. However, we can use the decidability and semi-decidability results presented in Chapter 3 to attack the problem. In particular, the semi-decision procedure in Section 3.3 forms the basis of a new sufficient completeness checker [75] for order-sorted Maude specifications.

This chapter is organized as follows. We begin in Section 5.1 with our definition of sufficient completeness in MEL, and discuss its relationship with sufficient completeness for simpler logics. In Section 5.2, we generalize the relationship between sufficient completeness and ground reducibility to membership equational logic. This relationship is the basis for all approaches to checking sufficient completeness that we are aware of, and its generalization to membership equational logic turns out to be rather subtle. In Section 5.3, we present our narrowing-based sufficient completeness checker, and in Section 5.4, we present our equational tree automata-based sufficient completeness checker. Finally, in Section 5.5, we conclude with a summary of our results and suggest directions

for future research.

5.1 Defining sufficient completeness in MEL

In MEL, the signature declares operator symbols, but it is left to the memberships to indicate when a term is "well-defined." By partitioning the memberships into constructor and defined sets, we can obtain a definition of sufficient completeness that is more faithful to MEL's notion of what it means for a term to be well-formed in a specification.

Definition 5.1.1. *Let $\mathcal{E} = E \cup M$ be a MEL specification where E contains the (conditional) equations and M contains the (conditional) memberships.*

Given a subset of memberships $M_\Omega \subseteq M$, called constructor memberships*, we define the* constructor subspecification *to be $\mathcal{E}_\Omega = E \cup M_\Omega$. Furthermore, we say that $\mathcal{E}$ is* sufficiently complete *relative to M_Ω iff $T_\mathcal{E}$ and $T_{\mathcal{E}_\Omega}$ are isomorphic.*

Since the only difference between $\mathcal{E}$ and $\mathcal{E}_\Omega$ involves the memberships $M_\Delta = \mathcal{E} - \mathcal{E}_\Omega$, it is not difficult to show that the above definition of sufficient completeness is equivalent to the property that for all ground terms $t \in T_\Sigma$ and sorts $s \in S$,

$$\mathcal{E} \vdash t : s \iff \mathcal{E}_\Omega \vdash t : s.$$

As shown below, this definition of sufficient completeness for MEL specifications easily generalizes the definition for unsorted specifications.

Theorem 5.1.2. *Given an unsorted theory $\overline{\mathcal{E}}$ over a signature $\overline{\Sigma}$ that is equipped with constructor subsignature $\overline{\Omega} \subseteq \overline{\Sigma}$, there is a MEL theory $\mathcal{E} = E \cup M_\Omega \cup M_\Delta$ over a MEL signature Σ such that $\overline{\mathcal{E}}$ is sufficiently-complete relative to $\overline{\Omega}$ iff $\mathcal{E}$ is sufficiently-complete relative to M_Ω.*

Proof. Let Σ be the specification containing a single kind k, single sort s, and for each $f \in \overline{\Sigma}$, a function symbol f with n inputs each with kind k and output kind k. Additionally, let

$$M_\Omega = \{\, f(x_1, \ldots, x_n) : s \textbf{ if } x_1 : s \wedge \ldots x_n : s \mid f \in \Omega_n \,\}$$
$$M_\Delta = \{\, f(x_1, \ldots, x_n) : s \textbf{ if } x_1 : s \wedge \ldots x_n : s \mid f \in \Sigma_n - \Omega_n \,\}$$

where Σ_n denotes the symbols with arity n in Σ.

It is easy to see that for $\mathcal{E}$ and $\mathcal{E}_\Omega = E \cup M_\Omega$, and all $t \in T_\Sigma$, we have:

$$\mathcal{E} \vdash t : s \iff (\exists u \in T_\Sigma)\ t =_{\overline{\mathcal{E}}} u \quad \text{and} \quad \mathcal{E}_\Omega \vdash t : s \iff (\exists u \in T_\Omega)\ t =_{\overline{\mathcal{E}}} u$$

Thus $\mathcal{E}$ is sufficiently complete relative to Ω iff $\mathcal{E}$ is sufficiently complete relative to M_Ω. □

To illustrate how the new definition of sufficient completeness is useful, we refer to the definition in Maude of unnested powerlists [113] which appeared

earlier in Figure 2.5. We reprint the definition here and some of the discussion to make this chapter more self-contained. Recall that unnested powerlists are non-empty lists of length 2^n for some $n \in \mathbb{N}$. Powerlists with the same length may be concatenated together with an operator |, called the *tie* operator. Additionally, powerlists with the same length may be interleaved together with an operator $\times$, called the *zip* operator. The concatenation or interleaving of powerlists with different lengths is not defined. For example, both $(1 \mid 2) \mid (3 \mid 4)$ and $(1 \mid 2) \times (3 \mid 4) = (1 \mid 3) \mid (2 \mid 4)$ are well-formed powerlists, but $(4 \mid 3) \times (1)$ and $(1 \mid 2) \mid (3)$ are not.

```
fmod POWERLIST is protecting NAT .
  sort Pow .
  op [_] : Nat -> Pow [ctor] .
  op _|_ : [Pow] [Pow] -> [Pow] .

  vars M N : Nat .
  vars P Q R S : Pow .

  op len : Pow -> Nat .
  eq len(P tie Q) = len(P) + len(Q) .
  eq len([N]) = 1 .

  cmb (P | Q) : Pow if len(P) = len(Q) .

  op _x_ : [Pow] [Pow] -> [Pow] .
  cmb (P x Q) : Pow if len(P) = len(Q) [metadata "dfn"].
  eq (P | Q) x (R | S) = (P x R) | (Q x S) .
  eq [M] x [N] = [M] | [N] .
endfm
```

The module `POWERLIST` imports the predefined module `NAT`, which declares the natural numbers and the expected arithmetic operations and relations. The sort `Pow` is introduced, which we use to represent well-formed powerlists; Maude automatically introduces also the kind `[Pow]` to denote the kind of the sort `Pow`. We also introduce four operators: `[_]` for representing the operation that forms powerlist elements; `_|_` for representing the powerlist *tie* operation; `_x_` for representing the powerlist *zip* operation; and `len` for representing the operation that computes the length of a powerlist. Since not all terms constructed with the operators `_|_` and `_x_` represent powerlists, those operators are declared at the kind level. For example, $([2] \mid [3]) \times [4]$ is not a powerlist. This is represented in `POWERLIST` by the fact that the term `([2] | [3]) x [4]` has kind `[Pow]`, but it does not belong to the sort `Pow`. On the other hand, since we want to use the `[_]` operator to *construct* powerlists (specifically, powerlists with only one element), we declare this operator at the sort level and with the `ctor` attribute. Finally, since we expect that the `len` operator applied to a powerlist will always evaluate to a natural number, we declare this operator at the sort level, but

without the `ctor` attribute.

In the variable declaration section, we associate to the variables `I` and `J` the sort `Nat`, and to the variables `P`, `Q`, `R`, and `S` the sort `Pow`. By doing this, we are in fact declaring: (i) that `I` and `J` are variables of the kind `[Nat]`, and `P`, `Q`, `R`, and `S` of the kind `[Pow]`, and (ii) that in all memberships and equations in which those variables appear, there is an implicit extra condition stating that those variables only range over the set of terms belonging to their associated sort. Finally, in the membership declaration section, we declare that both the *tie* and the *zip* of two powerlists are powerlists if they have equal length; however, since we do not want to use the `_x_` operator as a *constructor* for powerlists, but rather as a *defined* function, we declare the membership for the `_x_` operator with the `dfn` attribute.

The module `POWERLIST` illustrates one reason why the constructor subspecification $\mathcal{E}_\Omega$ omits the defined memberships, but has the same signature as $\mathcal{E}$. Observe that the constructor membership $(P \mid Q) : Pow$ **if** $\text{len}(P) = \text{len}(Q)$ references the symbol len, but len does not appear in the left-hand side of a constructor membership. If the constructor specification did not include len in its signature, the membership axiom for the tie operator | would not be expressible in the constructor subspecification's signature. That is, in MEL specifications, unlike unsorted or many-sorted specifications, the constructor subspecification may have to include the defined functions that appear in the conditions of memberships.

5.2 Ground reducibility for CERM systems

Existing approaches to checking sufficient completeness exploit the relationship between an equational theory $\mathcal{E}$ and its associated TRS $\mathcal{R}$. However, we have not defined what it means for $\mathcal{R}/A$ to be sufficient complete. In our definition of sufficient completeness for MEL theories, sufficient completeness was expressed as an equivalence between the initial algebra $T_\mathcal{E}$ and $T_{\mathcal{E}_\Omega}$. This property can be equivalently expressed as the property that $\mathcal{E}$ and $\mathcal{E}_\Omega$ have equivalent deductive power on *ground atomic formulas*, i.e., for all ground atomic formulas α,

$$\mathcal{E} \vdash \alpha \iff \mathcal{E}_\Omega \vdash \alpha.$$

Our definition of sufficient completeness for rewrite systems is the same, but instead of atomic formulas that are equations $t = u$ or memberships $t : s$, we consider ground *join conditions* $t \downarrow u$ and ground memberships $t : s$ deduced by rewriting.

Definition 5.2.1. *Given a CERM system $\mathcal{R}/A$ where $\mathcal{R} = R \cup M$ has rewrite rules R and memberships M, and a set of memberships $M_\Omega \subseteq M$, let $\mathcal{R}_\Omega = R \cup M_\Omega$.*

We say that $\mathcal{R}/A$ *is* sufficiently complete *with respect to* M_Ω *when for all ground terms* $t, u \in T_\Sigma$ *and sorts* $s \in S$*,*

$$\mathcal{R}/A \vdash t \downarrow u \iff \mathcal{R}_\Omega/A \vdash t \downarrow u \quad \text{and} \quad \mathcal{R}/A \vdash t : s \iff \mathcal{R}_\Omega/A \vdash t : s.$$

It is worth pointing out that with this definition if both $\mathcal{R}/A$ and $\mathcal{R}_\Omega/A$ are ground weakly-normalizing and confluent so that the canonical term algebras $\text{Can}_{\mathcal{R}/A}$ and $\text{Can}_{\mathcal{R}_\Omega/A}$ are well-defined, then the definition of sufficient completeness for CERM systems has a model theoretic definition just as the definition for MEL theories possesses. Specifically, $\mathcal{R}/A$ is sufficiently complete relative to M_Ω iff $\text{Can}_{\mathcal{R}/A}$ is isomorphic to $\text{Can}_{\mathcal{R}_\Omega/A}$.

The relationship between sufficient completeness for MEL theories and the associated CERM systems is captured in the following theorem:

Theorem 5.2.2 (Suff. Comp. Equivalence)**.** *Given a CERM system* $\mathcal{R}/A$ *where* $\mathcal{R} = R \cup M$ *has rewrite rules* R *and memberships* M*, and a set of memberships* $M_\Omega \subseteq M$*, let* $\mathcal{R}_\Omega = R \cup M_\Omega$*.*

- *If* $\mathcal{R}/A$ *is ground pattern-based, confluent, sort-preserving and sufficiently complete with respect to* M_Ω*, then* $\mathcal{E} = \mathcal{R} \cup A$ *is sufficiently complete with respect to* M_Ω*.*
- *If* $\mathcal{R}_\Omega/A$ *is ground pattern-based, confluent and sort-preserving, and* $\mathcal{E} = \mathcal{R} \cup A$ *is sufficiently complete with respect to* M_Ω*, then* $\mathcal{R}/A$ *is sufficiently complete with respect to* $\mathcal{R}_\Omega/A$*.*

Proof. Let $\mathcal{E}_\Omega$ be the MEL theory $\mathcal{E}_\Omega = \mathcal{R}_\Omega \cup A$. From Definition 5.2.1 since $\mathcal{R}_\Omega$ is a subset of $\mathcal{R}$, it is clear that $\mathcal{R}/A$ is *sufficiently complete* with respect to M_Ω iff for every $t, u \in T_\Sigma$ and $s \in S$:

$$\mathcal{R}/A \vdash t \downarrow u \implies \mathcal{R}_\Omega/A \vdash t \downarrow u \quad \text{and} \quad \mathcal{R}/A \vdash t : s \implies \mathcal{R}_\Omega/A \vdash t : s$$

Furthermore, in Definition 5.1.1, the algebras $T_\mathcal{E}$ and $T_{\mathcal{E}_\Omega}$ are isomorphic iff for every $t, u \in T_\Sigma$ and $s \in S$:

$$\mathcal{E} \vdash t = u \implies \mathcal{E}_\Omega \vdash t = u \qquad \text{and} \qquad \mathcal{E} \vdash t : s \implies \mathcal{E}_\Omega \vdash t : s$$

To prove the first part of this theorem, we can use Theorem 2.6.4 with $\mathcal{R}/A$ to show that

$$\begin{aligned} \mathcal{E} \vdash t = u &\iff \mathcal{R}/A \vdash t \downarrow u \iff \mathcal{R}_\Omega/A \vdash t \downarrow u \Rightarrow \mathcal{E}_\Omega \vdash t = u \\ \mathcal{E} \vdash t : s &\iff \mathcal{R}/A \vdash t : s \iff \mathcal{R}_\Omega/A \vdash t : s \Rightarrow \mathcal{E}_\Omega \vdash t : s \end{aligned}$$

To prove the second part, we can use Theorem 2.6.4 with $\mathcal{R}_\Omega/A$ to show that

$$\begin{array}{c} \mathcal{R}/A \vdash t \downarrow u \Rightarrow \mathcal{E} \vdash t = u \iff \mathcal{E}_\Omega \vdash t = u \iff \mathcal{R}_\Omega/A \vdash t \downarrow u \\ \mathcal{R} \vdash t : s \Rightarrow \mathcal{E} \vdash t : s \iff \mathcal{E}_\Omega \vdash t : s \iff \mathcal{R}_\Omega/A \vdash t : s \end{array}$$

□

These two results show conditions under which sufficient completeness of $\mathcal{R}/A$ is *necessary* for sufficient completeness of $\mathcal{E}$, and different conditions under which it is *sufficient* for sufficient completeness of $\mathcal{E}$. In total, there are many conditions, but in practice those conditions are often satisfied, and they are even checkable mechanically using tools available in Maude. We can show that a rewrite system is weakly-normalizing by showing that it terminates in general or terminates under context-sensitive rewriting, and the Maude termination tool [44] can be used to show both termination and context-sensitive termination. The Maude confluence checker [29] is capable of checking both confluence and sort-preservation. There is currently no tool explicitly designed to check that a specification is pattern-based, however for most specifications this can be shown by checking: (1) that the syntactic restrictions on variables hold, and (2) use the Maude narrowing procedure that is part of Maude-NRL Protocol Analyzer [48] to show that the right-hand side of each oriented condition can not be narrowed.

Now that we have defined what sufficient completeness means both equationally and operationally, we turn our attention to checking sufficient completeness. Ideally, we would like an algorithm capable of deciding if a specification is sufficiently complete, and give guidance on how to fix the specification if it is not. Unfortunately, this ideal is impossible to achieve in its full generality as the problem is undecidable even for unsorted specifications with confluent and terminating rewrite systems [89].

The undecidability result in [89] stems from the fact that, in general, constructor terms may be rewritten into terms with defined symbols. If the rewrite system is such that constructor terms may only be rewritten into other constructor terms, the rewrite system is said to be *constructor-preserving*, and sufficient completeness is decidable when an unsorted rewrite system is additionally normalizing and confluent [89]. Sufficient completeness is also decidable for many-sorted specifications that are normalizing, confluent, and terminating.

These approaches for checking sufficient completeness use the fact that a normalizing, confluent, and constructor-preserving unsorted rewrite systems is sufficiently complete iff every irreducible term is a constructor term. Due to the possibility that irreducible terms may not be well-defined constructors, and the fact that a defined operation may be a constructor on part of its domain, this property does not hold for MEL specifications. However in [74], we describe a similar property which plays the same role, but generalized to the context of MEL specifications. In that work, we did not name this property, but here we

call it *defined reducibility.*

Definition 5.2.3. *Given a CERM system $\mathcal{R}/A$ over a signature Σ that contains a set of memberships $M_\Delta \subseteq \mathcal{R}$, let $\mathcal{R}_\Omega$ denote the rules $\mathcal{R} - M_\Delta$.*

The CERM system $\mathcal{R}/A$ is defined reducible *relative to a set of memberships M_Δ iff for*

- *each membership $(\forall Y)\, t : s$ **if** $\alpha_1 \wedge \cdots \wedge \alpha_n$ in M_Δ and*
- *each ground substitution $\theta : Y \to T_\Sigma$ such that $\mathcal{R}_\Omega/A \vdash \alpha_i\theta$ for $i \in [1, n]$,*

either $t\theta$ is $\mathcal{R}/A$-reducible, or $\mathcal{R}_\Omega/A \vdash t\theta : s$.

Essentially, a CERM system is defined reducible when every irreducible ground term matching a defined membership with sort s is equivalent modulo the axioms to a well-defined term with sort s in the constructor subsystem. The tools for checking sufficient completeness below are actually tools for checking defined reducibility. However, as we will show below, sufficient completeness and defined reducibility are equivalent for a large class of CERM systems.

Techniques for checking sufficient completeness of unconditional specifications typically require that the specification is weakly-normalizing, i.e., that every term t rewrites to an irreducible term u. In the context of conditional rewriting, this condition is not strong enough. We need to develop a stronger notion of weak normalization in the context of CERM systems. A *reduction order* $\prec$ on T_A is a strict partial order that is Noetherian and closed with respect to context and substitution, and a strict subsort ordering $>$ is a strict partial ordering over the sorts S.

Definition 5.2.4. *Given a CERM system $\mathcal{R}/A$ over a signature $\Sigma = (K, F, S)$, a rewrite proof using the inference system in Figure 2.6 is* reductive *relative to a reduction order $\prec$ on T_A and subsort order $<$ on S when every use of Replacement and Membership inferences:*

$$\frac{t =_A C[l\theta] \quad \bigwedge_{i\in[1,m]} u_i\theta \to^* v_i\theta \quad \bigwedge_{j\in[1,n]} w_j\theta : s_j}{t \to C[r\theta]}$$

$$\frac{t =_A l\theta \quad \bigwedge_{i\in[1,m]} u_i\theta \to^* v_i\theta \quad \bigwedge_{j\in[1,n]} w_j\theta : s_j}{t : s}$$

satisfies that $[C[r\theta]] \prec [t]$, $[u_i\theta] \prec [t]$ for each u_i, and $[w_j\theta] \prec [t]$ or $w_j =_A [t]$ and $s_j < s$ for each w_j.

We say that $\mathcal{R}/A$ is ground weakly-reductive *when:*

1. *$\mathcal{R}/A$ is ground weakly-normalizing, i.e., for each term $t \in T_\Sigma$, there is a $\mathcal{R}/A$-irreducible term $u \in T_\Sigma$ such that $\mathcal{R}/A \vdash t \to^* u$.*
2. *There is a reductive order $\prec_{\mathcal{R}/A}$ and subsort ordering $<_{\mathcal{R}/A}$ where for every ground Σ-atomic formula α derivable from $\mathcal{R}/A$, i.e. $\mathcal{R}/A \vdash \alpha$, there is a proof of α that is reductive relative to $\prec_{\mathcal{R}/A}$ and $<_{\mathcal{R}/A}$.*

The subsort ordering $<_{\mathcal{R}/A}$ is analogous to the subterm ordering in order-sorted specifications. The subsort ordering in the previous definition is typically obtained by letting $s < s'$ if $\mathcal{R}$ contains a rule of the form $x : s'$ **if** $x : s$. This essentially recovers the subsort ordering from order-sorted logic, as membership equational logic generalizes order-sorted logic by adding a membership $x : s'$ **if** $x : s$ for distinct sorts $s \leq s'$ in an order-sorted signature $\Sigma = (S, F, \leq)$, It is not difficult to show that if $\mathcal{R}/A$ is operationally terminating [44], then $\mathcal{R}/A$ is weakly-reductive. The reductive ordering $\prec_{\mathcal{R}/A}$ can then be inferred from a termination proof by the Maude Termination Tool [29].

If $\mathcal{R}/A$ is ground weakly-reductive, then we can define the following Noetherian strict ordering $\ll_{\mathcal{R}/A}$ over atomic formulas to prove equivalence of sufficient completeness and defined reducibility.

Definition 5.2.5. *Let $\mathcal{R}/A$ be a CERM system that is ground weakly reductive, and let $\prec_{\mathcal{R}/A}$ and $<_{\mathcal{R}/A}$ be the reduction and subsort orderings respectively used to show $\mathcal{R}/A$ is weakly normalizing. The Noetherian strict $\ll_{\mathcal{R}/A}$ over ground atomic formulas can be defined as follows:*

$$
\begin{aligned}
t \to u \ll_{\mathcal{R}/A} v \to w \qquad & \text{if } [t] \prec_{\mathcal{R}/A} [v] \vee t =_A v \wedge [u] \prec_{\mathcal{R}/A} [w] \\
t \to u \ll_{\mathcal{R}/A} v : s \qquad & \text{if } [t] \prec_{\mathcal{R}/A} [v] \vee t =_A v \wedge [u] \prec_{\mathcal{R}/A} [v] \\
t : s \ll_{\mathcal{R}/A} v \to w \qquad & \text{if } t \prec_{\mathcal{R}/A} v \\
t : s \ll_{\mathcal{R}/A} v : s \qquad & \text{if } [t] \prec_{\mathcal{R}/A} [v] \vee t =_A v \wedge s <_{\mathcal{R}/A} s'.
\end{aligned}
$$

with $t, u, v, w \in T_\Sigma$ and $s, s' \in S$

An important observation is noted in the following lemma, whose proof is quite simple:

Lemma 5.2.6. *Let $\mathcal{R}/A$ be a CERM that is ground weakly reductive, and let $\prec_{\mathcal{R}/A}$ be the reductive order used to show $\mathcal{R}/A$ is weakly normalizing. If $\mathcal{R}/A \vdash t \to^* u$ with $t, u \in T_\Sigma$, then $u \prec_{\mathcal{R}/A} t$ iff $u \neq_A t$.*

Proof. By induction on proof trees that are reductive with respect to $\prec_{\mathcal{R}/A}$ using the Noetherian order $\leq_{\mathcal{R}/A}$. □

Theorem 5.2.7. *Let $\mathcal{R}/A$ be a ground weakly reductive and ground sort-preserving CERM system over a signature $\Sigma = (K, F, S)$ with rules R and memberships M.*

Given a set of memberships $M_\Omega \subseteq M$, let $\mathcal{R}_\Omega = R \cup M_\Omega$, and let $M_\Delta = M - M_\Omega$. The CERM system $\mathcal{R}/A$ is sufficiently complete with respect to M_Ω iff $\mathcal{R}/A$ is defined reducible relative to M_Δ.

Proof. First observe that, for each membership $(\forall Y)\ t : s$ **if** $\alpha_1 \wedge \cdots \wedge \alpha_n$ in M_Δ and substitution $\theta : Y \to T_\Sigma$ satisfying the requirements in the definition of defined reducibility, $\mathcal{R}/A \vdash t\theta : s$.

If $\mathcal{R}/A$ is sufficiently complete with respect to M_Ω, then $\mathcal{R}/A$ is defined reducible relative to M_Δ, as due to sufficient completeness, $\mathcal{R}/A \vdash t\theta : s$ implies $\mathcal{R}_\Omega/A \vdash t\theta$.

To show the other direction, we assume that $\mathcal{R}/A$ is defined reducible relative to M_Δ, and prove that for each pair of ground terms $t, u \in T_\Sigma$ and sort $s \in S$,

$$\begin{aligned} \mathcal{R}/A \vdash t \to^* u &\implies \mathcal{R}_\Omega/A \vdash t \to^* u, \text{ and} \\ \mathcal{R}/A \vdash t : s &\implies \mathcal{R}_\Omega/A \vdash t : s \end{aligned} \tag{5.1}$$

If this is true, then $\mathcal{R}/A$ is sufficiently complete relative to $\mathcal{R}_\Omega/A$.

As $\mathcal{R}/A$ is weakly normalizing, each atomic formula α entailed by $\mathcal{R}/A$ can be shown with a proof that is reductive relative to $\prec_{\mathcal{R}/A}$ and $<_{\mathcal{R}/A}$. Let $\ll_{\mathcal{R}/A}$ be the order of atomic formulas formed from $\prec_{\mathcal{R}/A}$. We show (5.1) by induction over $\ll_{\mathcal{R}/A}$ on reductive proofs. If the top of the proof is an **Equivalence** rule,

$$\frac{t =_A u}{t \to^* u}$$

then clearly $\mathcal{R}_\Omega/A \vdash t \to^* u$. If the top of the proof tree is a **Transitivity** rule, the left antecedent must be a **Rewrite** rule.

$$\frac{\dfrac{t =_A C[l\theta] \quad \bigwedge_{i\in[1,m]} u_i\theta \to v_i\theta \quad \bigwedge_{j\in[1,n]} W_j i\theta : s_j}{t \to C[r\theta]} \qquad \dfrac{\ldots}{C[r\theta] \to^* t'}}{t \to^* t'}$$

As the proof is reductive relative to $\prec_{\mathcal{R}/A}$, we have that:

- $C[r\theta] \to t' \ll_{\mathcal{R}/A} t \to t'$,
- $u_i\theta \to v_i\theta \ll_{\mathcal{R}/A} t \to t'$ for each $i \in [1, m]$, and
- $w_j\theta : s_j \ll_{\mathcal{R}/A} t \to t'$ for each $j \in [1, n]$.

By induction, each antecedent is provable in $\mathcal{R}_\Omega/A$, and thus $\mathcal{R}_\Omega/A \vdash t \to^* t'$.

We next consider the case where the top of the proof is **Subject Reduction**.

$$\frac{t \to u \quad u : s}{t : s}$$

In this case, the proof that $t \to u$ must be a **Rewrite** rule, and consequently $[u] \prec [t]$. A similar argument to the previous one allows us to conclude that $\mathcal{R}_\Omega/A \vdash t \to u$, and since $[u] \prec [t]$, it follows by induction on $\ll_{\mathcal{R}/A}$ that $\mathcal{R}_\Omega/A \vdash u : s$. Consequently $\mathcal{R}_\Omega/A \vdash t : s$.

Finally, we consider the case where the proof is a **Membership** rule:

$$\frac{t =_A l\theta \quad \bigwedge_{i\in[1,m]} u_i\theta \to v_i\theta \quad \bigwedge_{j\in[1,n]} W_j\theta : s_j}{t : s}$$

By induction, $\mathcal{R}_\Omega/A \vdash u_i\theta \to^* v_i\theta$ for each $i \in [1, m]$, and $\mathcal{R}_\Omega/A \vdash w_j\theta : s_j$ for each $j \in [1, n]$. If the membership used to prove $\mathcal{R}/A \vdash t : s$ is in M_Ω, then we

immediately have that $\mathcal{R}_\Omega/A \vdash t : s$. Otherwise, the membership must be in M_Δ. If t is $\mathcal{R}/A$-irreducible, then $\mathcal{R}_\Omega/A \vdash t\theta : s$ as $\mathcal{R}/A$ is defined reducible.

On the other hand, assume that t is $\mathcal{R}/A$-reducible. As $\mathcal{R}/A$ is weakly-normalizing, there is a $\mathcal{R}/A$-irreducible term $t\downarrow \in T_\Sigma$ such that $\mathcal{R}/A \vdash t \rightarrow^* t\downarrow$. Moreover, $\mathcal{R}/A \vdash t\downarrow : s$ as $\mathcal{R}/A$ is sort-preserving. As t is $\mathcal{R}/A$-reducible and $t\downarrow$ is $\mathcal{R}/A$-irreducible, it follows that $t \neq_A t\downarrow$, and therefore $t\downarrow \prec_{\mathcal{R}/A} t$ by Lemma 5.2.6. As $t \rightarrow t\downarrow \prec_{\mathcal{R}/A} t : s$ and $t\downarrow : s \prec_{\mathcal{R}/A} t : s$, it follows that $\mathcal{R}_\Omega/A \vdash t \rightarrow^* t\downarrow$ and $\mathcal{R}_\Omega/A \vdash t\downarrow : s$ by our induction hypothesis. Consequently, $\mathcal{R}_\Omega/A \vdash t : s$. □

We have not yet developed a general purpose tool for checking whether an arbitrary CERM system is defined reducing. This seems quite difficult due to the need to consider both conditional rules and rewriting modulo axioms. However, we have developed two different sufficient completeness checkers capable of checking important subcases that fall outside the domain supported by other sufficient completeness checkers:

- The checker of [73], which can check conditional specifications, but cannot check specification with rewriting modulo axioms. The checker can succeed in automatically checking sufficient completeness in some restricted cases, but in general, generates proof-obligations which imply defined reducibility for conditional MEL specifications if they can be shown true in the initial model of a specification. These proof obligations are passed directly to the Maude Inductive Theorem Prover (ITP) for discharging by the user.

- The equational tree automata-based checker of [75], which is capable of checking sufficient completeness for unconditional left-linear specifications with rewriting modulo axioms by casting sufficient completeness as an equational tree automata decision problem.

5.3 Checking sufficient completeness with Maude ITP

Our first sufficient completeness checker, the Maude ITP Sufficient Completeness Checker (ITP-SCC) is capable of checking Maude specifications with conditional rules such as the powerlist specification seen earlier. It does not support rewriting modulo axioms, so in this section the set of axioms $A = \varnothing$. We additionally assume that the memberships in $\mathcal{E}$ satisfy the following property:

Definition 5.3.1. *An MEL theory $\mathcal{E}$ is* properly sorted *over a signature $\Sigma = (K, F, S)$ iff there is an ordering $<$ over S such that for each memberships $t : s$* **if** $\overline{\alpha}$ *in $\mathcal{E}$:*

- *if t is a variable $x \in X$, and $\overline{\alpha}$ contains the membership formula $x : s'$, then $s' < s$; and*
- *for each variable $x \in \mathrm{vars}(t) \cup \mathrm{vars}(\overline{\alpha})$, there is a membership with the form $x : s \in \overline{\alpha}$.*

Neither of these two assumption are very severe — the first is required by Maude theories to obtain termination, while the second can be obtained from an arbitrary theory by introducing a fresh maximal sort s_k for each kind k, adding memberships so that every term $t \in T_{\Sigma,k}$ has sort s_k, and introducing memberships assumptions $x : s_k$ in memberships where needed.

The checker is itself written in Maude using *reflection*. The soundness of the tool is based on Theorem 5.2.7. There are two major components to the tool: a *sufficient completeness analyzer*, which generates proof obligations for sufficient completeness problems, and the Maude *Inductive Theorem Prover* (ITP), extended with additional commands to try to automatically prove those proof obligations. The tool has been run on a variety of different MEL specifications, and is available for download with source, documentation, and examples (including MEL specifications of ordered lists with sorting functions, stacks, and binary trees) from the tool's webpage:

`http://maude.cs.uiuc.edu/tools/scc/`

5.3.1 The sufficient completeness analyzer

The sufficient completeness analyzer follows the incremental constructor-based narrowing of patterns approach, but generalized to handle conditional specifications. The algorithm is similar to the algorithm for coverset induction described in Chapter 7. Given a MEL specification $\mathcal{E}$ written in Maude, annotated to indicate a constructor memberships M_Ω, the Maude sufficient completeness analyzer generates, in a two phase process, a set of proof obligations which if discharged, ensures the sufficient completeness of $\mathcal{E}$ relative to M_Ω. The sufficient completeness analyzer assumes that the CERM system $\mathcal{R}$ associated to $\mathcal{E}$ satisfies the conditions in Theorem 5.2.7.

Narrowing procedure

In its first phase, the analyzer returns a finite set Δ containing tuples $(t, s, \overline{\alpha})$ where t is a term in $T_\Sigma(X)_k$ for some $k \in K$, $s \in S_k$ and $\overline{\alpha}$ is a finite conjunction of atomic formulas $\overline{\alpha} = \alpha_1 \wedge \ldots \alpha_n$. The set Δ returned by the procedure has the property that if $t' \in T_\Sigma$ is a counterexample for sufficient completeness, then there exists a triple $(t, s, \overline{\alpha}) \in \Delta$ and a substitution $\theta : \mathrm{vars}(t) \to T_\Sigma$ such that $t' = \theta t$ and $T_{\mathcal{E}_\Omega} \vdash \overline{\alpha}\theta$. The set Δ is generated from the initial set

$$\Delta_0 = \{\, (t, s, \overline{\alpha}) \mid t : s \textbf{ if } \overline{\alpha} \in M_\Delta \,\}.$$

We then apply the rule (5.2) below until completion. This rule (5.2) uses the *expandability relation* $\blacktriangleleft$ and the *expand function* exp which are defined as follows:

Definition 5.3.2. *Let t, t' be terms in $T_\Sigma(X)$ such that $\mathrm{vars}(t) \cap \mathrm{vars}(t') = \emptyset$, and $x \in \mathrm{vars}(t)$. Then, $t \blacktriangleleft_x t'$ iff t and t' are unifiable and in the most general unifier $\theta = \mathrm{mgu}(t, t')$, $\theta(x)$ is not a variable.*

Definition 5.3.3. *Let $t \in T_\Sigma(Y)$, $s \in S_k$, $\overline{\alpha}$ a conjunction of atomic formulas with variables in Y, $x \in X$ with $x : s_x \in \overline{\alpha}$, and M a set of memberships whose variables have all been renamed to be disjoint from Y. Then,*

$$\exp(t, s, \overline{\alpha}, x, M) = \{\, (t[x/u], s, \overline{\alpha}[x/u] \wedge \overline{\alpha}') \mid u : s_x \textbf{ if } \overline{\alpha}' \in M \,\}$$

Finally, we define the inference rule that generates the set Δ. Note that this rule will only be applied a finite number of times, because of the condition $t \blacktriangleleft_x t'$ on the rule.

Δ-rule For each term t' in the left-hand side of an equation in $\mathcal{E}$,

$$\frac{\Delta' \cup \{(t, s, \overline{\alpha})\}}{\Delta' \cup \exp(t, s, \overline{\alpha}, x, M_\Omega)} \quad \text{if } x \in \mathrm{vars}(t) \text{ and } t \blacktriangleleft_x t' \tag{5.2}$$

The number of applications of this rule must terminate, because our assumptions that the MEL theory $\mathcal{E}$ is properly sorted. The termination argument can use the fact that each variable x used in the rule is replaced by a non-variable term t or the sort constraint $x : s$ in $\overline{\alpha}$ is replaced with a lower sort constraint $x : s'$ with $s' < S$.

Proof obligation generator

In its second phase, the SCC produces, from the set Δ, a set of proof obligations which if discharged, guarantees that $\mathcal{E}$ is sufficiently complete with respect to M_Ω under the assumptions in Theorems 5.2.2 and 5.2.7. Since a triple $(t, s, \overline{\alpha}) \in \Delta$ represents a set of potential counterexamples, the proof obligation generator produces a sentence which if proven inductively in $\mathcal{E}_\Omega$, implies that for every substitution $\theta : \mathrm{vars}(t) \to T_\Sigma$ at least one of the following holds:

a) $\mathcal{E}_\Omega \not\models \overline{\alpha}\theta$

b) $t\theta$ is *reducible*

c) There exists a membership $u : s'$ **if** $\overline{\alpha}'$ in M_Ω with $s' < s$ and a substitution $\tau : Y \to T_\Sigma$ such that $t\theta = u\tau$ and $\mathcal{E}_\Omega \models \overline{\alpha}'\tau$.

If the proofs are discharged, then by Theorem 5.2.7, $\mathcal{R}$ is sufficiently complete with respect to M_Ω. For each $(t, s, \overline{\alpha}) \in \Delta$, the proof obligation generator

constructs the sentence:

$$(\forall \text{vars}(t)) \left(\overline{\alpha} \implies \bigvee_{\substack{l=r \text{ if } \overline{\alpha}' \in \mathcal{E}, \\ \theta \text{ s.t. } C[l\theta]=t}} \overline{\alpha}'\theta \quad \vee \bigvee_{\substack{l:s' \text{ if } \overline{\alpha}' \in M_\Omega \text{ s.t. } s'<s, \\ \theta \text{ s.t. } l\theta=t}} \overline{\alpha}'\theta \right) \tag{5.3}$$

The first phase's algorithm is quite similar to previous algorithms in sufficient completeness checking for unsorted theories. The difference is that we specialize using constructor memberships rather than constructor symbols. Due to the conditions in the rules and memberships in the specification, our first phase is not a decision procedure. Instead, in the second phase of our algorithm, the patterns from the first phase are matched against the left-hand side of each rule in the specification. If the pattern matches an unconditional rule, we discard it. Otherwise, we generate a proof obligation for the pattern which is sufficient to show that instances of the pattern are reducible or constructor terms. These statements the second phase generated are then directly passed to the Maude Inductive Theorem Prover (ITP) to be discharged.

5.3.2 Maude ITP

The Maude ITP [27] is an experimental interactive tool for proving properties of the initial algebra $T_{\mathcal{E}}$ of MEL specifications in Maude. It is described in more detail in Chapter 7, but we will introduce some of the design decisions here to keep this chapter more self contained.

The ITP tool has been written entirely in Maude, and is in fact an *executable* specification in MEL of the formal inference system that it implements. The ITP inference system treats MEL specifications as *data* — for example, one inference may add to the specification an induction hypothesis as a new equational axiom. This *reflective* design, in which Maude equational specifications become data at the metalevel, is ideally suited for implementing the ITP. Using reflection to implement the ITP tool has one important additional advantage, namely, the ease to rapidly extend it by integrating other tools implemented in Maude using reflection, as it is the case of the SCC.

In the ITP, the user introduces commands which are interpreted as actions that may change the state of the proof, specifically the set of goals that remain to be proved, with each goal consisting of a formula to be proved and the MEL specification in which the formula must be proved. After executing the action requested by the user, the tool reports the resulting state of the proof. The main module implementing the ITP is the `ITP-TOOL` module. In this module, states of proofs, sets of goals, goals and formulas are represented by terms of different sorts, and the actions interpreting the ITP commands are represented as different, equationally defined functions over those terms.

The SCC has been integrated with an older version of the ITP by adding two two new commands, `scc` and `scc*`, to the ITP; the `scc*` command is an exten-

sion of `scc` that takes into account the information obtained by this command *at run-time*. We begin with the `scc` command. This command is implemented by extending the module `ITP-TOOL` with a new, equationally defined function that, given an equational specification $\mathcal{E}$, does the following:

- first, it calls on $\mathcal{E}$ the function `checkCompleteness`, which implements the sufficient completeness analyzer described in Section 5.3.1;
- then, it converts the resulting proof obligations into a set of ITP goals, which are all associated with $\mathcal{E}_\Omega$;
- finally, it eliminates from the state of the proof those goals that can be proved automatically using the ITP `auto*` command.[1]

5.3.3 Example

We illustrate the algorithm with the powerlist example from Section 5.1. In the powerlist specification, the relevant defined memberships M_Δ are:

```
cmb (P zip Q) : Pow if len(P) = len(Q) [metadata "dfn"].
mb len(P): Nat  [metadata "dfn"].
```

Our initial patterns with conditions are then just

$$P \times Q : \text{Pow} \textbf{ if } \text{len}(P) = \text{len}(Q) \wedge P : \text{Pow} \wedge Q : \text{Pow}$$
$$\text{len}(P) : \text{Nat} \textbf{ if } P : \text{Pow}$$

We have written each pattern as a conditional membership, because it has the information as a membership: a term for the pattern, a sort for the sort we know this term has in the overall theory $\mathcal{E}$, and a condition on the variables in the term. Since the first pattern can unify with, but does not match, the left hand side of the equation `[I] zip [J] = [I] tie [J]`, our algorithm expands P using the constructor memberships M_Ω with sort Pow. We then replace the first pattern with two new patterns:

$$[I] \times Q : \text{Pow} \textbf{ if } \text{len}([I]) = \text{len}(Q) \wedge I : \text{Nat} \wedge Q : \text{Pow}$$
$$(P_1 \mid P_2) \times Q : \text{Pow} \textbf{ if}$$
$$\quad \text{len}(P_1 \mid P_2) = \text{len}(Q) \wedge \text{len}(P_1) = \text{len}(P_2) \wedge P_1 : \text{Pow} \wedge P_2 : \text{Pow} \wedge Q : \text{Pow}$$

[1] The implementation of the `auto*` command in the ITP-SCC integrates its rewriting-based simplification strategy with a decision procedure for linear arithmetic with uninterpreted function symbols; this theory includes many of the formulas that one tends to encounter in proof obligations generated by the SCC tool.

This process is repeated with different patterns until terminating with the following patterns:

$$[I] \times [J] : \text{Pow} \textbf{ if } \text{len}([I]) = \text{len}([J]) \wedge I : \text{Nat} \wedge J : \text{Nat} \tag{5.4}$$

$$[I] \times (Q_1 \mid Q_2) : \text{Pow} \textbf{ if } \text{len}([I]) = \text{len}(Q_1 \mid Q_2) \wedge \text{len}(Q_1) = \text{len}(Q_2) \wedge I : \text{Nat} \wedge Q_1 : \text{Pow} \wedge Q_2 : \text{Pow} \tag{5.5}$$

$$(P_1 \mid P_2) \times [J] : \text{Pow} \textbf{ if } \text{len}(P_1 \mid P_2) = \text{len}([J]) \wedge \text{len}(P_1) = \text{len}(P_2) \wedge P_1 : \text{Pow} \wedge P_2 : \text{Pow} \wedge J : \text{Nat} \tag{5.6}$$

$$(P_1 \mid P_2) \times (Q_1 \mid Q_2) : \text{Pow} \textbf{ if } \text{len}(P_1 \mid P_2) = \text{len}(Q_1 \mid Q_2) \wedge \text{len}(P_1) = \text{len}(P_2) \wedge \text{len}(Q_1) = \text{len}(Q_2) \wedge P_1 : \text{Pow} \wedge P_2 : \text{Pow} \wedge Q_1 : \text{Pow} \wedge Q_2 : \text{Pow} \tag{5.7}$$

$$\text{len}([I]) : \text{Nat} \textbf{ if } I : \text{Nat} \tag{5.8}$$

$$\text{len}(P_1 \mid P_2) : \text{Nat} \textbf{ if } \text{len}(P_1) = \text{len}(P_2) \wedge P_1 : \text{Pow} \wedge P_2 : \text{Pow} \tag{5.9}$$

We can immediately discard patterns (5.4), and (5.7)–(5.9): they match the left-hand sides of unconditional equations in the powerlist specification, and thus all the ground instances of these patterns are reducible. Moreover, as the patterns (5.8) and (5.9) are the only patterns for the `len` membership and `len` does not recursively depend on the other defined membership, we can discharge the length membership and add it to the specification before discharging the remaining obligations.

Neither of the remaining patterns (5.5) and (5.6) can match an equation, but each pattern has conditions which we would like to show are unsatisfiable. These patterns are passed to the proof obligation generator which returns the following two proof obligations.

$$\text{len}([I]) = \text{len}(Q_1 \mid Q_2) \wedge \text{len}(Q_1) = \text{len}(Q_2) \wedge I : \text{Nat} \wedge Q_1 : \text{Pow} \wedge Q_2 : \text{Pow} \Rightarrow [I] \times (Q_1 \mid Q_2) : \text{Pow}$$

$$\text{len}(P_1 \mid P_2) = \text{len}([J]) \wedge \text{len}(P_1) = \text{len}(P_2) \wedge P_1 : \text{Pow} \wedge P_2 : \text{Pow} \wedge J : \text{Nat} \Rightarrow (P_1 \mid P_2) \times [J] : \text{Pow}$$

Both of these proof obligations can be discharged by the ITP automatically. By rewriting on the first proof obligation, we are able to reduce it to:

$$1 = \text{len}(Q_1) + \text{len}(Q_2) \wedge \text{len}(Q_1) = \text{len}(Q_2) \wedge I : \text{Nat} \wedge Q_1 : \text{Pow} \wedge Q_2 : \text{Pow} \Rightarrow [I] \times (Q_1 \mid Q_2) : \text{Pow}$$

Next, the ITP's linear arithmetic decision procedure can be used to show that the antecedents $1 = \text{len}(Q_1) + \text{len}(Q_2) \wedge \text{len}(Q_1) = \text{len}(Q_2)$ in the goal is un-

satisfiable, and therefore we can discharge the first obligation. The same two steps of rewriting and linear arithmetic are able to discharge the second obligation as well, and thus the powerlist specification in Section 5.1 is proved to be sufficiently complete.

5.4 Tree automata-based checking

The sufficient completeness checker in the previous section deals well with conditional rules, but has difficulty with specifications using the other important extension of CERM systems — rewriting modulo axioms. This section presents a second sufficient completeness checker based on *equational tree automata* techniques that is capable of checking specifications with rewriting modulo axioms. Due to the arbitrary conditions that may appear in memberships, it appears quite difficult to apply tree automata techniques to arbitrary MEL specifications. However, our tool is supports order-sorted, left-linear and unconditional specifications with rewriting modulo any combination of associativity, commutativity, and identity axioms. This class appears quite small relative to the much more general class of arbitrary MEL specifications, but it contains many interesting specifications that existing tools have not been able to check.

As an example, our tree automata-based checker is capable of handling the `NAT-LIST` example from Section 2.4 which we reproduce below:

```
fmod NAT-LIST is protecting NAT .
  sort NeList List .
  subsorts Nat < NeList < List .

  op nil : -> List [ctor].
  op __ : NeList NeList -> NeList [ctor assoc id: nil].
  op __ : List List -> List [assoc id: nil].

  var N : Nat .   var L : List .

  op head : NeList -> Nat .
  eq head(N L) = N .
  op end : NeList -> Nat .
  eq end(L N) = N .

  op reverse : List -> List .
  eq reverse(N L) = reverse(L) N .
  eq reverse(nil) = nil .
endfm
```

In this specification, the operator `nil` is a constructor, while the operator `__` is overloaded: it is defined on all lists, but only a constructor on non-empty lists. The operations `head` and `end` are partial operations which are only defined on non-empty lists while `reverse` is defined on all lists.

The previous sufficient completeness checker described in Section 5.3 is not able to show the sufficient completeness of `NAT-LIST`. That checker would specialize the associative append symbol to construct terms that are *wider*, such as $\mathrm{head}(l_1(l_2 l_3))$, that still would not match any equations. For this approach to work, one must be able to bound the width of the terms we consider. For unsorted and many-sorted left-linear specifications with rewriting modulo AC, this is possible as shown by Jouannaud and Kounalis [82]. However, it appears quite difficult to extend their results to the order-sorted case.

To deal with the order-sorted case, we cast the sufficient completeness problem with rewriting modulo axioms as a decision problem for *equational tree automata* [123]. Equational tree automata extend regular tree automata to allow some of the symbols to have equational properties such as associativity and commutativity. The automaton then recognizes languages that are closed modulo those equational properties. This is important, because when rewriting modulo, the set of reducible terms contains not only the set of terms that *syntactically* match a rule, but also terms equivalent modulo the axioms to syntactically matching terms.

The idea of using tree automata in checking sufficient completeness is not a new one. Tree automata techniques were used to yield a sufficient completeness checking algorithm that was optimal from the complexity theory point of view. It was shown in [90] that sufficient completeness was EXPTIME-hard, but an exponential time algorithm for checking it was unknown. This problem remained open for several years, until [32] described an exponential time algorithm for checking sufficient completeness that worked by casting the problem as a decision problem for reduction tree automata [24].

The results in [32] depend on the fact that reduction tree automata are capable of recognizing the reducible terms of a rewrite system. For rewriting modulo axioms A, this set of terms must be closed modulo A. In casting sufficient completeness of a specification as an equational tree automata with equations A, we lose the support for non-linear constraints of reduction tree automata, but gain the ability to recognize equationally closed sets of terms.

The class of CERM systems which our tool can handle correspond to the order-sorted subset that are ground weakly-normalizing, ground sort-preserving and have left-linear rules. This class is defined precisely below:

Definition 5.4.1. *A CERM system $\mathcal{R}/A$ is* TA checkable *when*

(a) $\mathcal{R}/A$ is ground weakly reductive and ground sort-preserving.

(b) Every axiom in $\mathcal{R}$ has the form α **if** $x_1 : s_1 \wedge \cdots \wedge x_n : s_n$ *where $x_1, \ldots, x_n$ are distinct,* $\mathrm{vars}(\alpha) \subseteq \{ x_1, \ldots, x_n \}$ *and each variable appears at most once in the left-hand-side of α.*

The details for how to convert the sufficient completeness property for rewriting modulo into a propositional emptiness problem were originally presented

in [74]. Our presentation here is slightly different, but captures the same basic idea. The key idea is to construct an automaton $\mathcal{A}_{\mathsf{SC}}$ with two different types of states: (1) for each sort $s \in S$, $\mathcal{A}_{\mathsf{SC}}$ contains states c_s and d_s: c_s recognizes terms with sort s using only the memberships in M_Ω, and d_s recognizes terms with a defined root operator and constructors underneath; (2) $\mathcal{A}_{\mathsf{SC}}$ contains states for recognizing intermediate subterms in the left-hand side of rules in $\mathcal{R}$ as well as a state r which recognizes $\mathcal{R}/A$-reducible terms. We then define the propositional formula $\phi = \neg r \wedge \bigvee_{s \in S} d_s \wedge \neg c_s$.

If we recall from Section 2.3 that $\overline{A}$ denotes the underlying unsorted equational theory obtained from the axioms A, then it is not difficult to show the following theorem relating sufficient completeness and $\mathcal{L}_\phi(\mathcal{A}_{\mathsf{SC}}/\overline{A})$.

Theorem 5.4.2. *Let $\mathcal{R}/A$ be a TA checkable CERM system with rules R, memberships M, and a signature $\Sigma = (K, F, S)$ Given a set of memberships $M_\Omega \subseteq M$, let $\mathcal{R}_\Omega = R \cup M_\Omega$.*

There effectively exists a equational tree automata $\mathcal{A}_{\mathsf{SC}}$ and propositional formula ϕ such that $\mathcal{R}/A$ is sufficiently complete relative to constructor memberships M_Ω iff $\mathcal{L}_\phi(\mathcal{A}_{\mathsf{SC}}/\overline{A}) = \varnothing$.

Proof. We observe that since the memberships in a TA checkable specification do not have equations in the conditions, if t is $\mathcal{R}/A$-irreducible and $\mathcal{R}/A \vdash t : s$ or respectively $\mathcal{R}_\Omega/A \vdash t : s$, then $M/A \vdash t : s$ or respectively $M_\Omega/A \vdash t : s$.

Every TA checkable CERM system $\mathcal{R}/A$ is ground weakly reductive and ground sort-preserving, consequently we can reduce checking the sufficiently completeness of $\mathcal{R}/A$ relative to M_Ω to checking defined reducibility of $\mathcal{R}/A$ relative to $M_\Delta = M - M_\Omega$. We check this by defining a language $\mathcal{L}_\phi(\mathcal{A}_{\mathsf{SC}}/\overline{A})$ which contains an equivalence class $[t] \in T_A$ iff there exists an $\mathcal{R}/A$-irreducible ground term $t \in T_\Sigma$ for which there exists

- a membership $(\forall \overline{x} : \overline{s})\, l : s$ in M_Δ and
- a ground substitution $\theta : \overline{x} \to T_\Sigma$

such that $M_\Omega/A \vdash \theta(x) : s_x$ for all variables $x \in \overline{x}$, $t =_A l\theta$, and $M_\Omega/A \nvdash t : s$. Since the equations in A are kind independent (see Def. 2.6.1), we have that $\mathcal{L}_\phi(\mathcal{A}_{\mathsf{SC}}/\overline{A}) = \varnothing$ iff $\mathcal{R}/A$ is sufficiently complete relative to M_Ω.

In order to define $\mathcal{L}_\phi(\mathcal{A}_{\mathsf{SC}}/\overline{A})$, we define the set of $I_\mathcal{R}$ which denotes the non-variable strict subterms appearing in the left-hand side of clauses in $\mathcal{R}$. The elements in $I_\mathcal{R}$ are further annotated with the sorts bound to each variable in a clause. Specifically,

$$I_\mathcal{R} = \{\, t[\overline{x} : \overline{s}] \mid (\forall \overline{x} : \overline{s})\, \alpha \text{ in } \mathcal{R} \wedge C[t] \in \mathrm{lhs}(\alpha) \wedge t \notin X \wedge C \neq \Box \,\}.$$

The states Q of the automaton $\mathcal{A}_{\mathsf{SC}}$ is the set

$$Q = \{\, r^\mu, q_\top \,\} \cup \{\, d_s, c_s \mid s \in S \,\} \cup \{\, c_{u[\overline{x}:\overline{s}]} \mid u[\overline{x} : \overline{s}] \in I_\mathcal{R} \,\}.$$

To simplify later notation for each variable x appearing the left-hand side of a rule $(\forall \overline{x} : \overline{s})\alpha$ in $\mathcal{R}$, we identify $c_{x[\overline{x}:\overline{s}}$ with c_{s_x} where s_x is the variable associated to x in $\overline{x} : \overline{s}$.

We define the clauses in $\mathcal{A}_{\mathtt{SC}}$ as follows.

- For each term $f(t_1, \ldots, t_n)[\overline{x} : \overline{s}] \in I_{\mathcal{R}}$, $\mathcal{A}_{\mathtt{SC}}$ contains

$$c_{f(t_1,\ldots,t_n)[\overline{x}:\overline{s}]}(f(x_1, \ldots, x_n)) \Leftarrow c_{t_1[\overline{x}:\overline{s}]}(x_1), \ldots, c_{t_n[\overline{x}:\overline{s}]}(x_n)$$

- For each constructor membership $(\forall \overline{x} : \overline{s})\ f(t_1, \ldots, t_n) : s$ in M_Ω, $\mathcal{A}_{\mathtt{SC}}$ contains

$$c_s(f(x_1, \ldots, x_n)) \Leftarrow c_{t_1[\overline{x}:\overline{s}]}(x_1), \ldots, c_{t_n[\overline{x}:\overline{s}]}(x_n)$$

- For each constructor membership $(\forall \overline{x} : \overline{s})\ x : s$ in M_Ω, $\mathcal{A}_{\mathtt{SC}}$ contains

$$c_s(x) \Leftarrow c_{s_x}(x).$$

- For each defined membership $(\forall \overline{x} : \overline{s})\, f(t_1, \ldots, t_n) : s$ in M_Δ, $\mathcal{A}_{\mathtt{SC}}$ contains

$$d_s(f(x_1, \ldots, x_n)) \Leftarrow c_{t_1[\overline{x}:\overline{s}]}(x_1), \ldots, c_{t_n[\overline{x}:\overline{s}]}(x_n)$$

- For each defined membership $(\forall \overline{x} : \overline{s})\ x : s$ in M_Δ, $\mathcal{A}_{\mathtt{SC}}$ contains

$$d_s(x) \Leftarrow d_{s_x}(x).$$

- For each operator $f : k_1 \ldots k_n \to k$ in F, $\mathcal{A}_{\mathtt{SC}}$ contains

$$q_\top(f(x_1, \ldots, x_n)) \Leftarrow q_\top(x_1), \ldots, q_\top(x_n).$$

- For each rule $(\forall \overline{x} : \overline{s})\ f(t_1, \ldots, t_n) \to u$ in R, $\mathcal{A}_{\mathtt{SC}}$ contains

$$r(f(x_1, \ldots, x_n)) \Leftarrow c_{t_1[\overline{x}:\overline{s}]}(x_1), \ldots, c_{t_n[\overline{x}:\overline{s}]}(x_n).$$

- For each rule $(\forall \overline{x} : \overline{s})\ y \to u$ in R, $\mathcal{A}_{\mathtt{SC}}$ contains

$$r(x) \Leftarrow c_{s_y}(x) \quad \text{and} \quad r(x) \Leftarrow d_{s_y}.$$

- Finally, for each operator $f \in F$ with arity n and index $i \in [1, n]$, $\mathcal{A}_{\mathtt{SC}}$ contains

$$r(f(x_1, \ldots, x_n)) \Leftarrow q_\top(x_1), \ldots, r(x_i), \ldots, q_\top(x_n).$$

By induction on $t \in T_{\overline{\Sigma}}$, we have

$$t \in \mathcal{L}_{c_s}(\mathcal{A}_{\mathsf{SC}}) \iff M_\Omega \vdash t : s, \text{ and}$$
$$t \in \mathcal{L}_{c_{u[\overline{x}:\overline{s}}}}(\mathcal{A}_{\mathsf{SC}}) \iff (\exists \theta) t = u\theta \wedge (\forall x \in \overline{x})\, M_\Omega \vdash \theta(x) : s.$$

We can use these results and the fact that $\mathcal{L}_{q_\top}(\mathcal{A}_{\mathsf{SC}}) = T_{\overline{\Sigma}}$, to show that $t \in \mathcal{L}_r(\mathcal{A}_{\mathsf{SC}})$ iff there is a t is $\mathcal{R}$-reducible, and to show that the terms in $\mathcal{L}_{d_s}(\mathcal{A}_{\mathsf{SC}})$ are those whose root has a sort using a defined membership and whose subterms are constructors.

To reduce sufficient completeness to a propositional emptiness problem for equational tree automata, we define the formula

$$\phi = \neg r \wedge \bigvee_{s \in S} d_s \wedge \neg c_s.$$

This formula defies a language accepting irreducible terms that accepted by the language $\mathcal{L}_{d_s}(\mathcal{A}_{\mathsf{SC}}/\overline{A})$ for some sort $s \in s$ that are not constructor terms with sort s. By the restrictions on A in the Definition 2.6.1 of CERM systems, it follows from Theorem 3.1.1 that $\mathcal{L}_\phi(\mathcal{A}_{\mathsf{SC}}/\overline{A})$ contains exactly the counterexamples to defined reducibility. Thus $\mathcal{R}/A$ is sufficiently complete iff $\mathcal{L}_\phi(\mathcal{A}_{\mathsf{SC}}/\overline{A}) = \varnothing$ by Theorem 5.2.7. □

The decidability of the above emptiness problem depends on the particular axioms A. It is decidable when the axioms in the specification are any combination of associativity, commutativity, and identity, except when a symbol is associative but not commutative. For the case of commutativity alone, this was shown in [123]. For symbols that are both associative and commutative, this was shown in [125]. Identity equations can be transformed into identity rewrite rules using a specialized completion procedure along the lines of coherence completion in [140], and then we can extend the emptiness test to only recognize terms that are in normal form with respect to identity rewrite rules.

For symbols that are associative and not commutative, the problem is undecidable. However, for these associative symbols, we can use the semi-algorithm presented in Chapter 3. The semi-algorithm presented in that work is capable of always showing non-emptiness if a language is non-empty, and capable of showing emptiness if the language is empty and certain regularity conditions are satisfied. What this means for sufficient completeness checking is that we can always find counterexamples to sufficient completeness if they exist, and can show sufficient completeness in most practical specifications, where the sorts in a specification are used to model regular data structures like lists or non-empty lists.

The implementation of the tree automata based SCC has two major components: an analyzer written in Maude that generates the tree automaton emptiness problem from a Maude specification; and a C++ library called CETA that

performs the emptiness check.

Analyzer. The analyzer accepts commands from the user, generates a propositional emptiness problem from a Maude specification, forwards the problem to CETA, and presents the user with the results. If the specification is not sufficiently complete, the tool shows the user a counterexample illustrating the error. The analyzer consists of approximately 900 lines of Maude code, and exploits Maude's support for reflection. The specifications it checks are also written in Maude.

If the user asks the tool to check the sufficient completeness of a specification that is not left-linear and unconditional, the tool transforms the specification by renaming variables and dropping conditions into a checkable order-sorted left-linear specification. Even if the tool is able to verify the sufficient completeness of the transformed specification, it warns the user that it cannot show the sufficient completeness of the original specification. However, any counterexamples found in the transformed specification are also counterexamples in the original specification. We have found this feature quite useful to identify errors in Maude specifications falling outside the decidable class — including the sufficient completeness checker itself.

CETA. The propositional tree automaton generated by the analyzer is forwarded to the CETA tree automata library which we have developed. CETA is a complex C++ library with approximately 10 thousand lines of code. Emptiness checking is performed by a subset construction algorithm extended with support for associative and commutativity axioms as described in Chapter 3. The reason that CETA is so large is that the subset construction algorithm relies on quite complex algorithms on context free grammars, semilinear sets, and finite automata.

We have found that CETA performs quite well for our purposes. Most examples can be verified in seconds. A table with a few of the checked specifications from the Maude prelude, Maude primer [108] and Maude book [28] is shown in Figure 5.1. All successfully checked modules are sufficiently complete, however modules are in italics if the sufficient completeness checker identified errors in early versions. The column labeled $|\mathcal{E}|$ indicates the total number of sorts, operators, and equations in the theory $\mathcal{E}$, while the column labeled |ETA| indicates the total number of states, operators, and rules in the corresponding automaton. The current version of the checker is not fast enough to verify itself in less than our time limit of 30 minutes, but has been able to successfully identity real sufficient completeness errors in early versions of the checker.

As an example, in Figure 5.2, we present a tool session in which we check two specifications: `NAT-LIST` from the previous section; and `NAT-LIST-ERROR` which updates `NAT-LIST` to change the operator declaration of `head` from `op head : NeList -> Nat` to `op head : List -> Nat`. Since the `NAT-LIST` specification contained an associative symbol, it fell outside the class known to be decidable. However, the CETA library is still able to show the automaton given by the

Module	$\|\mathcal{E}\|$	|ETA|	Time	Module	$\|\mathcal{E}\|$	|ETA|	Time
TRUTH-VALUE	3	22	0.33s	META-LEVEL	610	2011	2.52s
TRUTH	6	22	0.35s	*COUNTER*	56	206	0.44s
BOOL	19	60	0.38s	LOOP-MODE	116	439	0.69s
EXT-BOOL	25	74	0.38s	*CONFIGURATION*	18	105	0.35s
NAT	55	204	0.47s	*NAT-CONS*	33	135	0.37s
INT	96	262	0.55s	*MY-NAT-LIST*	30	109	0.36s
RAT	197	397	1.20s	NAT-LIST-FIX	33	114	0.45s
FLOAT	56	206	0.42s	*BLACKBOARD*	60	217	0.50s
STRING	74	288	0.57s	*CHESS-COVER*	80	308	0.53s
CONVERSION	262	677	1.35s	DIE-HARD	62	238	0.53s
RANDOM	56	208	0.45s	*JOSEPHUS*	63	245	0.51s
NAT-LIST	90	291	0.65s	*JOSEPHUS-GEN*	64	251	0.51s
QID-LIST	113	401	0.70s	*KHUN-PHAN*	66	258	0.46s
QID-SET	128	431	0.74s	*CHIPS*	70	273	0.45s
META-TERM	143	447	0.69s	RABBIT-HOP	68	254	0.54s
META-MODULE	499	1538	1.89s	CC-LOOP	1381	3837	>30m

Figure 5.1: SCC benchmarks

sufficient completeness analyzer was empty, and therefore the specification was sufficiently complete. The checker also finds the correct counterexample for NAT-LIST-ERROR.

5.5 Conclusions and future work

In this chapter, we have presented several contributions advancing methods for proving sufficient completeness to handle conditional specifications involving partial functions and where deduction is performed modulo axioms. Our main contributions include:

- We have studied the sufficient completeness of such specifications and their associated rewriting systems in greater generality than previous work, and arrived at a novel definition of sufficient completeness.
- We have extended the Maude Inductive Theorem Prover with a sufficient completeness checker for conditional specifications. This checker is not a decision procedure, but nevertheless we have been able to discharge sufficient completeness automatically for nontrivial examples by using the heuristics built into the ITP.
- Finally, we presented a sufficient completeness checker based on equational tree automata techniques that supports sufficient completeness checking with rewriting modulo any combination of associativity, commutativity, and identity.

Our work in developing sufficient completeness checkers for more complex equational specifications has already led to two complementary approaches, each able to handle specifications outside classes that could be handled by previous

```
Maude> in natlist.maude
=========================================
fmod NAT-LIST
=========================================
fmod NAT-LIST-ERROR
Maude> load scc.maude
Maude> loop init-scc .
Starting the Maude Sufficient Completeness Checker.
Maude> (scc NAT-LIST .)
Checking sufficient completeness of NAT-LIST ...
Success: NAT-LIST is sufficiently complete under the assumption that
    it is weakly-normalizing, ground confluent, and sort-preserving.
Maude> (scc NAT-LIST-ERROR .)
Checking sufficient completeness of NAT-LIST-ERROR ...
Failure: The term head(nil) is a counterexample as it is an
    irreducible term with sort Nat in NAT-LIST-ERROR that does not
    have sort Nat in the constructor subsignature.
```

Figure 5.2: Example SCC session

approaches. Although significant progress has been made, there is a great deal of opportunity both to develop new techniques and to improve the performance of existing techniques.

A number of further extensions of this work seem worth investigating. A first extension is to continue to push our characterization to consider checking specifications with context-sensitive rewriting (see the next chapter) and parametrized specifications. A second important topic is the generation of counterexamples to show lack of sufficient completeness: ground term counterexamples are practical and easy to generate, but investigating ways of symbolically describing sets of counterexamples may be quite useful for other purposes, such as generating induction schemes for theorem provers.

A third topic worth investigating is what we called the "second prong" in the introduction, namely, integrating sufficient completeness checking and inductive theorem proving in order to handle specifications outside the decidable subclasses. One recent approach [18] attempts to combine the tree automata and narrowing approaches with integration to a theorem prover to handle conditional, constrained rewrite specifications. This work is able to check the sufficient completeness of many-sorted rewrite specifications for which there may be conditional rules involving defined symbols and *constrained* rules involving constructor symbols. It accomplishes this by first constructing a constrained tree grammar to recognize irreducible constructor terms, and using the tree grammar during narrowing. Like the work in [73], this work targets the case of syntactic rewriting, but does not address specifications with rewriting modulo axioms.

A fourth topic that is important for scalability purposes is that of *modularity techniques* so that the sufficient completeness of a large equational specification

is not checked as a single monolithic module, but is ensured by checking a collection of submodules in which the specification is decomposed. This is quite natural in Maude, since large specifications are almost always obtained by composing many different modules together, and sufficient completeness techniques should be extended to exploit this additional composition information.

Further advances in these four areas should provide both foundations and algorithms in which to build a next-generation TA-based sufficient completeness tool for MEL specifications modulo axioms. This would make sufficient completeness checking available for a very wide class of specifications in Maude and other equational languages for specification and programming with advanced features.

Chapter 6

Completeness in context-sensitive rewriting

In previous chapters, we have discussed partiality and rewriting modulo axioms as important extensions that increase the expressitivity of rewrite specifications. Another important extension is that of user-programmable evaluation strategies based on *context-sensitive* (CS) rewriting (see, for example [101, 103, 145]). They allow very fine-grained control at the level of each individual function symbol on how the rewriting evaluation is performed. Their value and practical importance has been recognized in many equational languages. OBJ2 [52] was the first such language supporting them; and they are supported in all languages in the OBJ family, including CafeOBJ [51] and Maude [28]. In practice, CS rewriting can be used for two somewhat different purposes:

1. to increase the *efficiency* of a standard equational program without changing its meaning: for example, by restricting the evaluation of an if-then-else symbol to its first, Boolean argument to avoid wasteful or even non-terminating computations; and

2. as a way to compute with *infinite data structures* such as the infinite stream of all prime numbers, in a *lazy* way; in this second case, CS rewriting provides an elegant, finitary way of computing with infinite objects.

Expressiveness is substantially increased in both of these ways, since the user can *both* control the efficiency of program execution and support new applications involving infinite data structures.

This is all very well. However, there are a number of open research questions about how to *reason* formally about equational programs supporting CS rewriting for verification purposes. Two areas where important progress has been made are in methods for proving termination, e.g., [60, 102, 145] and confluence [103] of CS equational programs. But other important questions remain unexplored.

Imagine that you want to use an inductive theorem prover to verify some property about a CS equational program. No inductive theorem prover that we are aware of allows reasoning about CS programs. Is it ok to *ignore* the CS information and just reason about the underlying equational theory? We think that, in general, the answer is: *definitely not!* Why not? Because the *model*

on which the inductive reasoning principles are sound and the model of a CS program may be quite different.

What models are we talking about? Well, that is, one of the interesting research questions. For an inductive theorem prover, the model of interest is the initial algebra $T_{\mathcal{E}}$ of a specification $\mathcal{E}$. In fact, this *initial algebra semantics* is the standard mathematical semantics of equational programs in languages such as OBJ, CafeOBJ, and Maude. Furthermore, provided that the equational program is weakly normalizing and ground confluent, the initial algebra semantics fully *agrees* with the *operational semantics*, in the precise, mathematical sense that the initial algebra $T_{\mathcal{E}}$ and the *canonical term algebra* $\mathrm{Can}_{\mathcal{R}/A}$ of the rewrite system $\mathcal{R}/A$ associated to $\mathcal{E}$ are *isomorphic*. For CS rewriting the matter is less obvious, since we only have an operational semantics provided by the CS rewriting relation, but no mathematical models in the form of *algebras* have been put forward prior to our work. Therefore, the first contribution in this chapter is to put forward such an algebra, namely, the algebra $\mathrm{Can}^{\mu}_{\mathcal{R}/A}$ of *μ-canonical forms*, for μ the replacement map of the given CS program. We do so not just for vanilla-flavored, untyped CS programs, but for the more general and expressive CS programs with other features such as order-sorting and rewriting modulo axioms that one encounters in actual equational programming languages.

The importance of the algebra $\mathrm{Can}^{\mu}_{\mathcal{R}/A}$ is that it makes possible articulating and providing proof methods for three important CS *completeness problems*, namely:

1. *μ-canonical completeness*, which means satisfying the set-theoretic equality $\mathrm{Can}^{\mu}_{\mathcal{R}/A,s} = \mathrm{Can}_{\mathcal{R}/A,s}$ for each sort s in the specification;

2. *μ-semantic completeness*, which model-theoretically corresponds to the case where the surjective Σ-homomorphism $q : \mathrm{Can}^{\mu}_{\mathcal{R}/A} \rightarrow T_{\mathcal{E}}$, which we show always exists under minimal assumptions, is an *isomorphism*. Proof-theoretically this means that the *sound* way of proving ground E-equalities by CS rewriting is also *complete*, and that the Maude ITP is a suitable tool for reasoning about the context-sensitive specification;

3. *μ-sufficient completeness*, which is a new notion generalizing to the CS case the usual sufficient completeness of equational function definitions with respect to a signature of constructors. The subtlety here is that in general it would be *too strong* to require that constructors appear in all positions of a term t in μ-canonical form: we only make such a requirement for *replacing* positions in t.

We not only articulate these notions, but we also provide proof methods for them in the form of *decision procedures* under mild assumptions about the given CS program. Given that the CS programs we consider perform rewriting modulo axioms and are order-sorted, our methods are also based on *equational tree automata* described in Chapter 3 that can take into account both

sort information and reasoning modulo axioms. These decision procedures have been implemented in an extension of the tree automata-based Maude Sufficient Completeness Checker (SCC) discussed in Section 5.4, and we have used several Maude programs to illustrate both the basic ideas and the use of SCC in verifying CS completeness properties.

The chapter is organized as follows. In Section 6.1, we introduce the precise class of CS term rewrite systems we are considering. In Section 6.2, we define the canonical term algebra for a CS specification. In Section 6.3, we define the three notions of CS completeness, and in Section 6.4 we show how one can use equational tree automata techniques to check these completeness notions under appropriate assumptions. Finally, we discuss related work and suggest future avenues of research in Section 6.5. Much of this work has appeared previously in [70].

6.1 CS order-sorted term rewrite systems

We are interested in studying and analyzing context-sensitive rewriting for order-sorted term rewrite systems. In CS rewriting, there is a function $\mu : F \to \mathcal{P}(\mathbb{N})$, called the *replacement map*, which maps each function symbol $f \in F$ to a set of *replacing positions* $\mu(f) \subseteq \{1, \ldots, \text{arity}(f)\}$. The replacement map μ is used for restricting rewriting so that in rewriting a term $f(t_1, \ldots, t_n) \in T_\Sigma(X)$, the subterm t_i can only be rewritten if $i \in \mu(f)$. A *CS term rewrite system* is a pair $(\mathcal{R}/A, \mu)$ where μ is a replacement map for the signature of $\mathcal{R}/A$.

Given a replacement map μ, the set of positions that may be rewritten are called the *μ-replacing positions* and denoted by $\text{pos}^\mu(t)$. Formally, we have:

$$\text{pos}^\mu(x) = \{\epsilon\} \quad \text{and} \quad \text{pos}^\mu(f(t_1, \ldots, t_n)) = \{\epsilon\} \cup \bigcup_{i \in \mu(f)} \{iw \mid w \in \text{pos}^\mu(t_i)\}.$$

A context C is *μ-replacing* when the hole appears in a μ-replacing position.

We write $t \to_{\mathcal{R}/A,\mu} u$ if t rewrites to u using the rules in $\mathcal{R}/A$ and replacement map μ in a *single* rewrite step, i.e., there is a rule $l \to r$ in R such that $t =_A C[l\theta]$ and $u =_A C[r\theta]$ for some μ-replacing context C and substitution $\theta : X \to T_\Sigma(X)$. We let $\to^+_{\mathcal{R}/A,\mu}$ denote the transitive closure of $\to_{\mathcal{R}/A,\mu}$, and write $t \to^*_{\mathcal{R}/A,\mu} u$ iff $t \to^+_{\mathcal{R}/A,\mu} u$ or $t =_A u$. We write $t \downarrow^\mu_{\mathcal{R}/A} u$ if t and u can be rewritten to the same term, i.e., there is a term $v \in T_\Sigma(X)$ such that $t \to^*_{\mathcal{R}/A,\mu} v$ and $u \to^*_{\mathcal{R}/A,\mu} v$.

A term $t \in T_\Sigma(X)$ is *$(\mathcal{R}/A, \mu)$-reducible* iff there is a $u \in T_\Sigma(X)$ such that $t \to_{\mathcal{R}/A,\mu} u$, and *$(\mathcal{R}/A, \mu)$-irreducible* otherwise. We also say that a μ-irreducible term $t \in T_\Sigma(X)$ is in *μ-canonical* form. We write $t \to^!_{\mathcal{R}/A,\mu} u$ if $t \to^*_{\mathcal{R}/A,\mu} u$ and u is $(\mathcal{R}/A, \mu)$-irreducible. $\mathcal{R}/A$ is *μ-weakly normalizing* when for each term $t \in T_\Sigma(X)$ there is a term $u \in T_\Sigma(X)$ such that $t \to^!_{\mathcal{R}/A,\mu} u$. $\mathcal{R}/A$ is *μ-terminating* if the relation $\to_{\mathcal{R}/A,\mu}$ is Noetherian. $\mathcal{R}/A$ is *μ-confluent* if for

all $t, u, v \in T_\Sigma(X)$, $t \to^*_{\mathcal{R}/A,\mu} u$ and $t \to^*_{\mathcal{R}/A,\mu} v$ implies $u \downarrow^\mu_{\mathcal{R}/A} v$. $\mathcal{R}/A$ is μ-*sort-preserving* if for all terms $t \in T_\Sigma(X)_s$ and $u \in T_\Sigma(X)_{k_s}$, $t \to^*_{\mathcal{R}/A,\mu} u$ implies that there is a term $v \in T_\Sigma(X)_s$ such that $u \to^*_{\mathcal{R}/A,\mu} v$. When $\mathcal{R}/A$ is μ-weakly normalizing, μ-confluent, or μ-sort-preserving on ground terms, we say that it is ground μ-weakly normalizing, ground μ-confluent, or ground μ-sort-preserving, respectively.

When the replacement map μ allows rewriting at every subterm position, context sensitive rewriting specializes to rewriting in the ordinary sense. Let $\mu_\top$ be the replacement map $f \mapsto \{1, \ldots, \mathrm{arity}(f)\}$. It should be clear that $t \to_{\mathcal{R}/A} u$ iff $t \to^*_{\mathcal{R}/A,\mu_\top} u$, and $t \to^*_{\mathcal{R}/A} u$ iff $t \to^*_{\mathcal{R}/A,\mu_\top} u$. Additionally $\mathcal{R}/A$ is *weakly normalizing* iff it is $\mu_\top$-weakly normalizing. More generally, this convention extends to many other properties. For example, $\mathcal{R}/A$ is confluent iff it is $\mu_\top$-confluent.

6.2 CS canonical term algebras

When the Σ-rewrite system $\mathcal{R}/A$ is ground μ-weakly normalizing and ground μ-confluent, for each term $t \in T_\Sigma$, there is a $(\mathcal{R}/A, \mu)$-irreducible term, denoted by $t \downarrow^\mu_{\mathcal{R}/A}$ such that $t \to^!_{\mathcal{R}/A,\mu} t \downarrow^\mu_{\mathcal{R}/A}$, which is unique up to A. When $\mathcal{R}/A$ is additionally sort-preserving, we can then define a Σ-algebra of $(\mathcal{R}/A, \mu)$-canonical forms as follows:

Definition 6.2.1. *Let $\mathcal{R}/A$ be an order-sorted TRS with $\Sigma = (S, F, \leq)$ that is ground μ-weakly normalizing, ground μ-confluent and ground μ-sort-preserving. The* canonical term algebra *for $(\mathcal{R}/A, \mu)$ is the Σ-algebra $\mathrm{Can}^\mu_{\mathcal{R}/A}$ such that:*

- *for each sort $s \in S$, $\mathrm{Can}^\mu_{\mathcal{R}/A,s} = \{\, [t]_A \in T_{A,s} \mid t$ is $(\mathcal{R}/A, \mu)$-irreducible$\,\}$;*
- *for each $f \in F_{w,s}$, $\mathrm{Can}^\mu_{\mathcal{R}/A,f:w\to s}([t_1], \ldots, [t_n]) = [f(u_1, \ldots, u_n) \downarrow^\mu_{\mathcal{R}/A}]$ where $u_i \in [t_i] \cap T_{\Sigma,s_i}$ for $i \in [1, n]$.*

The algebra $\mathrm{Can}^\mu_{\mathcal{R}/A}$ has a strong computational meaning: it is exactly the algebra of *values* (μ-normal forms) that a user interacting with a system that evaluates $(\mathcal{R}/A, \mu)$ obtains by μ-reduction.[1] Therefore, it provides the perfect algebra for the *operational semantics* of $(\mathcal{R}/A, \mu)$. This model is in a sense situated *between* the term algebra T_Σ, and the model for the *mathematical semantics* of $\mathcal{R}/A$ as an equational theory, namely, the initial algebra $T_\mathcal{E}$ for the theory $\mathcal{E} = \mathcal{R} \cup A$. On the one hand, by initiality we have a unique homomorphism $\mathrm{Can}^\mu_{\mathcal{R}/A} : T_\Sigma \to \mathrm{Can}^\mu_{\mathcal{R}/A}$ which, as shown below, may not be surjective. On the other hand, $\mathrm{Can}^\mu_{\mathcal{R}/A}$ is *more concrete* than $T_\mathcal{E}$, and therefore a *sound*, but not necessarily complete, model for equational computation with $\mathcal{R}/A$. That is, we have:

[1] If $(\mathcal{R}/A, \mu)$ is μ-terminating, this is exactly true; if it is only μ-weakly normalizing, this requires either a μ-normalizing strategy, or the use of breadth-first search.

Proposition 6.2.2. *Given a TRS $\mathcal{R}/A$ over a signature $\Sigma = (S, F, \leq)$, let $\mathcal{E}$ denote the theory $\mathcal{E} = \mathcal{R} \cup A$.*

If $\mathcal{R}/A$ is ground μ-weakly normalizing, ground μ-confluent, and ground μ-sort-preserving, then the family of functions $\{q_s : \mathrm{Can}^{\mu}_{\mathcal{R}/A,s} \to T_{\mathcal{E},s}\}_{s \in S}$ with

$$q_s : [t]_A \mapsto [t]_{\mathcal{E}}$$

defines a surjective Σ-homomorphism $q : \mathrm{Can}^{\mu}_{\mathcal{R}/A} \to T_{\mathcal{E}}$.

Proof. The mapping q is a Σ-homomorphism, because for each $f \in F_{s_1 \ldots s_n, s}$ and $[t_1]_A \in \mathrm{Can}^{\mu}_{\mathcal{R}/A,s_1}, \ldots, [t_n]_A \in \mathrm{Can}^{\mu}_{\mathcal{R}/A,s_n}$, we can choose terms $u_i \in [t_i]_A \cap T_{\Sigma,s_i}$ for $i \in [1, n]$, and then we have,

$$T_{\mathcal{R} \cup A, f}(q([t_1]_A), \ldots, q([t_n]_A)) \\ = [f(u_1, \ldots, u_n)]_{\mathcal{E}} = q(\mathrm{Can}^{\mu}_{\mathcal{R}/A,f}([t_1]_A, \ldots, [t_n]_A))$$

as $\mathrm{Can}^{\mu}_{\mathcal{R}/A,f}([t_1]_A, \ldots, [t_n]_A) \subseteq [f(u_1, \ldots, u_n)]_{\mathcal{E}}$ by the soundness of $\to^*_{\mathcal{R}/A,\mu}$. □

One typically constructs ground terminating and confluent specifications in order to reason about the equivalence of two terms algebraically, and it is important to be able to reduce the equality problem $t =_{\mathcal{E}} u$ for the theory $\mathcal{E} = \mathcal{R} \cup A$ to the convergence problem $t \downarrow^{\mu}_{\mathcal{R}/A} u$. When considering ordinary (not context-sensitive) rewriting, we have $t =_{\mathcal{E}} u$ iff $t \downarrow_{\mathcal{R}/A} u$ iff $t \downarrow_{\mathcal{R}/A} =_A u \downarrow_{\mathcal{R}/A}$ for terms $t, u \in T_{\Sigma}$ when $\mathcal{R}$ is ground weakly normalizing, ground confluent, and ground sort-preserving. In this case, we are guaranteed that $\mathrm{Can}_{\mathcal{R}/A} = \mathrm{Can}^{\mu_\top}_{\mathcal{R}/A}$ is isomorphic to $T_{\mathcal{R} \cup A}$, thus obtaining a perfect agreement between the operational semantics of $\mathcal{R}/A$ and the mathematical, initial algebra semantics. In general, as we show below, this is *not* the case for CS rewriting, even if $\mathcal{R}/A$ is ground μ-terminating, ground μ-confluent, and ground μ-sort-preserving. That is $\mathrm{Can}^{\mu}_{\mathcal{R}/A}$ is *sound*, since $t \downarrow^{\mu}_{\mathcal{R}/A} u$ implies $t =_{\mathcal{E}} u$, but in general is *not complete*, i.e., $t =_{\mathcal{E}} u \not\Rightarrow t \downarrow^{\mu}_{\mathcal{R}/A} u$.

Consider the TRS $\mathcal{R}/A$ with single sort s, symbols $a : \to s$, $b : \to s$, and $f : s \to s$, and replacement map μ where $\mu(f) = \varnothing$ with the rules: $a \to f(a)$ and $b \to f(a)$. This specification is clearly μ-weakly normalizing, μ-confluent, and μ-sort-preserving. However, $T_{\mathcal{R} \cup A, s} = \{\, [a]_{\mathcal{R} \cup A} \}$ whereas $\mathrm{Can}^{\mu}_{\mathcal{R}/A,s}$ is the infinite set $\{\{f(a)\}, \{f(b)\}, \{f(f(a))\}, \{f(f(b))\}, \ldots\}$.

The algebra $\mathrm{Can}^{\mu}_{\mathcal{R}/A}$ differs from $\mathrm{Can}_{\mathcal{R}/A}$ in several other properties as well. In general, it is not the case that $\mathrm{Can}^{\mu}_{\mathcal{R}/A}(t) = t \downarrow^{\mu}_{\mathcal{R}/A}$. In the specification above, $\mathrm{Can}^{\mu}_{\mathcal{R}/A}(f(a)) = f(\mathrm{Can}^{\mu}_{\mathcal{R}/A}(a)) \downarrow^{\mu}_{\mathcal{R}/A} = f(f(a))$, whereas $f(a) \downarrow^{\mu}_{\mathcal{R}/A} = f(a)$. Additionally, the unique homomorphism $\mathrm{Can}^{\mu}_{\mathcal{R}/A} : T_A \to \mathrm{Can}^{\mu}_{\mathcal{R}/A}$ is neither surjective nor idempotent. For example, there is no term $t \in T_{\Sigma}$, such that $\mathrm{Can}^{\mu}_{\mathcal{R}/A}(\{t\}) = \{f(b)\}$, while $\mathrm{Can}^{\mu}_{\mathcal{R}/A}(\{a\}) = \{f(a)\}$ and $\mathrm{Can}^{\mu}_{\mathcal{R}/A}(\{f(a)\}) = \{f(f(a))\}$.

6.3 Completeness in context-sensitive rewriting

We have now shown that the usual requirements of μ-termination, μ-confluence, and μ-sort-preservation are insufficient to guarantee that the operational semantics of CS term rewriting corresponds to the mathematical semantics of the equational specification. One of the goals of this section is to investigate which additional conditions we need to impose to guarantee that CS rewriting serves as a sound and complete technique to deduce ground equalities, i.e., when is the canonical term algebra $\mathrm{Can}^{\mu}_{\mathcal{R}/A}$ isomorphic to the initial algebra $T_{\mathcal{R}\cup A}$.

In this section, we introduce three notions of completeness for CS term rewrite systems $(\mathcal{R}/A, \mu)$: (1) μ-canonical completeness; (2) μ-semantic completeness; and (3) μ-sufficient completeness. The first two notions of completeness are used to characterize the deductive power of CS rewriting. The third is used to analyze specifications that may not be complete in the first two senses, but may nevertheless represent useful applications of CS rewriting, such as specifying infinite data-structures. Later, in Section 6.4, we will show how these three completeness properties can be checked for specifications satisfying appropriate requirements such as left-linearity, ground μ-weak normalization, ground μ-confluence, and ground μ-sort-preservation.

6.3.1 Canonical completeness

The first property we consider is whether the canonical forms of CS rewriting and ordinary rewriting agree:

Definition 6.3.1. *A TRS $\mathcal{R}/A$ over a signature Σ is* μ-canonically complete *if every $(\mathcal{R}/A, \mu)$-irreducible term $t \in T_\Sigma$ is $\mathcal{R}/A$-irreducible.*

The theorem below shows that, for specifications that are ground μ-weakly normalizing and ground confluent, canonical completeness is enough to imply that CS and ordinary rewriting agree on convergence relations.

Theorem 6.3.2. *If a TRS $\mathcal{R}/A$ over Σ is ground μ-weakly normalizing, μ-canonically complete, and ground confluent, then for $t, u \in T_\Sigma$, $t \downarrow_{\mathcal{R}/A} u$ iff $t \downarrow^{\mu}_{\mathcal{R}/A} u$.*

Proof. We trivially have that $t \downarrow^{\mu}_{\mathcal{R}/A} u \Rightarrow t \downarrow_{\mathcal{R}/A} u$. To see $t \downarrow_{\mathcal{R}/A} u \Rightarrow t \downarrow^{\mu}_{\mathcal{R}/A} u$, since $\mathcal{R}/A$ is μ-weakly normalizing, there must be $(\mathcal{R}/A, \mu)$-irreducible terms $t', u' \in T_\Sigma$ such that $t \to^{!}_{\mathcal{R}/A,\mu} t'$ and $u \to^{!}_{\mathcal{R}/A,\mu} u'$. $\mathcal{R}/A$ is μ-canonically complete, so both t' and u' must be $\mathcal{R}/A$-irreducible as well. This implies that $t' =_A u'$ by the ground confluence of $\mathcal{R}/A$. □

As a corollary, we observe that this class of specifications is μ-confluent.

Corollary 6.3.3. *If $\mathcal{R}/A$ is ground μ-weakly normalizing, μ-canonically complete, and ground confluent, then $\mathcal{R}/A$ is ground μ-confluent.*

Proof. If $t \to^*_{\mathcal{R}/A,\mu} u$ and $t \to^*_{\mathcal{R}/A,\mu} v$, we have $u \downarrow_{\mathcal{R}/A} v$ by the ground confluence of $\mathcal{R}/A$. Thus, $u \downarrow^{\mu}_{\mathcal{R}/A} v$ by Theorem 6.3.2. □

In a similar vein, we can show ground μ-sort-preservation of $\mathcal{R}/A$ by showing that $\mathcal{R}/A$ is ground μ-weakly normalizing, μ-canonically complete, and sort-preserving.

Theorem 6.3.4. *If $\mathcal{R}/A$ is ground μ-weakly normalizing, μ-canonically complete, and ground sort-preserving, then $\mathcal{R}/A$ is ground μ-sort-preserving.*

Proof. Assume $t \in T_{\Sigma,s}$ and $t \to^*_{\mathcal{R}/A,\mu} u$ with $u \notin T_{\Sigma,s}$. Since $\mathcal{R}/A$ is ground μ-weakly normalizing and μ-canonically complete, there is an $\mathcal{R}/A$-irreducible term $u' \in T_\Sigma$ such that $u \to^*_{\mathcal{R}/A,\mu} u'$. Because $\mathcal{R}/A$ is ground sort-preserving, there must be a term $v \in T_{\Sigma,s}$ such that $u' \to^*_{\mathcal{R}/A} v$. However, u' is $\mathcal{R}/A$-irreducible, so $u' =_A v$. It follows that $u \to^*_{\mathcal{R}/A,\mu} v$, and $\mathcal{R}/A$ is ground μ-sort-preserving. □

Together, Corollary 6.3.3 and Theorem 6.3.4 provide a means to check μ-confluence and μ-sort-preservation for μ-weakly normalizing, μ-canonically complete, confluent, and sort-preserving specifications. Since one can prove μ-termination with existing tools [43, 61, 104], and check μ-canonical completeness of left-linear specifications with the decision procedure in Section 6.4.1, this eliminates the need for specialized CS-aware checking procedures for this class of specifications. The case of ground weak normalization and ground μ-weak normalization for μ-canonically complete specification yields a relation in the other direction.

Theorem 6.3.5. *If $\mathcal{R}/A$ is ground μ-weakly normalizing and μ-canonically complete, then $\mathcal{R}/A$ is ground weakly normalizing.*

Proof. If $\mathcal{R}/A$ is ground μ-weakly normalizing, then for each term $t \in T_\Sigma$, there is a $(\mathcal{R}/A, \mu)$-irreducible term $u \in T_\Sigma$ such that $t \to^!_{\mathcal{R}/A,\mu} u$. Since $\mathcal{R}/A$ is μ-canonically complete it follows that u is $\mathcal{R}/A$-irreducible as well, so $t \to^!_{\mathcal{R}/A} u$. □

On the other hand, if $\mathcal{R}/A$ is ground weakly normalizing and μ-canonically complete, it may not be ground μ-weakly normalizing. Let $\mathcal{E} = \varnothing$, and $\mathcal{R}/A$ have the rules $f(x) \to f(x)$, $a \to b$, and $f(b) \to b$. $\mathcal{R}/A$ is ground weakly normalizing, because every term can reduce to the $\mathcal{R}/A$-irreducible term b. Given the replacement map μ with $\mu(f) = \varnothing$, $\mathcal{R}/A$ is μ-canonically complete, because b is the only $(\mathcal{R}/A, \mu)$-irreducible term as well. However, $\mathcal{R}/A$ is not μ-weakly normalizing, because $f(a) \not\to^*_{\mathcal{R}/A,\mu} b$.

As an example of a μ-canonically complete specification, we present a Maude module in Figure 6.1 for computing the factorial of a natural number. This specification protects the built-in `NAT` specification, which contains constructor operators `0` and `s` for for zero and successor respectively, along with defined

```
fmod FACTORIAL is protecting NAT .
  var X Y Z : Nat .
  op p : Nat -> Nat .
  eq p(s(X)) = X .
  eq p(0) = 0 .
  op if0 : Nat Nat Nat -> Nat [strat(1 0)].
  eq if0(0, Y, Z) = Y .  eq if0(s(X), Y, Z) = Z .
  op fact : Nat -> Nat .
  eq fact(X) = if0(X, s(0), X * fact(p(X))) .
endfm
```

Figure 6.1: Factorial example

operators for plus and times. Predecessor `p` is defined as usual, and the operator `if0` is annotated with a strategy `strat(1 0)`, indicating that only the first argument should be evaluated. Since the other operators are not given a strategy, Maude uses its default strategy, which evaluates every argument. In effect, these declarations define a replacement map μ where $\mu(\texttt{if0}) = \{1\}$ and $\mu(f) = \{1, \ldots, \text{arity}(f)\}$ for $f \neq \texttt{if0}$. Using `if0`, factorial can be defined with a single equation.

Without the strategy declaration on `if0`, this specification is not terminating, and evaluating `fact(0)` quickly leads to a segmentation fault in the Maude interpreter. However, with the given replacement map μ, the specification is μ-terminating. Moreover, it is μ-canonically complete, ground μ-confluent, and ground μ-sort-preserving. Since there is only one sort, μ-sort-preservation is obvious. As the specification is left-linear, the decision procedure we introduce in Section 6.4.1 will allow us to automatically check its μ-canonically completeness. To see that it is ground μ-confluent one can just observe that it is confluent (indeed, orthogonal), and use Corollary 6.3.3.

6.3.2 Semantic completeness

Canonical completeness means that $\text{Can}_{\mathcal{R}/A,s} = \text{Can}^{\mu}_{\mathcal{R}/A,s}$ for each sort $s \in S$. By itself, this is not enough to immediately imply that $\text{Can}^{\mu}_{\mathcal{R}/A}$ and $T_{\mathcal{R}\cup A}$ are isomorphic. This is implied by another notion of completeness, called *μ-semantic completeness*, which we define below.

Definition 6.3.6. *A TRS $\mathcal{R}/A$ over Σ is* μ-semantically complete *iff for all $t, u \in T_\Sigma$, $t \downarrow^{\mu}_{\mathcal{R}/A} u$ iff $t =_{\mathcal{R}\cup A} u$.*

This definition at the syntactic level of terms captures the agreement between operational semantics and mathematical semantics that we want when the canonical algebra $\text{Can}^{\mu}_{\mathcal{R}/A}$ is well-defined.

Theorem 6.3.7. *If $\mathcal{R}/A$ is ground μ-weakly normalizing, ground μ-confluent, and ground μ-sort-preserving, then $\mathcal{R}/A$ is μ-semantically complete iff* $\text{Can}^{\mu}_{\mathcal{R}/A}$

is isomorphic to $T_{\mathcal{R}\cup A}$.

Proof. Let $\Sigma = (S, F, \leq)$ denote the signature for $\mathcal{R}/A$, and let $\mathcal{E} = \mathcal{R} \cup A$. If $\mathrm{Can}^{\mu}_{\mathcal{R}/A}$ is isomorphic to $T_{\mathcal{E}}$, they must satisfy the same equations, so $\mathcal{R}/A$ is μ-semantically complete. On the other hand, the equivalence

$$t =_{\mathcal{E}} u \iff t \downarrow^{\mu}_{\mathcal{R}/A} =_A u \downarrow^{\mu}_{\mathcal{R}/A}$$

implies that the surjective map $q_s : [t]_A \mapsto [t]_{\mathcal{E}}$ is injective for each $s \in S$, and therefore bijective. To see that $q : \mathrm{Can}^{\mu}_{\mathcal{R}/A} \to T_{\mathcal{E}}$ is an Σ-isomorphism, we should check that q^{-1} is also a Σ-homomorphism. Consider $[t]_{\mathcal{E}}$, then $q^{-1}([t]_{\mathcal{E}}) = [t \downarrow^{\mu}_{\mathcal{R}/A}]$, and for any operator $f : s_1 \dots s_n \to s$ in Σ, given $[t_i] \in T_{\mathcal{E},s_i}$ for $i \in [1, n]$, we can choose a term $t_i \downarrow^{\mu}_{\mathcal{R}/A} \in [t_i]_{\mathcal{E}} \cap T_{\Sigma,s_i}$ for $i \in [1, n]$, so that

$$\begin{aligned} &q^{-1}(T_{\mathcal{E},f}([t_1], \dots, [t_n])) = q^{-1}\left([f(t_1 \downarrow^{\mu}_{\mathcal{R}/A}, \dots, t_n \downarrow^{\mu}_{\mathcal{R}/A})]_{\mathcal{E}}\right) \\ &= [f(t_1 \downarrow^{\mu}_{\mathcal{R}/A}, \dots, t_n \downarrow^{\mu}_{\mathcal{R}/A}) \downarrow^{\mu}_{\mathcal{R}/A}]_A = \mathrm{Can}^{\mu}_{\mathcal{R}/A,f}\left(q^{-1}([t_1]), \dots, q^{-1}([t_n])\right). \end{aligned}$$

□

The next question that we address is how to check that a specification is μ-semantically complete. The results in the previous section on μ-canonical completeness lead to the following result:

Theorem 6.3.8. *A TRS $\mathcal{R}/A$ that is ground μ-weakly normalizing, μ-canonically complete, ground confluent, and ground sort-preserving is μ-semantically complete.*

Proof. Let $\mathcal{E} = \mathcal{R} \cup A$. By Theorem 6.3.5, $\mathcal{R}/A$ is ground weakly-normalizing. It follows that $t =_{\mathcal{E}} u$ iff $t \downarrow_{\mathcal{R}/A} u$ since $\mathcal{R}/A$ is ground confluent and sort-preserving as well. By Theorem 6.3.2, we have that $t \downarrow_{\mathcal{R}/A} u$ iff $t \downarrow^{\mu}_{\mathcal{R}/A} u$. So $\mathcal{R}/A$ is μ-semantically complete. □

As a corollary, we can easily obtain checkable conditions under which all three of the algebras $\mathrm{Can}^{\mu}_{\mathcal{R}/A}$, $\mathrm{Can}_{\mathcal{R}/A}$ and $T_{\mathcal{R}\cup A}$ are isomorphic.

Corollary 6.3.9. *If $\mathcal{R}/A$ is ground μ-weakly normalizing, μ-canonically complete, confluent, and sort-preserving, then* $\mathrm{Can}^{\mu}_{\mathcal{R}/A}$ *and* $\mathrm{Can}_{\mathcal{R}/A}$ *are both well-defined and isomorphic to $T_{\mathcal{R}\cup A}$.*

Proof. We know that $\mathrm{Can}_{\mathcal{R}/A}$ is well-defined and isomorphic to $T_{\mathcal{R}\cup A}$ since $\mathcal{R}/A$ is weakly normalizing by Theorem 6.3.5. We also know by Theorems 6.3.8 and 6.3.7, that $\mathrm{Can}^{\mu}_{\mathcal{R}/A}$ is isomorphic to $T_{\mathcal{R}\cup A}$. Therefore, all three algebras are isomorphic. □

When the TRS $\mathcal{R}/A$ is ground μ-weakly normalizing, ground confluent, and ground sort-preserving, μ-canonical completeness is a *sufficient* condition to show μ-semantic completeness, but it turns out not to be a *necessary* condition.

For example, let $\mathcal{R}$ have the rules: $f(f(x)) \to f(x)$, $a \to b$, and $f(b) \to f(a)$, and let μ be the replacement map with $\mu(f) = \varnothing$. The initial algebra contains two equivalence classes: one with the constants a and b, the other with terms containing f. The $(\mathcal{R}, \mu)$-canonical terms are b and $f(a)$, and it is easy to show that $\mathrm{Can}^{\mu}_{\mathcal{R}/A}$ and $T_{\mathcal{R} \cup A}$ are isomorphic. Since $\mathcal{R}$ is also ground μ-weakly normalizing, ground μ-confluent and ground μ-sort decreasing, $\mathcal{R}$ is μ-semantically complete by Theorem 6.3.7. However, $f(a)$ is $\mathcal{R}$-reducible, leaving b the only $\mathcal{R}$-irreducible term, and so $\mathcal{R}$ is not μ-canonically complete. In addition to not being μ-canonically complete, $\mathcal{R}$ is not ground weakly normalizing. However, if $\mathcal{R}$ is ground weakly normalizing, μ-semantic completeness implies μ-canonical completeness.

Theorem 6.3.10. *If $\mathcal{R}/A$ is ground weakly normalizing and μ-semantically complete, then $\mathcal{R}/A$ is μ-canonically complete.*

Proof. Let $\mathcal{E} = \mathcal{R} \cup A$, and let $t \in T_\Sigma$ be an $(\mathcal{R}/A, \mu)$-irreducible term. As $\mathcal{R}/A$ is weakly-normalizing, there is a $\mathcal{R}/A$-irreducible term $u \in T_\Sigma$ such that $t \to^*_{\mathcal{R}/A} u$. We know that $t =_{\mathcal{E}} u$ by the soundness of rewriting, and since $\mathcal{R}/A$ is μ-semantically complete, we have $t \downarrow^{\mu}_{\mathcal{R}/A} u$. However, both t and u are $(\mathcal{R}/A, \mu)$-irreducible, so $t =_A u$. Since u is $\mathcal{R}/A$-irreducible as well, it follows that t is $\mathcal{R}$-irreducible. □

In other words, if $\mathcal{R}/A$ is ground weakly normalizing and not μ-canonically complete, it is not μ-semantically complete either.

6.3.3 Context-sensitive sufficient completeness

Although μ-canonical completeness and μ-semantic completeness are useful notions of completeness in CS rewriting, there are many interesting applications of CS rewriting, especially those involving infinite data structures, that are not μ-semantically complete. As an example, we present a typed version of a specification of infinite lists from [101] in Figure 6.2. In that specification, the term `from(M)` represents the infinite list "`M : M + 1 : ...`", and there are functions for obtaining the ith element in a list and the first n elements in the list. This specification is an interesting use of CS rewriting to obtain a terminating method to execute a non-terminating rewrite system. Although the equation for `from` is non-terminating, it is μ-terminating because of the strategy on "`:`".

The specification `INF-LIST` is not μ-canonically complete, and its canonical algebra is not isomorphic to the initial algebra of the equational theory given by its axioms. For example, the terms $\mathtt{0} : \mathtt{from}(\mathtt{s}(\mathtt{0}))$ and $\mathtt{0} : \mathtt{s}(\mathtt{0}) : \mathtt{from}(\mathtt{s}(\mathtt{s}(\mathtt{0})))$ are distinct μ-canonical terms, however

$$\mathtt{0} : \mathtt{from}(\mathtt{s}(\mathtt{0})) =_{\mathtt{INF-LIST}} \mathtt{0} : \mathtt{s}(\mathtt{0}) : \mathtt{from}(\mathtt{s}(\mathtt{s}(\mathtt{0}))).$$

In order to check properties of specifications like `INF-LIST` that are not μ-semantically complete, we therefore need techniques that analyze CS specifica-

```
fmod INF-LIST is protecting NAT .
  sorts Nat? List .
  subsort Nat < Nat? .
  op none : -> Nat? [ctor].
  op [] : -> List [ctor].
  op _:_ : Nat List -> List [ctor strat(1 0)].
  vars M N : Nat . var L : List .

  op sel : Nat List -> Nat? .
  eq sel(0, N : L) = N .
  eq sel(s(M), N : L) = sel(M, L) .

  op from : Nat -> List .
  eq from(M) = M : from(s(M)) .

  op first : Nat List -> List .
  eq first(0, L) = [] .
  eq first(s(M), N : L) = N : first(M, L) .
endfm
```

Figure 6.2: Infinite lists example

tions directly. The case of μ-termination is well understood when $A = \varnothing$ or contains only AC and free symbols [60, 102, 145]. The case of μ-confluence has already been studied in [103].

Another interesting property that seems not to have been studied for CS specifications is sufficient completeness. Sufficient completeness in term rewriting specifications means that enough equations have been defined so that all terms reduce to constructor terms. For example, a sufficiently complete specification involving arithmetic over the natural numbers should reduce every term containing plus and times to a term containing only zero and successor.

Although simple, this definition of sufficient completeness seems *too strong* in the context of CS specifications. The reason is that the non-replacing positions of a symbol intentionally do not reduce their arguments. Accordingly, our definition of μ-sufficient completeness allows defined symbols in the non-replacing positions of canonical terms, provided that all *replacing* positions have constructor symbols.

Definition 6.3.11. *Let $\mathcal{R}/A$ be a ground μ-weakly normalizing and ground μ-sort-preserving TRS over a signature $\Sigma = (S, F, \leq)$ equipped with an indexed family of constructor symbols $C = \{C_{w,s}\}_{(w,s)\in S^*\times S}$ with each $C_{w,s} \subseteq F_{w,s}$. We say that $\mathcal{R}/A$ is* μ-sufficiently complete relative to C *iff for all $(\mathcal{R}/A, \mu)$-irreducible terms $t \in T_\Sigma$, $\mathrm{pos}^\mu(t) \subseteq \mathrm{pos}_C(t)$ where*

$$\mathrm{pos}_C(t) = \{\, i \in \mathrm{pos}(t) \mid t_{|i} = c(\overline{t}) \wedge c \in C_{w,s} \wedge \overline{t} \in T_{\Sigma,w} \,\}.$$

Our definition of μ-sufficient completeness reduces to the usual definition of

ground reducibility when every position is a replacing position, i.e., $\mu = \mu_\top$.

Theorem 6.3.12. *If $\mathcal{R}/A$ is ground μ-weakly normalizing, μ-canonically complete, and ground sort-preserving, then $\mathcal{R}/A$ is μ-sufficiently complete relative to C iff it is sufficiently complete relative to C.*

Proof. First, it should be noted that our definition of μ-sufficiently completeness only applies if $\mathcal{R}/A$ is μ-sort decreasing. However, given the conditions, we know that it is μ-sort-preserving by Theorem 6.3.4.

If $\mathcal{R}/A$ is μ-canonically complete, then $\mathcal{R}/A$ and $(\mathcal{R}/A, \mu)$ have the same canonical forms. It follows then that if $\mathcal{R}/A$ is sufficiently complete relative to C, $\mathcal{R}/A$ must be μ-sufficiently complete. On the other hand, suppose that $\mathcal{R}/A$ is μ-sufficiently complete relative to C. The characterization of sufficient completeness given by Theorem 5 of [74] shows that $\mathcal{R}/A$ is sufficiently complete iff each term $t \in T_{\Sigma,s}$ with a defined symbol at the root and constructor subterms is $\mathcal{R}/A$-reducible. However, since $\mathcal{R}/A$ is μ-sufficiently complete and the root must be a replacing position, we know that each $(\mathcal{R}/A, \mu)$-irreducible term $t \in T_\Sigma$ must have a constructor at the root. So any term with a defined symbol at the root must be $(\mathcal{R}/A, \mu)$-reducible, and consequently, $\mathcal{R}/A$-reducible due to $\mathcal{R}/A$ being μ-canonically complete. □

6.4 Checking completeness properties

In the left-linear case, we are able to reduce the μ-canonical completeness and μ-sufficient completeness properties to the *propositional emptiness problem* for equational tree automata. We are further able to use the results of Theorem 6.3.8 to have sufficient conditions for showing the μ-semantic completeness of $\mathcal{R}$ when $\mathcal{R}$ is left-linear, μ-weakly normalizing, μ-canonically complete, ground confluent, and ground sort-preserving. We can reduce the CS completeness properties for a TRS $\mathcal{R}/A$ into a propositional emptiness test for an equational tree automaton over the unsorted theory $\overline{A}$. When A consists of any combination of associativity, commutativity, and identity axioms, we can use the techniques described in Chapter 3 to check the corresponding propositional emptiness problem.

6.4.1 Checking canonical completeness

From the definition of μ-canonical completeness, we know that $\mathcal{R}/A$ is not μ-canonically complete iff there is a term $t \in T_\Sigma$ that is $\mathcal{R}/A$-reducible and $(\mathcal{R}/A, \mu)$-irreducible. Therefore, we can reduce the μ-canonical completeness problem to a propositional emptiness problem of an automaton $\mathcal{A}$ by defining a language containing precisely equivalence classes $[t] \in T_\mathcal{E}$ that are counterexamples.

In order to recognize ground terms that are $\mathcal{R}/A$ and $(\mathcal{R}/A, \mu)$-reducible in a tree automata framework, we will additionally need to recognize terms matching subterms appearing in the left-hand side of rules in $\mathcal{R}$.

Definition 6.4.1. *Given a set of rules $\mathcal{R}$ over a signature Σ, the* intermediate terms $I_{\mathcal{R}}$ *denote the set of quantified non-variable strict subterms appearing in the left-hand side of a rule in $\mathcal{R}$ along with the sort-constraints on the variables in the terms, i.e.,*

$$I_{\mathcal{R}} = \{t \mid C[t] \in \mathrm{lhs}(\mathcal{R}) \wedge t \notin X \wedge C \neq \Box\}.$$

where $\mathrm{lhs}(\mathcal{R})$ *denotes the left-hand sides of the rules in $\mathcal{R}$, i.e.,* $\mathrm{lhs}(\mathcal{R}) = \{l \mid l \to r \in \mathcal{R}\}$.

Theorem 6.4.2. *Given a left-linear TRS $\mathcal{R}/A$, one can effectively construct an automaton $\mathcal{A}_{\mathsf{cc}}$ and formula ϕ over the states in $\mathcal{A}_{\mathsf{cc}}$ such that $\mathcal{R}$ is μ-canonically complete iff $\mathcal{L}_\phi(\mathcal{A}_{\mathsf{cc}}/\overline{A}) = \varnothing$.*

Proof. Let $\Sigma = (S, F, \leq)$ denote the signature of $\mathcal{R}/A$. We define $\mathcal{A}_{\mathsf{cc}}$ so that its states recognize well-sorted terms in T_A, terms that match subterms in $I_{\mathcal{R}}$ modulo A, and $\mathcal{R}/A$-reducible and $\mathcal{R}/A$-reducible terms. Formally, $\mathcal{A}_{\mathsf{cc}}$ is an automaton over the signature $\overline{\Sigma}$ and states Q where

$$Q = \{\, r, r^\mu, q_\top \,\} \cup \{\, q_s \mid s \in S \,\} \cup \{\, q_u \mid u \in I_{\mathcal{R}} \,\}.$$

To simplify later notation, for each variable $x_s \in X$, we let $q_{x_s} = q_s$. The automaton $\mathcal{A}_{\mathsf{cc}}$ is then defined as follows.

- For each declaration $f : s_1 \dots s_n \to s$, $\mathcal{A}_{\mathsf{cc}}$ contains
$$q_s(f(x_1, \dots, x_n)) \Leftarrow q_{s_1}(x_1), \dots, q_{s_n}(x_n).$$
- For distinct sorts $s, s' \in S$ such that $s < s'$, $\mathcal{A}_{\mathsf{cc}}$ contains $q_{s'}(x) \Leftarrow q_s(x)$.
- For each sort $s \in S$, $\mathcal{A}_{\mathsf{cc}}$ contains $q_\top(x) \Leftarrow q_s(x)$.
- For each term $f(t_1, \dots, t_n) \in I_{\mathcal{R}}$, $\mathcal{A}_{\mathsf{cc}}$ contains
$$q_{f(t_1,\dots,t_n)}(f(x_1, \dots, x_n)) \Leftarrow q_{t_1}(x_1), \dots, q_{t_n}(x_n).$$
- For each left-hand side $f(t_1, \dots, t_n) \in \mathrm{lhs}(R)$, $\mathcal{A}_{\mathsf{cc}}$ contains
$$r^\mu(f(x_1, \dots, x_n)) \Leftarrow q_{t_1}(x_1), \dots, q_{t_n}(x_n).$$
- For each operator $f \in F$ with arity n and index $i \in \mu(f)$, $\mathcal{A}_{\mathsf{cc}}$ contains
$$r^\mu(f(x_1, \dots, x_n)) \Leftarrow q_\top(x_1), \dots, r^\mu(x_i), \dots, q_\top(x_n).$$
- $\mathcal{A}_{\mathsf{cc}}$ contains the rule $r(x) \Leftarrow r^\mu(x)$.
- Finally, for each operator $f \in F$ with arity n and index $i \in [1, n]$, $\mathcal{A}_{\mathsf{cc}}$ contains $r(f(x_1, \dots, x_n)) \Leftarrow q_\top(x_1), \dots, r(x_i), \dots, q_\top(x_n)$.

By induction on $t \in T_{\overline{\Sigma}}$, we have $t \in \mathcal{L}_{q_s}(\mathcal{A}_{\mathsf{cc}})$ iff $t \in T_{\Sigma,s}$. A similar inductive argument allows us to show that $t \in \mathcal{L}_{q_u}(\mathcal{A}_{\mathsf{cc}})$ iff there is a substitution θ such that $t = u\theta$. In turn, this allows us to show that $t \in \mathcal{L}_{r^\mu}(\mathcal{A}_{\mathsf{cc}})$ iff there is a μ-replacing context C, substitution θ, and rule $l \to r$ in $\mathcal{R}$ such that $t = C[l\theta]$, and to show that $t \in \mathcal{L}_r(\mathcal{A}_{\mathsf{cc}})$ iff there is a context C, substitution θ, and rule $l \to r$ in R such that $t = C[l\theta]$.

To reduce μ-canonical completeness to a propositional emptiness problem we let ϕ be the formula $\phi = q_\top \wedge r \wedge \neg r^\mu$. Since A is sort-independent, it follows from Theorem 3.1.1 that $\mathcal{L}_\phi(\mathcal{A}_{\mathsf{cc}}/\overline{A})$ contains an equivalence classes with a well-sorted term that is $\mathcal{R}/A$-reducible and $(\mathcal{R}/A, \mu)$-irreducible. Thus $\mathcal{L}_\phi(\mathcal{A}_{\mathsf{cc}}/\overline{A}) = \varnothing$ iff $\mathcal{R}/A$ is μ-canonically complete. □

The algorithm for constructing the tree automaton $\mathcal{A}_{\mathsf{cc}}$ from a Maude specification has been implemented, and integrated into the Maude Sufficient Completeness Checker presented in Section 5.4. By using the tool to check the μ-canonical completeness of the `FACTORIAL` specification given in Section 6.3.1, we are able to verify that it is μ-canonically complete:

```
Maude> (ccc FACTORIAL .)
Checking canonical completeness of FACTORIAL ...
Success: FACTORIAL is canonically complete.
```

By using the tool to check the `INF-LIST` specification, we find a counterexample showing that the specification is not μ-canonically complete:

```
Maude> (ccc INF-LIST .)
Checking canonical completeness of INF-LIST ...
Failure: The term 0 : first(0,[]) is a counterexample that is
    mu-irreducible, but reducible under ordinary rewriting.
```

6.4.2 Checking semantic completeness

Since we were able to check the μ-canonical completeness of a left-linear specification $\mathcal{R}$ using the results in the previous section, using the results in Theorem 6.3.8, the μ-semantic completeness of specifications can be mechanically checked by showing: (1) μ-canonical completeness with the checker in the previous section; (2) μ-terminating with a CS termination tool such as [43, 61, 104]; and (3) confluence and sort-preservation with a tool such as the Maude Church-Rosser checker. This allows us to show that, for example, the `FACTORIAL` specification is μ-semantically complete.

6.4.3 Checking sufficient completeness

Using our definition of μ-sufficient completeness, we are able to extend the Maude Sufficient Completeness Checker in Section 5.4 to the CS case.

Theorem 6.4.3. *Given a left-linear TRS $\mathcal{R}/A$ that is ground μ-weakly normalizing and ground μ-sort-preserving, one can construct an automaton $\mathcal{A}_{\mathsf{SC}}$ and formula ϕ such that $\mathcal{R}/A$ is μ-sufficiently complete relative to constructors C iff $\mathcal{L}_\phi(\mathcal{A}_{\mathsf{SC}}/\overline{A}) = \varnothing$.*

Proof. Let $\Sigma = (S, F, \leq)$ denote the signature of $\mathcal{R}/A$.

The language $\mathcal{L}_\phi(\mathcal{A}_{\mathsf{SC}}/\overline{A})$ is defined so that it accepts the equivalence class $[t]_{\overline{A}}$ of a a $(\mathcal{R}, \mu)$-irreducible term $t \in T_\Sigma$ with a replacement position $i \in \mathrm{pos}^\mu(t)$ that is not in $\mathrm{pos}_C(t)$. The states Q of the automaton $\mathcal{A}_{\mathsf{SC}}$ is the set

$$Q = \{\, r^\mu, q_\top \,\} \cup \{\, q_s, c_s \mid s \in S \,\} \cup \{\, q_u \mid u \in I_\mathcal{R} \,\}.$$

As before, to simplify later notation for each variable $x_s \in X$, we let $q_{x_s} = q_s$. Additionally, for each constructor symbol $c \in C$ with arity n and each index $i \in [1, n]$, we define the function $\mathrm{rep}^\mu_{c,i} : S \to Q$ as the function so that

$$\mathrm{rep}^\mu_{c,i}(s) = c_s \text{ if } i \in \mu(c) \qquad\qquad \mathrm{rep}^\mu_{c,i}(s) = q_s \text{ otherwise.}$$

We define $\mathcal{A}_{\mathsf{SC}}$ as follows.

- For each declaration $f : s_1 \dots s_n \to s$ in F, $\mathcal{A}_{\mathsf{SC}}$ contains
$$q_s(f(x_1, \dots, x_n)) \Leftarrow q_{s_1}(x_1), \dots, q_{s_n}(x_n).$$
- For each constructor declaration $c : s_1 \dots s_n \to s$ in C, $\mathcal{A}_{\mathsf{SC}}$ contains
$$c_s(c(x_1, \dots, x_n)) \Leftarrow \mathrm{rep}^\mu_{c,1}(s_1)(x_1), \dots, \mathrm{rep}^\mu_{c,n}(s_n)(x_n).$$
- For distinct sorts $s, s' \in S$ such that $s \leq s'$, $\mathcal{A}_{\mathsf{SC}}$ contains $q_{s'}(x) \Leftarrow q_s(x)$ and $c_{s'}(x) \Leftarrow c_s(x)$.
- For each sort $s \in S$, $\mathcal{A}_{\mathsf{SC}}$ contains $q_\top(x) \Leftarrow q_s(x)$.
- For each term $f(t_1, \dots, t_n) \in I_\mathcal{R}$, $\mathcal{A}_{\mathsf{SC}}$ contains
$$q_{f(t_1,\dots,t_n)}(f(x_1, \dots, x_n)) \Leftarrow q_{t_1}(x_1), \dots, q_{t_n}(x_n).$$
- For each left-hand side $f(t_1, \dots, t_n) \in \mathrm{lhs}(R)$, $\mathcal{A}_{\mathsf{SC}}$ contains
$$r^\mu(f(x_1, \dots, x_n)) \Leftarrow q_{t_1}(x_1), \dots, q_{t_n}(x_n).$$

- Finally, for each operator $f \in F$ with arity n and index $i \in \mu(f)$, $\mathcal{A}_{\mathsf{sc}}$ contains

$$r^\mu(f(x_1, \ldots, x_n)) \Leftarrow q_\top(x_1), \ldots, r^\mu(x_i), \ldots, q_\top(x_n).$$

The rest of our proof is similar to that used in Theorem 6.4.2. By induction on $t \in T_{\overline{\Sigma}}$, we have $t \in \mathcal{L}_{q_s}(\mathcal{A}_{\mathsf{sc}})$ iff $t \in T_{\Sigma,s}$. A similar argument shows that $\mathcal{L}_{q_u}(\mathcal{A}_{\mathsf{sc}})$ iff there is a substitution θ such that $t = u\theta$. This allows us to show that $t \in \mathcal{L}_{r^\mu}(\mathcal{A}_{\mathsf{sc}})$ iff there is a μ-replacing context C, substitution θ, and rule $l \to r$ in $\mathcal{R}$ such that $t = C[l\theta]$. Finally, we can show that $t \in \mathcal{L}_{c_s}(\mathcal{A}_{\mathsf{sc}})$ iff $t \in T_{\Sigma,s}$ and $\mathrm{pos}^\mu(t) \subseteq \mathrm{pos}_C(t)$.

To reduce μ-sufficient completeness to a propositional emptiness we define the formula

$$\phi = \neg r^\mu \wedge \bigvee_{s \in S} q_s \wedge \neg c_s.$$

As A is sort-independent, it follows from Theorem 3.1.1 that $\mathcal{L}_\phi(\mathcal{A}_{\mathsf{sc}}/\overline{A})$ contains equivalence classes which contain a well-sorted term $t \in T_\Sigma$ which is $(\mathcal{R}/A, \mu)$-irreducible while $\mathrm{pos}^\mu(t) \not\subseteq \mathrm{pos}_C(t)$. It follows that $\mathcal{L}_\phi(\mathcal{A}_{\mathsf{sc}}/\overline{A}) = \varnothing$ iff $\mathcal{R}/A$ is μ-semantically complete. □

We have also implemented an algorithm for constructing the automaton $\mathcal{A}_{\mathsf{sc}}$ from a CS Maude specification automatically. In this case, the checker succeeds on the `FACTORIAL` example, as expected:

```
Maude> (mu-scc FACTORIAL .)
Checking the mu-sufficient completeness of FACTORIAL ...
Success: FACTORIAL is mu-sufficiently complete assuming that it is
    ground mu-weakly normalizing and ground mu-sort-preserving.
```

Running the checker on the `INF-LIST` example yields an error:

```
Maude> (mu-scc INF-LIST .)
Checking the mu-sufficient completeness of INF-LIST ...
Failure: The term sel(0,[]) is an mu-irreducible term with sort Nat?
 in INF-LIST with defined symbols in replacement positions.
```

It turns out that the rewrite system given in [101] was missing equations for defining `sel` and `first` when the second argument was the empty list. If we add the equations "`sel(M, []) = none`" and "`first(M, []) = []`" to the Maude specification, the μ-sufficient completeness check succeeds.

6.5 Related work and conclusions

An earlier paper by Lucas [103] has a section on relating the $\mathcal{R}$ and $(\mathcal{R}, \mu)$-canonical forms. In one of the results, a replacement map $\mu^B_{\mathcal{R}}$ is constructed from

$\mathcal{R}$ and the subset of symbols $B \subseteq F$, and results show that $\mathcal{R}$ is μ-canonically complete if the $(\mathcal{R}, \mu)$-irreducible terms are in T_B, and $\mu \supseteq \mu^B_{\mathcal{R}}$. This condition is sufficient to show that the `FACTORIAL` example is μ-canonically complete. However it is easy to give examples where $\mathcal{R}$ is μ-canonically complete, but $\mu \not\supseteq \mu^B_{\mathcal{R}}$. Since we have now a decision procedure for μ-canonical completeness, by varying the replacement map μ, one can use our results to find all *minimal* replacement maps μ for which $\mathcal{R}$ is μ-canonically complete.

It would be useful to investigate the relationships between the work we have presented here and infinite rewriting and infinite normal forms, e.g., [17, 40], which has been extended to the CS case in [100]. In particular, it seems interesting to investigate the relations between algebras of finite and infinite terms, and the extension of sufficient completeness to infinite normal forms.

We have proposed a new model-theoretic semantics for order-sorted CS specifications in the form of the μ-canonical term algebra $\mathrm{Can}^{\mu}_{\mathcal{R}/A}$. And we have investigated three notions of CS completeness: (1) μ-canonical completeness with respect to canonical forms; (2) μ-semantic completeness with respect to equational deduction; and (3) μ-sufficient completeness with respect to constructors. We have also proposed and implemented decision procedures based on propositional tree automata that, under reasonable assumptions on the CS specification (which can be discharged by other existing tools), ensure that it satisfies the different μ-completeness properties. These results provide new ways of reasoning formally about CS equational programs, not only allowing a programmer to check that his/her program behaves as desired, but also to prove properties: for example, it is sound to use an inductive theorem prover to reason about a μ-semantically complete CS program, whereas in general such reasoning may be unsound, since $\mathrm{Can}^{\mu}_{\mathcal{R}/A}$ may not satisfy the equations of $\mathcal{R}$ and may have "junk" data outside the image from the initial algebra.

We think that it would be useful to extend the concepts and results presented here to: (1) more general conditional CS specifications in membership equational logic [111]; (2) CS specifications with non-left-linear rules, for which the tree automata techniques proposed in [80] could be quite useful; and (3) infinite μ-normal forms and infinitary rewriting, as discussed above.

Chapter 7

Inductive theorem proving

Inductive theorem proving is one of the most successful verification techniques for proving complex properties about software algorithms. Many different inductive theorem provers have been developed over the years including ACL2 [91], Coq [12], HOL [65], Isabelle [121], Larch [68], the Maude ITP [27], PVS [128], RRL [88] and SPIKE [11]. These tools support a wide variety of different techniques, logics, and technical approaches.

One common characteristic of inductive theorem provers is that they are almost always *interactive* theorem provers, and proving challenging theorems requires trained user intervention. The advantage of user interaction is that the user can direct the theorem prover to show theorems that cannot be proven by fully automatic techniques. However, it is important from the user's perspective that the theorem prover not require too much input. It is often the case that the user already "knows" the theorem is true, and wants the prover to perform the necessary steps to prove it. However, with current technology the prover will often need direction on problems that appear trivial to the user. Many inductive theorem proof attempts are abandoned when the user decides the theorem prover requires too much involvement. This author has personally abandoned several such proof attempts due to the proof requiring too much work.

Techniques to improve and/or reduce user interaction has been a major line of research in inductive theorem proving, and there are at least three different research directions aimed at improving a user's experience with the inductive theorem prover:

1. Better techniques and heuristics for generating *induction schemas*;
2. More powerful and better integrated *automated reasoning* algorithms and decision procedures to eliminate cases generated from the chosen induction scheme.
3. Improvements to the logic and proof assistant to help the user *understand* the current state of the proof and interact with the prover in a more natural way.

The main focus on this chapter is on the different improvements we have made to the ITP as part of this thesis, and how they relate to these three areas. In addition, we review some of the main existent features of the ITP as well as

discuss improvements in a fourth direction of *extensibility*. These improvements are summarized below.

Induction schemas. The ITP has traditionally supported two forms of induction: structural induction and induction over the less than relation $<$ on the natural numbers. We have extended this with an additional induction scheme: coverset induction [146]. Coverset induction generates induction schemes by analyzing recursive calls in an operation defined by a complete set of terminating rewrite rules, and adding coverset induction to the ITP required developing a new form of coverset induction for membership equational logic.

In addition to extending coverset induction to membership equational logic, we extended coverset induction in several other directions. First, we added a command in the ITP to define an alternate set of memberships for representing the elements in a sort. These alternate set of memberships can be used by coverset induction to generate more appropriate induction schemes. Second, due to our experience with the Powerlist case study, we found it helpful to allow coverset induction to take additional patterns other than the one used to generate the induction scheme. These additional patterns are used to further specialize the subcases generated by the theorem prover, and if used intelligently can help reduce a conjecture to a set of proof obligations that can be proven in a fully automatic way.

Automated Reasoning. We have also improved the core automated tactics used be the Maude ITP by developing new commands and extending existing ones. The main new feature that we have added is the ability to prove that a commutative relation in the user's theory is an equivalence relation, and then automatically propagate facts implied by the transitivity of equivalence relation. This feature has lead to much simpler proofs in the Powerlist case study in the following chapter. The other new command is a command `eq-split` which instantiates universally quantified variables in a goal to better match the left-hand sides of rules in the specification. This is essentially a form of constructor splitting that uses the left-hand side of rules in the specification to control the splitting process.

We have also improved existing commands in several ways: (1) we have improved the `auto` command to automatically split conjunctions into multiple subgoals which are then automatically simplified; (2) we have added a new congruence closure algorithm which fixes the spurious Maude metalavel warnings and improves the performance of the old algorithm; (3) we have added commands to enable and disable rules; and (4) we have improved the parsing of formulas to extract better inference rules from lemmas and the antecedents of a goal which we are proving.

User understandability. In order to help the user better understand what to do next, we have added additional commands for displaying information about the current state of the proof and figure out which rules can be applied. The new commands include a command `red` for evaluating arbitrary terms in the

current theory, a command `show-hyps` for displaying the current hypotheses, and a command `show-rules with` for identifying all inference rules that contain a given operator. These commands help the user to figure out what existing lemmas one may want to apply as well as devise new lemmas to simplify the current goal.

Extensibility. Finally, we have improved the ITP to make it a better platform for future extension. Our first change was to update the ITP to be compatible with the new Maude 2.3 metalavel — the latest public version of Maude at the time of this writing. While doing this, we refactored the ITP's source code to be more readable and documented the different data-structures and invariants of the ITP. Finally, we have overhauled the command parsing component to generate better error messages and become more extensible. One benefit of this work is that Ralf Sasse was able to port his JAVA+ITP tool [132] to the new version of ITP while making no changes to the source code of the ITP. Previously, JAVA+ITP required Java-specific changes to the ITP source code.

In the rest of this chapter we discuss the main improvements to the ITP in more detail. We first start by briefly introducing some of the existing commands that the ITP already supports. In Section 7.2, we describe our basic approach to coverset induction in membership equational logic, and prove its soundness. In Section 7.3, we describe the main enhancements to this basic approach that we have made in the coverset induction's implementation. In Section 7.4, we describe our approach to propagating additional facts when a relation is known to be an equivalence relation. Finally, in Section 7.5, we discuss the other commands that we have added, and in Section 7.6, we discuss possible future research directions for the Maude ITP.

7.1 The Maude ITP

The Maude ITP is an experimental interactive tool for proving properties of the initial algebra $T_{\mathcal{E}}$ of a membership equational logic (MEL) specification $\mathcal{E}$ written in Maude. The ITP has been written entirely in Maude, and is in fact an *executable* specification in MEL of the formal inference system that it implements. The ITP inference system treats MEL specifications as *data* — for example, the ITP command `imp` adds the hypotheses of the current goal as rules to the current goal's associated theory. This makes the ITP a *reflective* design, in which Maude equational specifications become data at the metalevel. Indeed, the fact that membership equational logic is a reflective logic [26] and that Maude efficiently supports reflective MEL computations is systematically exploited in this tool. A similar reflective design has been adopted to develop other formal tools in Maude [25]. Using reflection to implement the ITP tool has one important additional advantage, namely, the ease to rapidly extend it by integrating other tools implemented in Maude using reflection, as it is the case of the sufficient completeness checker presented in Chapter 5.

An ITP session begins with the user providing a Maude theory $\mathcal{E}$ whose equations and rules have been oriented into a conditional rewrite membership system $\mathcal{R}$ along with a first-order MEL formula ϕ which the user wishes to prove holds in the initial algebra $T_{\mathcal{E}}$. The ITP is *interactive* and requires user input to discharge the formula ϕ. At each point in an ITP session, the ITP maintains the sequence of goals remaining to be proved. Each goal has an associated *formula* and an associated *theory* which extends the original theory with lemmas and assumed hypotheses introduced by the user. Once all the goals are discharged, then the original conjecture ϕ has been proven to be true.

The ITP offers many different commands available to the user to aid in the task of proving the remaining goals. A tutorial on the ITP can be found in [27]. Although a complete reference is beyond the scope of this section, we introduce a few of the commands here that are mentioned in this chapter and used in the powerlist case study in the next chapter.

ind and **ind***. The `ind` and `ind*` commands each take a variable x that is universally quantified in the current formula, and perform structural induction on x. The current formula should have the form $(\forall x)\ x : s \implies \phi$, and the structural induction is obtained from the constructor memberships from the original specification. Specifically, for each constructor membership $t : s'$ **if** $\overline{\alpha}$ in the original specification with s' a subsort of s, we create a subgoal

$$(\forall \mathrm{vars}(t))\ \overline{\alpha} \implies \phi[x/t].$$

The current goal is then replaced by these subgoals. The difference between `ind` and `ind*` is that `ind*` will automatically call the `auto` command for simplifying each subgoal, while `ind` will output the unsimplified subgoals. In many proof attempts, `ind*` will completely eliminate all the subgoals, so a single `ind*` command can often discharge the current goal without generating any new subgoals.

cns. The `cns` command performs universal quantifier elimination when the current formula has the form $(\forall Y)\ \phi$. When this command is issued, the ITP introduces a fresh constant c_y for each variable $y \in Y$. This constant is added to the current module, each variable y appearing in the formula ϕ is replaced by c_y, and the outermost quantifier is dropped from the current goal.

e-inst. The `e-inst` command is the counterpart to `cns` for performing existential quantifier elimination. When the current formula has the form $(\exists Y)\ \phi$ and the user issues the command

```
(e-inst with sub .)
```

where *sub* is a substitution $\theta : Y \to T_\Sigma(X)$, the ITP replaces the current formula $(\exists Y)\ \phi$ with the formula $\phi\theta$.

imp. The `imp` command can be used when the current formula is an implication $\phi \implies \psi$ to introduce additional inference rules into the current module. When invoked, the ITP parses ϕ to extract one or more conditional rules to add to the module. Each such rule is labeled as an hypothesis, numbered, and added to the current module. If the antecedent ϕ contains formulas that cannot be added to the module, then each one is added to the current goal as an auxiliary labeled formula which may be instantiated later by the user. After extracting and labeling the rules in ϕ the conditions in ϕ are eliminated and the current formula is replaced with ψ.

lem. The `lem` command is used to introduce lemmas to help prove the current formula. The syntax of the command is

`(lem` *name* `:` *formula* `.)`

When this command is issued, a new goal with the given name is added to the current list of goals with the given formula. In addition, the given formula is parsed with the same algorithm used by `imp` command, and one or more new inference rules may be added to the current module. If the formula cannot be parsed as a rule, it is added as a formula which may be used with the `a-inst` command described below.

a-inst. The `a-inst` goal is primarily used for instantiating universally quantified labeled formulas added by `imp` or `lem` for introducing lemmas. Usually, these are rules that cannot be executed, because the condition contains extra variables. The syntax of `a-inst` is

`(a-inst` *name* `with` *sub* `.)`

where *name* is a labeled formula or inference rule $(\forall Y)\ \phi$ appearing in the current goal, and *sub* is a substitution $\theta : Y \to T_\Sigma(X)$. After this command is issued, the ITP parses $\phi\theta$ to extract new rules, and adds them to the current module.

auto. The `auto` command is perhaps the most commonly used ITP command. It attempts to automatically discharge the current goal through a variety of tactics. If the current goal is a universally quantified formula, it uses the `cns` command to perform quantifier elimination. If the current goal is an implication, it assumes the hypotheses by using the `imp` command. Otherwise, the `auto` command rewrite all the terms in the current formula and the current hypotheses. If the terms in an equation are reduced to the same term $t = t$, then the equation is replaced with `true`. The current goal is eliminated if it reduces to `true`, or if one of the equations in the hypotheses is reduced to `true` = `false`. It would be useful to extend this elimination in future work to more general forms

of hypotheses. For example, it would be useful to eliminate goals containing unsatisfiable hypotheses such as $c + c = 1$ where c is a constant with sort `Nat` by using a linear arithmetic decision procedure.

In this work, we have extended the `auto` command in two ways. The first is to perform equivalence propagation as described later in Section 7.4. The second extension occurs when the current formula is a conjunction $\phi_1 \wedge \cdots \wedge \phi_n$ with $n \geq 2$, the extended `auto` command will automatically split the conjunction into n separate goals with the formulas $\phi_1, \ldots, \phi_n$ respectively. It will then invoke the `auto` command each subgoal. Previously the `auto` command would halt on conjunctions.

This section introduced several of the commands which already exist in the ITP. In the remaining sections we will discuss the additional commands that have been added to the ITP in order to improve its induction features, core reasoning capabilities, and user interface.

7.2 Coverset induction

Coverset induction was introduced in [146] as a method for generating induction schemes for equational specifications where the operations are defined via a terminating and *sufficiently complete* set of rewrite rules (see Chapter 5 for a discussion of sufficient completeness). The idea builds upon Boyer and Moore's idea to use terminating function definitions to generate induction schemes [20]. In coverset induction, the theorem prover chooses a recursively defined function symbol f either by analyzing the current goal with heuristics or by taking a function provided by the user. The prover then extracts an induction schema where each goal is drawn from the left-hand sides of the equations containing f and whose induction hypotheses are obtained from the corresponding occurrences of f on the right-hand side.

Our work on cover set induction in the Maude ITP is based on unpublished joint work with Manuel Clavel, Deepak Kapur, and José Meseguer. It generalizes coverset induction in several different ways.

- The equations in a specification may be conditional.
- The operations and constructors may be partial.
- The induction scheme may be formed from a more general term, where the function we generate the scheme for is allowed to have non-variable arguments, and where the function need not be sufficiently complete.

These three features are essential for the powerlist case study discussed in Chapter 8.

Coverset induction requires that the specification is terminating, and for conditional rewriting the subject of termination is quite complicated. As shown

in [44], it is *insufficient* to merely require that the rewrite relation $\rightarrow_{\mathcal{R}}^{*}$ be well-founded as a rewriting engine may loop when evaluating a condition without ever even applying a single rewrite rule. For CERM systems, perhaps the most effective notion of termination is the notion of *operational termination* described in [44]. The core idea is that an entailment system relative to a specific theory is operationally terminating if there are no well-formed infinite *proof trees*. For our purposes in this work, we define a slightly simpler notion that reuses this basic idea, but is defined in terms of the equational theory $\mathcal{E}$ rather than the rewrite theory $\mathcal{R}$. For the results in this section, we assume that the memberships in $\mathcal{E}$ can all be considered *constructor memberships* (see Chapter 5). We also assume that $\mathcal{E}$ satisfies the following property:

Definition 7.2.1. *A MEL theory $\mathcal{E}$ over a signature $\Sigma = (K, F, S)$ is* equationally reductive *if there is a well-founded ordering $\succ_{\mathcal{E}}$ over terms in T_Σ such that:*

- *For each conditional equation $(\forall Y)\ l = r$ **if** $\overline{\alpha}$ and each substitution $\theta : Y \rightarrow T_\Sigma$ such that $T_{\mathcal{E}} \models \overline{\alpha}\theta$, $l\theta \succ_{\mathcal{E}} r\theta$.*
- *If $t \succ_{\mathcal{E}} u$ then $t \succ_{\mathcal{E}} v$ for all $v \in \text{subterms}(u)$.*

Our algorithm for generating coverset induction schemes is quite similar to the narrowing algorithm used in the sufficient completeness checker described in Section 5.3. The algorithm is a function of several inputs: (1) a MEL theory $\mathcal{E}$ over a signature $\mathcal{E}$ whose equations have been oriented into a set of terminating conditional rewrite and membership rules $\mathcal{R}$, (2) a term $p \in T_\Sigma(X)$ called the pattern term where $\text{vars}(t) = \{ x_1, \ldots, x_n \}$, and (3) a first-order formula ϕ. These inputs are required to satisfy several different properties:

- The formula ϕ has the form

$$\phi = (\forall x_1 \ldots x_n)\ x_1 : s_1 \wedge \cdots \wedge x_n : s_n \implies \phi' \tag{7.1}$$

- The memberships in $\mathcal{E}$ are *properly sorted* (see Definition 5.3.1).
- The equational theory $\mathcal{E}$ is *equationally reductive* modulo A.

The algorithm outputs a set of induction cases $\{ \phi_1, \ldots \phi_m \}$ such that

$$T_{\mathcal{E}} \models \phi \iff T_{\mathcal{E}} \models \phi_1 \wedge \cdots \wedge \phi_m. \tag{7.2}$$

By itself, the property in (7.2) is trivial to satisfy — the set $\{ \phi \}$ would satisfy this property. The utility of coverset induction stems from the fact that coverset induction should generate formulas $\phi_1, \ldots, \phi_m$ that are easier to prove — preferably without using further induction. Our case study in the next chapter will show how coverset induction is useful in proving properties of powerlists.

In unsorted logics, coverset induction for a defined symbol f generates a single induction case for each equation $f(t_1, \ldots, t_n) = u$ in the specification. This approach relies on the specification being sufficiently complete relative to a set of constructor symbols C with $f \notin C$. However, as we argue in Chapter 5, this notion of sufficient completeness is inadequate in the context of membership equation logic due to partiality and potential overloading of constructor symbols. Accordingly, in the context of membership equational logic, we use a different approach for generating coverset induction schemes that does not rely on sufficient completeness explicitly. The techniques are similar to the sufficient completeness checking approach described in Section 5.3. Similar to that approach for sufficient completeness checking, our algorithm for generating coverset induction schemes consists of two phases: a *goal narrowing* phase, and an *induction generation* phase.

Goal narrowing. Given a formula of the form (7.1), the narrowing procedure returns a finite set Δ^* of *conditional terms* $t_{\overline{\alpha}}$ where $t \in T_\Sigma(X)$ and $\overline{\alpha}$ a conjunction of membership and equality atomic formulas. The *variables* in a conditional term are denoted by $\mathrm{vars}(t_{\overline{\alpha}}) = \mathrm{vars}(t) \cup \mathrm{vars}(\overline{\alpha})$. A ground term $u \in T_\Sigma$ is a *ground instance* of $t_{\overline{\alpha}}$ if there is a substitution $\theta : \mathrm{vars}(t_{\overline{\alpha}}) \to T_\Sigma$ such that $u =_{\mathcal{E}} t\theta$ and $T_{\mathcal{E}} \models \overline{\alpha}\theta$. By construction, we guarantee that each ground instance of the conditional term $p_{x_1:s_1 \wedge \cdots \wedge x_n:s_n}$ is a ground instance of some conditional term in Δ^*.

We construct Δ^* incrementally by starting with an initial set Δ^0 and applying an inference rule to obtain $\Delta^1, \Delta^2, \ldots$ until completion to obtain Δ^*. Strictly speaking, this process is not confluent, and different values for Δ^* may be obtained depending on the strategy that controls how the inference rule is applied. We show below that no matter which strategy is used, the resulting set Δ^* will have the same ground instances. We define the initial set Δ^0 as follows:

$$\Delta^0 = \{\, p_{x_1:s_1 \wedge \cdots \wedge x_n:s_n} \,\}.$$

We then apply the rule (7.3) below until completion. This rule defines the *expandability relation* $\blacktriangleleft$ which is defined below:

Definition 7.2.2. *Let t, t' be terms in $T_\Sigma(X)$ such that $\mathrm{vars}(t) \cap \mathrm{vars}(t') = \emptyset$, and $x \in \mathrm{vars}(t)$. Then, $t \blacktriangleleft_x t'$ iff t and t' are unifiable and in the most general unifier $\theta = \mathrm{mgu}(t, t')$, $\theta(x)$ is not a variable.*

Next, we define the inference rule that generates the set Δ. Note that this rule will only be applied a finite number of times, because of the condition $t \blacktriangleleft_x t'$ on the rule.

$$\frac{t_{x:s \wedge \overline{\alpha}} \in \Delta^i \quad \exists x \in \mathrm{vars}(t), l \in \mathrm{lhs}(\mathcal{E}) \text{ s.t. } t \blacktriangleleft_x l}{\Delta^{i+1} := \Delta^i \setminus \{\, t_{x:s \wedge \overline{\alpha}} \,\} \cup \{\, t[x/u]_{\overline{\alpha}[x/u] \wedge \overline{\alpha}_2} \mid u : s \textbf{ if } \overline{\alpha}_2 \in \mathcal{E} \,\}} \tag{7.3}$$

As a simple example, suppose we wanted to prove that

$$(\forall pq)p : \text{Powerlist} \wedge q : \text{Powerlist} \wedge \text{len}(p) = \text{len}(q) \implies \text{len}(p \times q) = \text{len}(p) + \text{len}(q)$$

in the specification `POWERLIST` of powerlists given in Figure 2.5. We can prove this by using coverset induction on the term $p \times q$ where p and q have sort Powerlist. The relevant equations are

$$(p_1 \mid p_2) \times (R \mid S) = (p_1 \times q_1) \mid (p_2 \times q_2) \qquad [m] \times [n] = [M] \mid [N]$$

In this case, both the variables p and q in the coverset term are expandable, because $p \times q \blacktriangleleft_p [m] \times [n]$ and $p \times q \blacktriangleleft_q [m] \times [n]$. We can expand either variable, but in this case we choose to expand p. The relevant memberships are

$$p \mid q : \text{Powerlist} \textbf{ if } p : \text{Powerlist} \wedge q : \text{Powerlist} \wedge \text{len}(p) = \text{len}(q)$$
$$[m] : \text{Powerlist} \textbf{ if } m : \text{Nat}$$

Expanding using these memberships results in the set

$$\Delta^1 = \{\, ((p_1 \mid p_2) \times q)_{p_1, p_2, q : \text{Powerlist} \wedge \text{len}(p_1) = \text{len}(p_2)}, ([m] \times q)_{m : \text{Nat} \wedge q : \text{Powerlist}} \,\}$$

where p_1, p_2, q : Powerlist is equivalent to the conjunction

$$p_1 : \text{Powerlist} \wedge p_2 : \text{Powerlist} \wedge q : \text{Powerlist}.$$

In both of the terms in Δ^1 the variable q is expandable. In the first case, this is because of the first equation on $\times$, and in the second case, this is because of the second equation on $\times$. We can expand these in either order, to yield the set

$$\begin{aligned}\Delta^3 = \{\, & ((p_1 \mid p_2) \times (q_1 \mid q_2))_{p_1, p_2, q_1, q_2 : \text{Powerlist} \wedge \text{len}(p_1) = \text{len}(p_2) \wedge \text{len}(q_1) = \text{len}(q_2)}, \\ & ((p_1 \mid p_2) \times [n])_{p_1, p_2 : \text{Powerlist} \wedge n : \text{Nat} \wedge \text{len}(p_1) = \text{len}(p_2)}, \\ & ([m] \times (q_1 \mid q_2))_{m : \text{Nat} \wedge q_1, q_2 : \text{Powerlist} \wedge \text{len}(q_1) = \text{len}(q_2)}, \\ & ([m] \times [n])_{m, n : \text{Nat}} \,\}\end{aligned}$$

In this case, the algorithm terminates with Δ^3, because the variables are appearing in terms are not further expandable.

It is not difficult to show that each application of the inference rule preserves the ground instances of Δ^i.

Proposition 7.2.3. *Let Δ^{i+1} denote a set of conditional terms obtained from Δ^i by applying the inference rule* (7.3).

A term $u \in T_\Sigma$ is a ground instance of a conditional term in Δ^i iff it is a ground instance of a conditional term in Δ^{i+1}.

Proof. To show this, we must show two things:

(1) If v is a ground instance of $t_{x:s\wedge\overline{\alpha}}$ then $t\theta$ is a ground instance of some pattern in the set

$$\{\, t[x/u]_{\overline{\alpha}[x/u]\wedge\overline{\alpha}_2} \mid u : s \textbf{ if } \overline{\alpha}_2 \in \mathcal{E} \,\} \tag{7.4}$$

where the variables in $\mathcal{E}$ have been renamed to be distinct from $\mathrm{vars}(t_{x:s\wedge\overline{\alpha}})$.

(2) If v is a ground instance of one of the conditional terms in (7.4), then $t\theta$ is a ground instance of $t_{x:s\wedge\overline{\alpha}}$.

Let $Y = \mathrm{vars}(t_{x:s\wedge\overline{\alpha}})$. Showing (2) is quite easy. To show (1), we note that if v is a ground instance of $t_{x:s\wedge\overline{\alpha}}$ then there is a substitution $\theta : Y \to T_\Sigma$ such that $v =_{\mathcal{E}} t\theta$, $T_{\mathcal{E}} \models \theta(x) : s$, and $T_{\mathcal{E}} \models \overline{\alpha}\theta$. As $T_{\mathcal{E}} \models \theta(x) : s$, there must exist a membership $(\forall Z)\, u : s$ **if** $\overline{\alpha}_2$ in $\mathcal{E}$ and substitution ψ such that $\theta(x) =_{\mathcal{E}} u\psi$ and $T_{\mathcal{E}} \models \overline{\alpha}_2\psi$.

As $\mathrm{vars}(u) \cap \mathrm{vars}(t_{x:s\wedge\overline{\alpha}}) = \varnothing$, we can define the substitution $\psi' : Y \setminus \{\, x \,\} \cup Z \to T_\Sigma$ such that

$$\psi'(y) = \psi(y) \text{ if } y \in Z \qquad\qquad \psi'(y) = \theta(y) \text{ otherwise}$$

It is not difficult to show that $v =_{\mathcal{E}} t\theta =_{\mathcal{E}} t[x/u]\psi'$, $T_{\mathcal{E}} \models \overline{\alpha}[x/u]\psi'$, and $T_{\mathcal{E}} \models \overline{\alpha}_2\psi'$. It follows that v is a ground instance of $t[x/u]_{\overline{\alpha}[x/u]\wedge\overline{\alpha}_2}$. □

The following corollary captures the main correctness property of the narrowing phase. It is an immediate consequence of the previous proposition.

Corollary 7.2.4. *If Δ^* is obtained from Δ^0 by applying the inference rule* (7.3) *until completion, then each ground instance $p\theta$ of $p_{x_1:s_1\wedge\cdots\wedge x_n:s_n}$ is a ground instance of some conditional term in Δ^*.*

Induction generation. The induction generation phase takes the set Δ^* generated by narrowing along with the theory $\mathcal{E}$, pattern $p \in T_\Sigma(X)$, and goal formula ϕ. The phase outputs a subgoal for each conditional term in Δ^*. We first observe that for each conditional term in $t_{\overline{\alpha}} \in \Delta^*$, there must be a substitution $\theta : \mathrm{vars}(p) \to T_\Sigma(X)$ such that $t = p\theta$. The substitution θ will be used for the induction subgoal. In order to definite the induction hypotheses, we find the instances of p that appear in the right hand side of a rule $l\theta = r\theta$ **if** $\overline{\alpha}\theta$ where $l = r$ **if** $\overline{\alpha}$ is an equation in $\mathcal{E}$. Specifically, the induction formula $\mathrm{IC}_\phi(p\theta_{\overline{\alpha}})$ of a condition term $p\theta_{\overline{\alpha}} \in \Delta^*$ is defined as follows:

$$\mathrm{IC}_\phi(p\theta_{\overline{\alpha}}) = (\forall \mathrm{vars}(p\theta_{\overline{\alpha}}))\ \overline{\alpha} \wedge \bigwedge_{\substack{l=r \textbf{ if } \overline{\alpha}_2\in\mathcal{E} \\ (\exists\psi)\ l\psi=p\theta \\ p\rho\in\mathrm{subterms}(r\psi)}} (\overline{\alpha}_2\psi \implies \phi\rho) \implies \phi'\theta$$

The correctness of our coverset induction algorithm is shown by the following theorem:

Theorem 7.2.5. *Let $\mathcal{E}$ denote a MEL theory that is equationally reductive with respect to the well-founded ordering $\succ_{\mathcal{E}}$. Given a formula ϕ of the form*

$$(\forall x_1 \ldots x_n)\, x_1 : s_1 \wedge \cdots \wedge x_n : s_n \implies \phi'$$

and a term $p \in T_\Sigma(\{ x_1, \ldots, x_n \})$, let Δ^* be a set of conditional terms obtained by applying the inference rule (7.3) *until completion starting from the initial set $\{ p_{x_1:s_1 \wedge \cdots \wedge x_n:s_n} \}$. It is the case that*

$$T_{\mathcal{E}} \models \phi \iff \bigwedge_{p\theta_{\overline{\alpha}} \in \Delta^*} T_{\mathcal{E}} \models \mathrm{IC}_\phi(p\theta_{\overline{\alpha}}).$$

Proof. Let $Y = \{ x_1, \ldots, x_n \}$.

We first show that $T_{\mathcal{E}} \models \phi$ implies that $T_{\mathcal{E}} \models \mathrm{IC}_\phi(p\theta_{\overline{\alpha}})$ for each conditional term $p\theta_{\overline{\alpha}}$ in Δ^*. We prove this by assuming $T_{\mathcal{E}} \models \phi$ and showing that for each ground substitution $\rho : \mathrm{vars}(p\theta_{\overline{\alpha}}) \to T_\Sigma$, $T_{\mathcal{E}} \models \overline{\alpha}\rho$ implies $T_{\mathcal{E}} \models \phi'\theta\rho$.

Note that if $T_{\mathcal{E}} \models \overline{\alpha}\rho$, then $p\theta\rho$ is a ground instance of $p\theta_{\overline{\alpha}}$. By Corollary 7.2.4, this implies that $p\rho\theta$ is a ground instance of $p_{x_1:s_1 \wedge \cdots \wedge x_n:s_n}$, and thus for each $T_{\mathcal{E}} \models x_i\theta\rho : s_i$ for each $i \in [1, n]$. As $T_{\mathcal{E}} \models \phi$ by assumption, we know that for each ground substitution $\psi : Y \to T_\Sigma$ if $\mathcal{E} \vdash x_i\psi : s_i$ for $i \in [1, n]$, then $T_{\mathcal{E}} \models \phi'\psi$. If we let $\psi = \theta\rho$, it follows that $T_{\mathcal{E}} \models \phi'\theta\rho$.

It remains to show if $T_{\mathcal{E}} \models \mathrm{IC}_\phi(p\theta_{\overline{\alpha}})$ for all conditional terms $p\theta_{\overline{\alpha}}$ in Δ^*, then $T_{\mathcal{E}} \models \phi$. First, we use $\succ_{\mathcal{E}}$ and p to define an well-founded ordering $\succ_{\mathcal{E},p}$ over ground substitutions $\theta : \mathrm{vars}(p) \to T_\Sigma$ by letting

$$\theta_1 \succ_{\mathcal{E},p} \theta_2 \iff p\theta_1 \succ_{\mathcal{E}} p\theta_2.$$

We show that $T_{\mathcal{E}} \models \phi$ by showing by induction on $\succ_{\mathcal{E},p}$ that for each substitution ψ such that $T_{\mathcal{E}} \models \psi(x_i) : s_i$ for $i \in [1, n]$, $T_{\mathcal{E}} \models \phi'\psi$.

Given a substitution ψ where $T_{\mathcal{E}} \models \psi(x_i) : s_i$ for each $i \in [1, n]$, we know that $p\psi$ is a ground instance of $p_{x_1:s_1 \wedge \cdots \wedge x_n:s_n}$. It follows by Corollary 7.2.4 that there is a conditional term $p\theta_{\overline{\alpha}} \in \Delta^*$ such that $\theta\rho = \psi$ and $T_{\mathcal{E}} \models \overline{\alpha}\rho$ for some ground substitution $\rho : \mathrm{vars}(p\theta_{\overline{\alpha}}) \to T_\Sigma$. Since $\psi = \theta\rho$, we can show $T_{\mathcal{E}} \models \phi'\psi$ by showing $T_{\mathcal{E}} \models \phi'\theta\rho$.

Our hypothesis has assumed that $T_{\mathcal{E}} \models \mathrm{IC}_\phi(p\theta_{\overline{\alpha}})$. Consequently,

$$T_{\mathcal{E}} \models \overline{\alpha}\rho \wedge \bigwedge_{\substack{l=r \textbf{ if } \overline{\alpha}_2 \in \mathcal{E} \\ (\exists \psi_2)\, l\psi_2 = p\theta \\ p\rho_2 \in \mathrm{subterms}(r\psi_2)}} (\overline{\alpha}_2\psi_2\rho \implies \phi\rho_2\rho) \implies \phi'\theta\rho.$$

Therefore, we can show that $T_{\mathcal{E}} \models \phi'\theta\rho$ by showing that $T_{\mathcal{E}} \models \phi\rho_2\rho$ for each rule $l = r$ **if** $\overline{\alpha}_2$ in $\mathcal{E}$, each substitution ψ_2 such that $l\psi_2 = p\theta$ and $T_{\mathcal{E}} \models \overline{\alpha}\psi_2\rho$, and each substitution ρ_2 such that $p\rho_2 \in \mathrm{subterms}(r\psi_2)$.

However, this follows by our induction hypothesis as $p\theta\rho = l\psi_2\rho \succ_{\mathcal{E}} r\psi_2\rho$ and $p\rho_2\rho \in \mathrm{subterms}(r\psi_2\rho)$ implies that $p\theta\rho \succ_{\mathcal{E}} p\rho_2\rho$. Consequently, $\psi =$

$\theta\rho \succ_{\mathcal{E},p} \rho_2\rho$ and so by our induction hypothesis $T_{\mathcal{E}} \models \phi\rho_2\rho$. □

7.3 Coverset induction in the Maude ITP

We have extended the Maude ITP with two commands `cov` and `cov*` which apply coverset induction when invoked by the user. The difference between the two commands is that `cov*` will automatically simplify all of the subgoals generated by coverset induction with the `auto` command, while `cov` will leave them unchanged. Each command takes the pattern as an argument with the syntax:

`(cov on` *pattern*`)` `(cov* on` *pattern*`)`

where *pattern* is a term whose variables are universally quantified in the current formula.

One useful feature of coverset induction is that, in addition to generating potentially useful induction hypotheses, it specializes terms appearing in the current problem to match additional rules in the specification. This allows them to be simplified by rewriting. Splitting based on constructors is called *constructor splitting*, and has been quite useful in our powerlist case study. The ITP already offers a command `ctor-term-split` to do this, but the command only replaced a single variable with its constructor memberships, and did not explicitly attempt to match a term against the left-hand sides of equations. As a consequence, `ctor-term-split` often had to be invoked several times to achieve the required matching. In contrast, a single coverset induction command would have done the job. Coverset induction also potentially introduced induction hypotheses even if they were not necessary, and so we decided to add the commands `eq-split` and `eq-split*` which essentially perform coverset induction, but do not add the induction hypotheses.

`(eq-split on` *pattern* `.)` `(eq-split* on` *pattern* `.)`

The difference between these two commands is that `eq-split*` invokes the `auto` command to attempt to automatically discharge each subgoal while `eq-split` does not modify the generated goals.

Coverset induction is the main type of induction that we use in the powerlist case study in Chapter 8. Our experience led us to improve the theoretical algorithm described in the previous section in several ways: (1) apply the Δ-rule only to variables that are *demanded* by the most equations in $\mathcal{E}$; (2) use a simple *subsumption* test to eliminate redundant induction cases; (3) allow substitutions in subgoals to be further specialized by *additional patterns*; and (4) allow the user to define *alternative constructors* for sorts in the specification.

7.3.1 Most demanded variables

One of the main differences between our approach to coverset induction in membership equational logic and the original idea in unsorted logics is that our scheme may potentially generate multiple induction subgoals for each equation. One heuristic to reduce the number of subgoals is to use an intelligent strategy for deciding when to expand a variable x using the rule (7.3). For the ITP, if multiple variables may be expanded by the rule (7.3), we expand the variable demanded by a maximal number of equations. This is similar to the idea of *natural narrowing* [46], with the main difference being that we are narrowing a variable to consider *all* ground instances of a term rather than just considering *specific* instances that unify with a set of patterns.

To illustrate why the choice of which variable to expand is important, consider the theory $\mathcal{E}$ with constants a and b and a binary symbol f. In addition $\mathcal{E}$ contains constructor memberships $a : s$ and $b : s$, and equations:

$$f(a, x) \qquad f(b, a) \qquad f(b, b)$$

If asked to perform coverset induction with the pattern $f(x_1, x_2)_{x_1:s \wedge x_2:s}$, we can expand either x_1 or x_2. If we expand x_2 in $f(x_1, x_2)_{x_1:s \wedge x_2:s}$ using the rule (7.3), we get the patterns

$$\{\, f(x_1, a)_{x_1:s}, f(x_1, b)_{x_1:s} \,\}.$$

We next can only expand x_1 in both of these goals and the narrowing phase terminates with four cases:

$$\{\, f(a, a), f(a, b), f(b, a), f(a, b) \,\}$$

The schema generation phases will then produce four separate subgoals, one for each term. However, if we expand x_1 first, we obtain the patterns

$$\{\, f(a, x_2)_{x_2:s}, f(b, x_2)_{x_2:s} \,\}.$$

We then only need to expand x_2 in $f(b, x_2)$, and thus the narrowing phase terminates with the patterns

$$\{\, f(a, x_2)_{x_2:s}, f(b, a), f(b, b) \,\}$$

which are then instantiated into three subgoals.

Given a pattern p and an equation $l = r$ **if** $\overline{\alpha}$, we say that a variable $x \in \mathrm{vars}(p)$ is *demanded* by l iff there is a most-general unifier $\theta = \mathrm{mgu}(p, t)$ such that $\theta(x)$ is not a variable. The strategy we have chosen for the ITP is to only apply the rule (7.3) to variables that are demanded by a maximal number of equations in $\mathcal{E}$. This is a greedy algorithm, but has worked well in the Powerlist

case study described in the next chapter.

7.3.2 Subsumption checking

A second optimization that we use is to perform subsumption checking to reduce the number of cases. We say that a conditional term $t_{\overline{\alpha}}$ *subsumes* $t'_{\overline{\alpha}'}$ iff each ground instance of $t'_{\overline{\alpha}'}$ is a ground instance of $t_{\overline{\alpha}}$. Due to the potential for arbitrary conditions $\overline{\alpha}$ and $\overline{\alpha}'$, it is in general undecidable whether one conditional term subsumes another. This means that in general, our cover-set induction approach will always generate an induction scheme, but it may have redundant cases. To eliminate some of those redundant cases, we have implemented an approximate solution that can often detect subsumption syntactically. To understand our approach, first observe that $t_{\overline{\alpha}}$ subsumes $t'_{\overline{\alpha}'}$ if there is a substitution θ such that:

- $t\theta = t'$,
- for each equation $l = r \in \overline{\alpha}$, $l\theta = r\theta \in \overline{\alpha}'$, and
- for each membership $l : s \in \overline{\alpha}$, there is a membership $l\theta : s' \in \overline{\alpha}'$ such that s' is a subsort of s.

These properties are easy to check, and the ITP will remove a conditional term $t'_{\overline{\alpha}'}$ from the current conditional terms Δ^i when a different conditional term $t_{\overline{\alpha}}$ is detected to satisfy these three properties. This removal is sound, because the set of ground instances of Δ^i are preserved.

To see how this subsumption is useful, consider the following theory $\mathcal{E}$ over a signature with sorts $s_1 < s_2$, and unary functions c and f that contains the memberships:

$$x : s_2 \textbf{ if } x : s_1 \qquad c(x) : s_1 \textbf{ if } x : s_1 \qquad c(x) : s_2 \textbf{ if } x : s_2.$$

Additionally, $\mathcal{E}$ contains a single equation with the left-hand side $f(c(x))$. Given the initial pattern $f(x)_{x:s_2}$, we expand x with the memberships $x : s_2$ **if** $x : s_1$ and $c(x) : s_2$ **if** $x : s_2$. This yields the set

$$\{\, f(x)_{x:s_1}, f(c(x))_{x:s_2} \,\}$$

We then must further instantiate $f(x)_{x:s_1}$ to yield the final set

$$\{\, f(c(x))_{x:s_1}, f(c(x))_{x:s_2} \,\}$$

However, since s_1 is a subsort of s_2, we have an extra pattern $f(c(x))_{x:s_1}$ which would generate a redundant induction case. By using our syntactic check, the ITP will automatically remove $f(c(x))_{x:s_1}$.

7.3.3 Additional patterns

As shown in the next chapter, many of the lemmas in our powerlist case study are discharged with a single `cov*` command. However, for many of the lemmas where this failed, we could discharge them automatically if we performed additional constructor splitting on terms appearing in the subgoal. This process requires multiple commands, and we decided to automate the process by introducing two coverset induction commands that take additional patterns, called *split patterns*, which are used for splitting the subgoals. They are not used to generate the induction hypothesis. The two commands have the following syntax:

```
(cov-split  on pattern split split-patterns)
(cov-split* on pattern split split-patterns)
```

where *pattern* is the term used for coverset induction and *split-patterns* is a semicolon separated list of terms.

The `cov-split` (resp. `cov-split*`) commands can be thought of as performing coverset induction with the given pattern, and then `eq-split` (resp. `eq-split*`) on the induction cases. The actual implementation is somewhat different, and is achieved by modifying the narrowing phase described in the previous section. Essentially, our narrowing phase additionally guarantees for each pattern p in the split patterns and each left-hand side l of an equation in $\mathcal{E}$ that

$$\text{mgu}(p, l) \neq \varnothing \implies p \text{ matches } l.$$

Our experience with the powerlist case study has shown that virtually all of the lemmas involving coverset induction and `eq-split` could be solved in a single `cov-split*` command.

7.3.4 Alternative constructors

The fourth and final extension to coverset induction that we have implemented in the ITP is the ability to define alternative constructor declarations with the command

$$(\texttt{ctor-def}\ \mathit{name} : \texttt{A\{}x : s\texttt{\}} \\ (\texttt{E\{}Y_1\texttt{\}}\ t_1 = x\ \texttt{\&}\ \mathit{cond}_1)\ \texttt{V}\ \ldots\ \texttt{V}\ (\texttt{E\{}Y_n\texttt{\}}\ t_n = x\ \texttt{\&}\ \mathit{cond}_n)\ .)$$

where each formula cond_i is a (possibly empty) conjunction of equations and memberships.

After giving this command to the ITP, the ITP creates a subgoal which requires the user to prove the given formula, and it adds a set of alternative memberships

$$M_{name} = \{\, t_1 : s \textbf{ if } \mathit{cond}_1 \qquad \ldots \qquad t_n : s \textbf{ if } \mathit{cond}_n \,\}$$

to the current goal. These memberships can then be used in lieu of the normal constructor memberships with sort s during the narrowing phase of coverset induction. In order to specify which alternative memberships to use, we added the following four commands:

```
(cov using names on pattern)
(cov* using names on pattern)
(cov-split  using names on pattern split split-patterns)
(cov-split* using names on pattern split split-patterns)
```

where *names* is a semicolon separated list with the names of constructor definition names. Each name in the list must be associated to memberships for a distinct sort, and when a name is provides we replace the memberships in $\mathcal{E}$ for that name with the memberships M_{name} for the purposes of instantiating a variable using the rule (7.3).

Alternate constructors are used in the powerlist case study in several places. A key property of powerlists is that each powerlist with more than one element can be represented as either the *concatenation* $P \mid Q$ of two powerlists or the *interleaving* $P \times Q$ of two powerlists. In our Maude specification of powerlists, we use a membership with $\mid$ as the main constructor, but prove an alternate set of constructors with $\times$. For operations that are most naturally defined using $\times$, we use the alternate constructors for coverset induction.

In some cases, one may want to make the alternative constructor definitions the default constructors. This can be done with the command

```
(set-default-ctor name.)
```

After issuing this command the memberships with the given name will be used for their associated sort whenever constructor narrowing occurs. The default set of memberships for a sort can be used by calling `set-default-ctor` with the name of the sort.

7.4 Equivalence propagation

In addition to coverset induction, we have extended the ITP with specialized support for equivalence relations. As membership equational logic is functional and does not support relations other than the sort predicates, a *relation* in the context of this section is a binary function whose output kind is the kind used by the built-in Boolean type. A relation p is an *equivalence relation* over a sort $s \in S$ for our purposes if it is labeled with the `comm` attribute marking the symbol as commutative and satisfies the following two properties:

$T_{\mathcal{E}} \models (\forall x : s)\ p(x, x) = \texttt{true}$
$T_{\mathcal{E}} \models (\forall x : s, y : s, z : s)\ p(x, y) = \texttt{true} \wedge p(y, z) = \texttt{true} \implies p(x, z) = \texttt{true}$

where `true` refers to the operator for true in the predefined `BOOL` module and $(\forall x : s)\ \phi$ is syntactic sugar for the formula $(\forall x)\ x : s \implies \phi$.

Equivalence relations benefit from specialized automated reasoning support, because ordinary rewriting cannot deal with extra variable y in the condition of the transitivity axiom

$$p(x, z) \textbf{ if } p(x, y) \wedge p(y, z).$$

Our solution to this has been to extend the ITP with two commands: a command `defequiv` for defining equivalence relations, and `equiv-propagate` for propagating facts implied by transitivity. In addition, the built-in *auto* command has been extended to also perform equivalence propagation in addition to its other tactics.

To indicate that a given operation is an equivalence relation, we have added the following command to the ITP:

`(defequiv` *p* `on` *sort* `.)`

The operation p must be labeled with the commutativity attribute. When this command is issued to the ITP, the ITP generates two subgoals — one for each equation that an equivalence relation must satisfy, and then records in the original goal that the operation p is an equivalence relation for arguments with a sort *sort*

Once one or more equivalence relations are added using `defequiv`, equivalence propagation will automatically occur when the user calls the `auto` command which automatically applies several tactics including rewriting, equivalence propagation, and hypothesis simplification to resolve the formula. In addition, the user may request equivalence propagation to occur with the command `(equiv-propagate .)` When equivalence propagation is used with either command, for each predicate p that is an equivalence relation on s, we apply the following rule until completion.

$$\frac{p(t, u) = \texttt{true}, p(u, v) = \texttt{true} \in \mathcal{E} \text{ s.t. } p(t, v) \downarrow_{\mathcal{E}} \neq \texttt{true}}{\mathcal{E}' := \mathcal{E} \uplus \{\, p(t, v) = \texttt{true} \,\}}$$

where $\mathcal{E}$ denotes the theory containing the current module and any hypotheses assumed in the current goal, and $p(t, v) \downarrow_{\mathcal{E}}$ denotes the term obtained by rewriting $p(t, v)$ with the equations in $\mathcal{E}$ oriented as rules. After applying the rule, we replace the current goal's theory $\mathcal{E}$ with $\mathcal{E}'$, and then repeat the process until either: (1) the rule can no longer be applied, or (2) we detect a conflict because $p(t, u) =$ `true`, $p(u, v) =$ `true` and $p(t, v) \downarrow_{\mathcal{E}} =$ `false`. If a conflict is detected, then the current hypotheses are unsatisfiable, and the current goal is immediately discarded.

7.5 Other commands

In this section we describe the other commands which we added to the ITP for developing a proof strategy, and debugging failed proofs.

Enable/Disable. The `enable` and `disable` commands control the executability of the different rewrite rules and memberships in the current goal. The rules can either come from hypotheses in the current module, lemmas that were previously added, or labeled equations in the original user's module.

`(enable` *rule-name* `.)` `(disable` *rule-name* `.)`

If `enable` is called with the name of a rule in the module labeled with the attribute `nonexec`, it will discard the `nonexec` attribute, thus enabling the rule during rewriting. Conversely, when the `disable` command is given for a rule that is not labeled with the `nonexec` attribute, it will add the `nonexec` attribute to the rule, thus disabling it when rewriting is used to simplify goals. These commands can be used in debugging to help identify non-terminating hypotheses or lemmas. They also can be used for *information hiding*. It is often useful to prove lemmas that state the essential properties of an operation, and then disable the operation's definition.

Reduction. It can be difficult to remember all the lemmas and hypotheses added to a module, and sometimes rules may fail to apply because a condition cannot be resolved. Unfortunately, there is no automatic way to fix the second problem, but to aid the debugging process, we have added the command

`(red` *term* `.)`

which computes the canonical form and least sort of an arbitrary term in the current module.

Showing rules. A large part of the success of inductive theorem provers stems from a user's ability to construct a set of terminating rules that yield unique normal forms for terms appearing in the current goal. In the ITP, the rules depend not only on the definitions in the user's module and lemmas, but also on the hypotheses added to the current proof attempt. Unlike the lemmas, the hypotheses are usually different for each subgoal, and so it is often helpful to see the current hypotheses assumed in the current goal. To do this, we added the command `(show-hyps .)` to show the current hypotheses.

In addition to the hypotheses, it is often useful to see *all* of the rules related to a given symbol appearing in the current goal. Although the existing `show-all` command will display all of the rules, it can be tedious to sort through them in larger proofs to see the rules that are currently relevant. For more targeted searches, we added the command

```
(show-rules with op .)
```

which will display the equations and memberships whose left-hand side references *op*.

7.6 Conclusions and future work

In this chapter we have presented several improvements to the Maude Inductive Theorem Prover. This includes a general form of coverset induction, equivalence propagation, and several different commands to help debug proofs using the rewriting engine. These techniques have already proven useful in the powerlist case study which we will discuss in the next chapter. However, there are still many ways the ITP could be improved.

Our approach to coverset induction is similar to the first sufficient completeness checking algorithm described in Section 5.3. It would also be interesting to develop equational tree automata-based techniques for generating coverset induction schemes. These techniques should allow us to generate better coverset induction schemas with specifications supporting rewriting modulo axioms. This is related to the work in [86] which considered rewriting modulo linear arithmetic. Our work would build upon that to consider other theories such as the combinations of associativity, commutativity, and identity supported by our CETA tree automata library.

A second research direction would be to extend the `auto` tactic to take advantage of recent advances in efficient decision procedures for uninterpreted function symbols, linear arithmetic, bit vectors, and arrays. These decision procedures have been successfully combined with algorithms for Boolean satisfiability as the basis for SAT Modulo Theories (SMT)-based theorem proving, and there are currently many different SMT-theorem provers including Barcelogic [13], CVC3 [9], MathSAT [23], Yices [45], and Z3 [116]. One recent direction is to combine the capabilities of SMT-based theorem provers with matching modulo ground equations [115] and superposition [114]. It seems worth investigating whether these techniques can be combined with the (conditional) rewriting supported by the ITP in an efficient way.

A third direction for further research is to further decouple the reasoning capabilities in the ITP from the ITP's user interface. This would enable the core reasoning of the ITP to more easily be integrated in other tools such as Full Maude or Real-time Maude. This should be possible since all of these tools use the core Maude metalevel infrastructure. One step in this direction that we have taken in the ITP is to make the module used to define the ITP's commands an argument to the initial ITP state. This allows us to define new commands to the ITP in a modular way without altering any of the existing ITP's source code. After making this change, Ralf Sasse was able to port the JAVA+ITP [132] extension to the newest version of the ITP with no code changes to the ITP's

source code.

It is an exciting time for automated theorem proving as their techniques have become fundamental to many different formal analysis tools. The ITP has a great deal of potential to become useful across the different Maude tools. However, it is important to carefully validate these ideas on different domains.

Chapter 8

Powerlist case study

To evaluate the Maude ITP and the extensions introduced in the previous chapter, we present a case study in which we mechanically prove the correctness of many different algorithms over Misra's powerlist [113] data type. Powerlists are non-empty lists which can be nested, but where two lists P and Q can be concatenated to form $P \mid Q$ if and only if they are *similar* — that is they have the same number of nested levels and the same length at each level. Powerlists satisfy the nice algebraic property that the interleaving $P \times Q$ of two similar powerlists P and Q can be distributed over concatenation,

$$(P_1 \mid P_2) \times (Q_1 \mid Q_2) = (P_1 \times Q_1) \mid (P_2 \times Q_2) \qquad S_1 \times S_2 = S_1 \mid S_2$$

where P_1, P_2, Q_1, and Q_2 are all similar powerlists, while S_1 and S_2 denote single elements or nested powerlists. We sometimes refer to $\mid$ as the *tie* operator, and $\times$ as the *zip* operator.

It is not difficult to see that each powerlist has 2^n elements for some $n \in \mathbb{N}$. A crucial property of powerlists is that each powerlist with more than one element can be represented as either the concatenation $P_1 \mid P_2$ or the interleaving $Q_1 \times Q_2$ of two similar powerlists. Moreover, the conversion between the interleaving and concatenation representations can be done efficiently in parallel architectures [96]. Powerlists can also succinctly describe many different parallel algorithms in a way that leads to elegant correctness proofs [1, 96, 113].

Powerlists have served as a challenge problem for inductive theorem provers. There are several papers which reproduce the different correctness properties of powerlists in both ACL2 [55–57] and the RRL [85, 87]. Unfortunately, it is widely felt that these proofs lack the simplicity of hand proofs. Attempts to replicate that simplicity have lead to new techniques for simplifying inductive theorem proofs [84], however there have not yet been any completely satisfactory solutions. The aim of this work is to replicate many of these proofs using the Maude ITP in a natural way that achieves the simplicity of the hand proofs. We hope to use this experience to identify both strengths of the ITP that may be useful in other theorem provers and weaknesses where we can improve the ITP.

An important reason why the powerlist proofs are complicated in other formal tools is that powerlists cannot be naturally formalized in the logics sup-

ported by those tools. For example, ACL2's logic is an applicative subset of Common Lisp, and powerlists were encoded in [56] as trees formed from `cons` and `nil` rather than treating the tie and/or zip operators directly as constructors. Our goal is to determine if membership equational logic can avoid this problem. One important feature of MEL is that the partial powerlist constructors of $|$ and $\times$ can be naturally expressed as conditional memberships.

Our results are encouraging, and we have verified many of the powerlists theorems in existing work in the Maude ITP. We have formally proven many of the theorems in [113] on basic properties of powerlists, and on the Fast Fourier Transform (FFT), inverse FFT, and the Batcher sort algorithm. We also have proven the results in [1] on representing arbitrary size ripple-carry and carry-lookahead adders in powerlists. The proofs scripts for the different proofs contain over 100 different theorems and lemmas. The vast majority of these were proven using a single ITP command using coverset induction or the `cov-split` described in the previous chapter.

The rest of this chapter is organized as follows. In the next section we introduce our parameterized specification of powerlists in membership equational logic. This parameterized specification forms the basis for all of the later proofs, and is instantiated in later proofs for powerlists over natural numbers, complex numbers, and bits. In Section 8.2, we show how basic algorithms on powerlists can be proven correct in the ITP. Many of the lemmas in this section are reused in later proofs. In Section 8.3, we present our definitions and correctness proofs for the FFT and inverse FFT. In Section 8.4, we present our results on the Batcher sort algorithm. In Section 8.5, we present our results on the ripple-carry and carry-lookahead adders. Finally, we conclude in Section 8.6 with a discussion of how our work relates to existing work as well as promising future directions of research. To improve the flow of this chapter, we will not always show the full proof scripts, however they are included in Appendices A–D.

8.1 Powerlists in membership equational logic

Powerlists [113] are often treated as an abstract data type in which the basic element type is left undefined. They could be natural numbers, complex numbers, bits, or any other data type. Formally, this is most naturally captured by defining powerlists as a parametric data type. The parameter must contain at least one sort for defining the elements of the powerlist, but some of the algorithms described in [113] require other operations as well. For example, sorting the elements in a powerlist requires that the parameter is equipped with a total order, while computing a prefix sum requires that the parameter is equipped with an associative operation.

As explained in [28], parametric data types are supported in Maude by defining one or more named theories for the parameters, and a parametric functional module taking the theory as an argument. Maude also provides several built-in

theories in its prelude — the most basic of which is the TRIV theory defined below:

```
fth TRIV is
  sort Elt .
endfth
```

The only elements of this theory are an implicit kind [Elt] and a single sort Elt with the kind [Elt].

Given the theory TRIV, we then define powerlists in Maude using the specification in Figure 8.1. The module POWERLIST defines three sorts for unnested powerlists: Elt{X} for powerlists with a single element, NsPowerlist{X} for powerlists with multiple elements, and Powerlist{X} for all unnested powerlists. The module also defines three sorts for potentially nested powerlists: Scalar{X} for powerlists with a single element or nesting, NsPowerlist*{X} for potentially nested powerlists with length greater than two, and Powerlist*{X} for all powerlists. The operator [_] represents a single element powerlist, <_> represents a single nested powerlist, and _tie_ represents the concatenation of two powerlists. Note that, in general, P tie Q is not a well-sorted term. Instead, we define the similarity relation sim? and two conditional memberships to define when P tie Q has sorts NsPowerlist{X} and NsPowerlist*{X}. The last operation we introduce is the zip operator which interleaves two powerlists. Later proofs will show how it can be an alternate constructor for powerlists.

We have shown how to define parametric theories, but we still need to show how to use them via *instantiation*. Instantiating POWERLIST requires a mapping which associates the sort X$Elt in the TRIV parameter theory of POWERLIST with a concrete sort in a Maude module. This is done by defining a *view* from the parameter theory to a concrete module. The Maude prelude also includes several predefined views. For example, TRIV maps its Elt sort to the sort Nat corresponding to the natural numbers in the built-in module NAT with the view:

```
view Nat from TRIV to NAT is
  sort Elt to Nat .
endv
```

Note that the identifier Nat is used in two distinct ways in this definition: it is used as the name of the *view* as well as the name of the *sort* for representing the natural numbers in the module NAT. This overloading is not required, but as a useful convention adopted in the Maude prelude for naming views from TRIV.

The POWERLIST specification can then be instantiated with the view Nat by importing it into another module with the line

```
protecting POWERLIST{Nat} .
```

Note that Nat refers to the name of the view, and not to the sort. The parameter X in POWERLIST will be replaced with Nat in the names of the sorts, so the sorts in POWERLIST imported by this line will be named Elt{Nat}, NsPowerlist{Nat},

```
fmod POWERLIST{X :: TRIV} is protecting NAT .
  --- Sorts for unnested powerlists.
  sorts Elt{X} NsPowerlist{X} Powerlist{X} .
  subsort Elt{X} NsPowerlist{X} < Powerlist{X} .
  --- Sorts for nested powerlists.
  sorts Scalar{X} NsPowerlist*{X} Powerlist*{X} .
  subsort Scalar{X} NsPowerlist*{X} < Powerlist*{X} .
  --- Subsorts relating unnested and nested powerlists.
  subsort Elt{X} < Scalar{X} .
  subsort NsPowerlist{X} < NsPowerlist*{X} .
  subsort Powerlist{X} < Powerlist*{X} .

  --- Construct single element powerlist.
  op [_] : X$Elt -> Elt{X} [ctor].
  --- Construct nested powerlist.
  op <_> : Powerlist*{X} -> Scalar{X} [ctor].
  --- Partial tie constructor.
  op _tie_ : Powerlist*{X} Powerlist*{X} ~> Powerlist*{X} [prec 35].

  var E E' : X$Elt .
  var S S' : Scalar{X} .
  var P Q : Powerlist{X} .
  var P* P1* P2* Q* Q1* Q2* : Powerlist*{X} .
  var NsP* : NsPowerlist*{X} .

  --- Similarity predicate.
  op sim? : Powerlist*{X} Powerlist*{X} -> Bool [comm].
  eq sim?([ E ], [ E' ])  = true .
  eq sim?(< P* >, [ E ]) = false .
  eq sim?(< P* >, < Q* >) = sim?(P*, Q*) .
  eq sim?(NsP*, S) = false .
  eq sim?(P1* tie P2*, Q1* tie Q2*) = sim?(P1*, Q1*) .

  --- Constructor memberships.
  cmb P tie Q : NsPowerlist{X} if sim?(P, Q) .
  cmb P* tie Q* : NsPowerlist*{X} if sim?(P*, Q*) .

  --- Definition of zip.
  op _zip_ : Powerlist{X} Powerlist{X} ~> Powerlist{X} .
  eq S zip S' = S tie S' [label def-zip-1].
  eq (P1* tie P2*) zip (Q1* tie Q2*)
   = (P1* zip Q1*) tie (P2* zip Q2*) [label def-zip-2].
  ... More operations
endfm
```

Figure 8.1: Powerlists in Maude

`Powerlist{Nat}`, `Scalar{Nat}`, `NsPowerlist*{Nat}`, and `Powerlist*{Nat}`. The `protecting` import will also import all of the operators, equations, and memberships in `POWERLIST` with the appropriate instantiated sorts.

One relevant feature of Maude is sort and operator *renaming*. Renaming is useful for the powerlist case study, because at present the ITP does not directly support parameterized theories. Moreover, we would like to reuse many of the basic powerlist lemmas across results for powerlists over different elements, including powerlists over natural numbers, complex numbers, and bits. To solve this problem, we use Maude's renaming feature to rename the sorts to be consistent each of the different types of elements. For example, we use the following import statement for defining powerlists over the natural numbers:

```
protecting POWERLIST{Nat}
 * (sort          Elt{Nat} to   Elt,
    sort       Scalar{Nat} to   Scalar,
    sort  NsPowerlist{Nat} to NsPowerlist,
    sort    Powerlist{Nat} to   Powerlist,
    sort NsPowerlist*{Nat} to NsPowerlist*,
    sort   Powerlist*{Nat} to   Powerlist*) .
```

For powerlists over complex numbers, we use the following module.

```
protecting POWERLIST{Complex}
 * (sort          Elt{Complex} to Elt,
    sort       Scalar{Complex} to Scalar,
    sort  NsPowerlist{Complex} to NsPowerlist,
    sort    Powerlist{Complex} to Powerlist,
    sort NsPowerlist*{Complex} to NsPowerlist*,
    sort   Powerlist*{Complex} to Powerlist*) .
```

These renamings map the sorts to the same syntactic name. Though just syntactic sugar, this naming convention allows us to easily define many powerlist theorems in a *reusable* way. Our ITP proof scripts has been grouped into files containing reusable lemmas that are often independent of the basic element type. This considerably reduces the number of lemmas that we must write out, and allows us to get many of the benefits of direct support for parameterized theories without the ITP actually supporting them at the moment.

8.2 Basic results

In later sections, we will show how to prove the correctness of advanced algorithms on powerlists, but we first build up a library of basic results that we will reuse in later proofs. In this section, we will prove properties about the similarity relation, the zip operation, the logarithm of the length of the powerlist, and operations for permuting the elements in different ways.

8.2.1 Similarity

One of the most fundamental defined relations on powerlists is the *similarity* relation, as it is fundamental in the definition of what a powerlist is. In our Maude `POWERLIST` module in Figure 8.1, the similarity relation is given by the recursively defined operation `sim?`.

The operator `sim?` represents an equivalence relation, but Maude does not allow rewriting over different equivalence relations. Prior to the new support for defining equivalence relations discussed in Section 7.4, we would have to prove a transitivity lemma, and then manually instantiate it whenever it was necessary. Fortunately, we can now have the lemma applied automatically by issuing the following command:

```
(defequiv sim? on Powerlist* .)
```

This command records that `sim?` is an equivalence relation over terms with sort `Powerlist*` and adds a reflexivity rule in the current goal. As `sim?` was declared to be commutative, the command generates two proof obligations: one to show that `sim?` is reflexive; the other to show that `sim?` is transitive.

```
==================================
label-sel: *er-sim?-reflex-Powerlist*@0
==================================
A{V0#0:Powerlist*}
 sim?(V0#0:Powerlist*,V0#0:Powerlist*) = true

==================================
label: *er-sim?-trans-Powerlist*@0
==================================
A{V0#0:Powerlist* ; V0#1:Powerlist* ; V0#2:Powerlist*}
   sim?(V0#2:Powerlist*,V0#1:Powerlist*) = true
 & sim?(V0#0:Powerlist*,V0#1:Powerlist*) = true
 ==> sim?(V0#0:Powerlist*,V0#2:Powerlist*) = true
```

The first subgoal `*er-sim?-reflex-Powerlist*@0` can be proven immediately with coverset induction.

```
(cov* on sim?(V0#0, V0#0) .)
```

Coverset induction actually generated three subgoals when this command was issued: one for elements, one for nested powerlists, and the third for concatenation. However, by using the `cov*` command, we automatically simplified each of the subgoals and discharged them automatically.

The second goal `*er-sim?-trans-Powerlist*@0` is more complex. It can be proven via coverset induction with splitting. In this case, coverset induction with splitting generates 20 separate subgoals. All but 2 subgoals are discharged automatically — the remaining two require manually instantiating the induction hypotheses for transitivity.

```
(cov-split* on sim?(V0#0, V0#2)
    split (sim?(V0#0, V0#1)) ; (sim?(V0#1, V0#2)) .)
--- Nesting subgoal.
(a-inst hyp-2 with (V0#1:Powerlist* <- V1#2*Powerlist*) .)
(auto .)
--- Concatenation subgoal.
(a-inst hyp-5 with (V0#1:Powerlist* <- V1#2*Powerlist*) .)
(auto .)
```

After proving this lemma, we will never need to manually instantiate transitivity of `sim?` again, as that will automatically be done by equivalence propagation.

We initially found it surprising that the `cov-split` command generated 20 separate subgoals for this problem. This seemed like a large number. However, in this case we are splitting on three different powerlist variables, and for each variable there are three basic cases: single elements, nested powerlists, and the concatenation of two powerlists. There are not $3^3 = 27$ different cases, because `sim?` has been defined so that some combinations of arguments do not require further splitting. For example, given the definition of `sim?`, splitting will not occur if one of the arguments has sort `NsPowerlist*` while the other has sort `Scalar` despite the fact that a scalar could be either a nesting or a single element.

As explained in Section 7.3.1, the number of subgoals generated depends on the strategy used during narrowing. An early version of coverset induction which did not implement the optimization described in 7.3.1 generated 31 subgoal cases. One culprit for the additional cases were the two subsort declarations:

```
subsort NsPowerlist{X} < NsPowerlist*{X} .
subsort Powerlist{X} < Powerlist*{X} .
```

These subsort declarations implicitly declare the memberships

$$x : \texttt{NsPowerlist*\{X\}} \textbf{ if } x : \texttt{NsPowerlist\{X\}}$$
$$x : \texttt{Powerlist*\{X\}} \textbf{ if } x : \texttt{Powerlist\{X\}}$$

In Maude, the memberships corresponding to subsort declarations are considered *constructor memberships* as defined in Chapter 5. Consequently, the narrowing algorithm considers them when expanding variables. Since the memberships obtained from these subsort declarations are implied by the other memberships, the cases they introduce in constructor splitting are redundant. However, we cannot omit these subsort declarations due to Maude's *preregularity* requirement — every term must have a least sort. One way to work around this is to give alternative constructor definitions which omit these memberships. The constructor memberships for `Powerlist*` follows quite easily.

```
(ctor-def plist* :
  A{P:Powerlist*}
   ((P) : NsPowerlist* V (P) : Scalar) .)
(eq-split* on (sim?(P, P)) .)
(set-default-ctor plist* .)
```

The last command sets `plist*` as the default constructor memberships.

For `NsPowerlist*` the proof is more complicated, because the ITP currently requires that existential quantifiers are manually instantiated using the `e-inst` command.

```
ctor-def nsplist* :
  A{NsP:NsPowerlist*}
    (E{P:Powerlist* ; Q:Powerlist*}
      ((P tie Q) = (NsP)
     & (sim?(P, Q)) = (true))) .)
(eq-split on (sim?(NsP, NsP)) .)
(cns .)
(imp .)
(e-inst
  with ((P:Powerlist* <- (V0#0*Powerlist*)) ;
        (Q:Powerlist* <- (V0#1*Powerlist*))) .)
(auto .)
(set-default-ctor nsplist* .)
```

This proof uses `eq-split` to instantiate `NsP*` with its constructor memberships, and then uses `cns` and `imp` to perform quantifier elimination and assume the implications. The existential quantifier is then eliminated, and the proof can be automatically proven.

We need one more lemma about similarity for later proofs. Specifically, we show that the similarity is a right congruence with respect to `tie`. As `sim?` is a function, this congruence lemma takes the following form:

```
(lem tie-r-sim* :
  A{P:Powerlist* ; Q1:Powerlist* ; Q2:Powerlist* ; R:Powerlist*}
   ((sim?(P, Q2)) = (true)
    & (sim?(Q1, Q2)) = (true)
     => (sim?(P tie Q1, R)) = (sim?(P tie Q2, R))) .)
(eq-split* on sim?(P tie Q1, R) .)
```

The rule formed from this lemma contains a free variable `Q2*` that does not appear in the left-hand side. Consequently, this rule is not directly executable by the ITP. Fortunately, we only need to do this twice in all of the proofs in this chapter. However, in the future it may be worth investigating other techniques for handling congruence rules.

8.2.2 Zip and unzip

We now turn our attention to another fundamental powerlist operation: the *zip* operator which is defined as follows:

```
  op _zip_ : Powerlist{X} Powerlist{X} ~> Powerlist{X} .
  eq S zip S' = S tie S' [label def-zip-1].
  eq (P1* tie P2*) zip (Q1* tie Q2*)
   = (P1* zip Q1*) tie (P2* zip Q2*) [label def-zip-2].
```

The two equations are labeled so that we can enable and disable them in proofs. We use `~>` in the operator declaration instead of `->` to indicate that the operator is partial. Consequently, the operator declaration for `zip` does not implicitly declare a membership stating that the zip of two powerlists is a powerlist.

Given the definition, our first task is to define a lemma stating when `zip` is well-defined, and when the `zip` of two powerlists is similar to another. This is done in the following lemma appearing in `lem-zip-sim.itp`:

```
(lem zip-sim* :
  A{P:Powerlist* ; Q:Powerlist* ; R:Powerlist*}
   ((sim?(P, Q)) = (true)
    => (P zip Q) : NsPowerlist*
     & ((sim?(P zip Q, R)) = (sim?(P tie Q, R)))) .)
(cov-split* on P zip Q split (sim?(P tie Q, R)) .)
--- Induction case 8.2:
(a-inst lem-tie-r-sim*
  with (Q1:Powerlist* <- V0#1*Powerlist*) ;
       (Q2:Powerlist* <- V0#5*Powerlist*) .)
(auto .)
```

By proving this lemma, the following two inference rules are automatically added to the current goal:

```
  cmb P zip Q : NsPowerlist* if sim?(P, Q) = true
    [label lem-zip-sim* metadata "|cat:lem|"].
  ceq sim?(P zip Q, R) = sim?(P tie Q, R) if sim?(P, Q) = true
    [label lem-zip-sim*2 metadata "|cat:lem|"].
```

Observe that the ITP is capable of automatically parsing this lemma as two separate rules, and that they are labeled with different names. This enhanced parsing was one of the features added to the ITP in this work. These rules may be disabled, and are labeled with the `metadata` attribute `"|cat:lem|"` so that they are not treated as axioms for generating induction schemes. In the future, we will not explicitly state the rules that a lemma introduces, because they are straightforward to derive from the lemma itself.

The previous lemma shows that zipping two similar powerlists together yields a non-scalar powerlist. For unnested powerlists, we prove the following theorem:

```
(lem zip-sim :
  A{P:Powerlist ; Q:Powerlist}
   ((sim?(P, Q)) = (true) => (P zip Q) : NsPowerlist) .)
(cov* on P zip Q .)
```

A major goal in this section is to show that every non-scalar powerlist can be expressed as the zip of two similar powerlists. This will be done through an alternative constructor definition, which will require two operations for *unzipping powerlists*:

```
  --- Left powerlist obtained by unzipping.
  op unzip-l : NsPowerlist*{X} -> Powerlist*{X} .
  eq unzip-l(S tie S') = S .
  eq unzip-l(NsP* tie NsQ*) = unzip-l(NsP*) tie unzip-l(NsQ*) .
  --- Right powerlist obtained by unzipping.
  op unzip-r : NsPowerlist*{X} -> Powerlist*{X} .
  eq unzip-r(S tie S') = S' .
  eq unzip-r(NsP* tie NsQ*) = unzip-r(NsP*) tie unzip-r(NsQ*) .
```

These two functions recursive deinterleave a non-scalar powerlist. The correctness for `unzip-l` for powerlists over natural numbers is shown in the following theorems in `unzip-l.itp`:

```
(goal unzip-l-zip : POWERLIST-NAT
  |- A{P:Powerlist* ; Q:Powerlist*}
      ((sim?(P, Q)) = (true)
        => (unzip-l(P zip Q)) = (P)) .)
load lem-sim-basics.itp
load lem-zip-sim.itp
(cov* on (P zip Q) .)
```

The correctness proof for `unzip-r` is virtually identical:

```
(goal unzip-r-zip : POWERLIST-NAT
  |- A{P:Powerlist* ; Q:Powerlist*}
      ((sim?(P, Q)) = (true)
        => (unzip-r(P zip Q)) = (Q)) .)
load lem-sim-basics.itp
load lem-zip-sim.itp
(cov* on (P zip Q) .)
```

We next show that every non-scalar powerlist can be represented as the zip of two powerlists. Showing this requires the previous lemmas on zip as well as the following two lemmas:

```
(lem zip-unzip :
  A{NsP:NsPowerlist*}
   ((unzip-l(NsP) zip unzip-r(NsP)) = (NsP)) .)
(cov* on (unzip-l(NsP)) .)


(lem sim-unzip-l-unzip-r-1 :
  A{NsP:NsPowerlist*}
   ((sim?(unzip-l(NsP), unzip-r(NsP))) = (true)) .)
(cov* on (unzip-l(NsP)) .)
```

The alternative constructor definitions for nested and unnested powerlists follow easily, but require manually instantiating the existentially quantified variables. This is done with the `e-inst` command and using the `unzip-l` and `unzip-r` operators to obtain the left and right sides of the `zip` operator. The script for defining the alternative constructor memberships is given in Figure 8.2, and it creates the following alternative constructor memberships:

```
(ctor-def zip* :
  A{NsP:NsPowerlist*}
   (E{P:Powerlist* ; Q:Powerlist*}
     ((P zip Q) = (NsP) & (sim?(P, Q)) = (true))) .)
(cns .)
(e-inst with ((P:Powerlist* <- (unzip-l(NsP*NsPowerlist*))) ;
              (Q:Powerlist* <- (unzip-r(NsP*NsPowerlist*)))) .)
(auto .)

(ctor-def zip :
  A{NsP:NsPowerlist}
   (E{P:Powerlist ; Q:Powerlist}
     ((P zip Q) = (NsP) & (sim?(P, Q)) = (true))) .)
(cns .)
(e-inst with ((P:Powerlist <- (unzip-l(NsP*NsPowerlist))) ;
              (Q:Powerlist <- (unzip-r(NsP*NsPowerlist)))) .)
(auto .)
```

Figure 8.2: Alternative constructor definitions for `zip`

```
cmb P*:Powerlist* zip Q*:Powerlist* : NsPowerlist*
 if sim?(P*:Powerlist*, Q:Powerlist*) = true
cmb P:Powerlist zip Q:Powerlist : NsPowerlist
 if sim?(P:Powerlist, Q:Powerlist) = true
```

8.2.3 Lgl

As powerlists may only be combined if they have the same length, each powerlist must have length 2^n for some natural number $n \in \mathbb{N}$. Accordingly, we can define the operation `lgl` which computes the *logarithm* in base 2 of their length as follows:

```
op lgl : Powerlist*{X} -> Nat .
eq lgl(S) = 0 .  eq lgl(P* tie Q*) = s lgl(P*) .
```

This function will be useful later in Section 8.3. For now, we just show two basic results in the file `lem-lgl.itp`:

1. If two powerlists are similar, then the logarithm of their length is the same:

   ```
   (lem sim-lgl :
     A{P:Powerlist ; Q:Powerlist}
       ((sim?(P, Q)) = (true) => (lgl(P)) = (lgl(Q))) .)
   (cov* on sim?(P, Q) .)
   ```

2. The logarithm of the length of `zip lgl(P* zip Q*)` is the successor of the `lgl(P*)`.

```
fmod POWERLIST-PERMUTATIONS{X :: TRIV} is
  pr POWERLIST{X} .
  var N : Nat .
  var NsP* : NsPowerlist*{X} .
  var P* Q* : Powerlist*{X} .
  var S : Scalar{X} .

  --- Rotates elements in powerlist right by one.
  op rr : Powerlist*{X} -> Powerlist*{X} .
  eq rr(S) = S .
  eq rr(P* zip Q*) = rr(Q*) zip P* .
  --- Rotates elements in powerlist left by one.
  op rl : Powerlist*{X} -> Powerlist*{X} .
  eq rl(S) = S .
  eq rl(P* zip Q*) = Q* zip rl(P*) .
  --- Reverse powerlist.
  op rev : Powerlist*{X} -> Powerlist*{X} .
  eq rev(S) = S .
  eq rev(P* tie Q*) = rev(Q*) tie rev(P*) .

  --- Rotates bitstring of indices right by one bit.
  op rs : Powerlist*{X} -> Powerlist*{X} .
  eq rs(S) = S .
  eq rs(P* zip Q*) = P* tie Q* .
  --- Rotates bitstring of indices left by one bit.
  op ls : Powerlist*{X} -> Powerlist*{X} .
  eq ls(S) = S .
  eq ls(P* tie Q*) = P* zip Q* .
  --- Inverts bitstring of indices.
  op inv : Powerlist*{X} -> Powerlist*{X} .
  eq inv(S) = S .
  eq inv(P* tie Q*) = inv(P*) zip inv(Q*) .
endfm
```

Figure 8.3: Operations for permuting powerlists

```
(lem lgl-zip :
  A{P:Powerlist* ; Q:Powerlist*}
   ((sim?(P, Q)) = (true)
    => (lgl(P zip Q)) = (s lgl(P))) .)
(cov* on P zip Q .)
```

8.2.4 Permutations

We conclude this section with six functions for permuting elements of powerlists. Three of the functions are best viewed as permuting the *elements* of a powerlist, while the other three functions are best viewed as permuting the *indices* of the powerlist. Both types of permutation can be naturally expressed in the powerlist notation. The six functions are defined in the module `POWERLIST-PERMUTATIONS` presented in Figure 8.3.

As an example to illustrate these operations, consider the following powerlist

P and its indices taken from [113] over letters in the alphabet:

$$\begin{array}{rl} \texttt{P} = [& a \quad b \quad c \quad d \quad e \quad f \quad g \quad h \ ] \\ \texttt{P's indices} = [& 000 \quad 100 \quad 010 \quad 110 \quad 001 \quad 101 \quad 011 \quad 111 \ ] \end{array}$$

The indices show the position of each element encoded as a bitstring with the least-significant bit first. The operations `rr` and `rl` rotate the elements to the right and left respectively.

$$\begin{array}{rl} \texttt{rr(P)} = [& h \quad a \quad b \quad c \quad d \quad e \quad f \quad g \ ] \\ \texttt{rr(P)'s indices} = [& 111 \quad 000 \quad 100 \quad 010 \quad 110 \quad 001 \quad 101 \quad 011 \ ] \\ \texttt{rl(P)} = [& b \quad c \quad d \quad e \quad f \quad g \quad h \quad a \ ] \\ \texttt{rl(P)'s indices} = [& 100 \quad 010 \quad 110 \quad 001 \quad 101 \quad 011 \quad 111 \quad 000 \ ] \end{array}$$

The indices show the original position of each element in the list.

We show in `rl-rr.itp` and `rr-rl.itp` that `rl` and `rr` are inverses.

```
(goal rl-rr : POWERLIST-PERMUTATIONS-NAT
  |- A{P:Powerlist*} ((rl(rr(P))) = (P)) .)
load lem-sim-basics.itp
load lem-zip-ctor.itp
(cov* using zip* on rr(P) .)

(goal rr-rl : POWERLIST-PERMUTATIONS-NAT
  |- A{P:Powerlist*}
    ((rr(rl(P))) = (P)) .)
load lem-sim-basics.itp
load lem-zip-ctor.itp
(cov* using zip* on rl(P) .)
```

where `POWERLIST-PERMUTATIONS-NAT` imports `POWERLIST-PERMUTATIONS` with the view `Nat`.

The operation `rev` reverse the elements.

$$\begin{array}{rl} \texttt{rev(P)} = [& h \quad g \quad f \quad e \quad d \quad c \quad b \quad a \ ] \\ \texttt{rev(P)'s indices} = [& 111 \quad 011 \quad 101 \quad 001 \quad 110 \quad 010 \quad 100 \quad 000 \ ] \end{array}$$

It is interesting to observe that `rev` toggles the bits in the indices between 0 and 1. In `lem-rev-basics.itp`, we show that `rev` commutes with `zip` and that `rev(P)` is similar to `P`. The proofs are straightforward coverset induction proofs with a couple of case splits in the second lemma.

```
(lem rev-zip :
  A{P*:Powerlist* ; Q*:Powerlist*}
   ((sim?(P*, Q*)) = (true)
     => (rev(P* zip Q*)) = (rev(Q*) zip rev(P*))) .)
(cov* on (P* zip Q*) .)
```

```
(lem sim?-rev :
  A{P*:Powerlist* ; Q*:Powerlist*}
    ((sim?(rev(P*), Q*)) = (sim?(P*, Q*))) .)
(cov-split* on rev(P*) split (sim?(P*, Q*)) .)
--- Induction case.
(split on (sim?(V0#0*Powerlist*,V0#3*Powerlist*)) .)
(auto .)
(split on (sim?(V0#0*Powerlist*,V0#1*Powerlist*)) .)
(auto .)
(auto .)
```

We also show that `rev` is a self-inverse in `lem-rev-rev.itp`, and the identity `rev(rr(rev(rr(P)))) = P` in `rev-rr-rev-rr.itp`. The proofs are quite straightforward, so we omit them here, but they are available in Appendix A.

The operations `rs` and `ls` can be viewed as rotating the bits in the bitstrings used to encode the indices to the right and left

$$\begin{aligned}
\texttt{rs(P)} &= [\ a \quad c \quad e \quad g \quad b \quad d \quad f \quad h\ ] \\
\texttt{rs(P)}\text{'s indices} &= [\ 000 \quad 010 \quad 001 \quad 011 \quad 100 \quad 110 \quad 101 \quad 111\ ] \\
\texttt{ls(P)} &= [\ a \quad e \quad b \quad f \quad c \quad g \quad d \quad h\ ] \\
\texttt{ls(P)}\text{'s indices} &= [\ 000 \quad 001 \quad 100 \quad 101 \quad 010 \quad 011 \quad 110 \quad 111\ ]
\end{aligned}$$

In `ls-rs.itp` and `rs-ls.itp`, we show that the operations `rs` and `ls` are inverses. The proof is quite straightforward and does not require coverset induction — only splitting.

Finally, the `inv` operation can be viewed as reversing the order of the indices.

$$\begin{aligned}
\texttt{inv(P)} &= [\ a \quad e \quad c \quad g \quad b \quad f \quad d \quad h\ ] \\
\texttt{inv(P)}\text{'s indices} &= [\ 000 \quad 001 \quad 010 \quad 011 \quad 100 \quad 101 \quad 110 \quad 111\ ]
\end{aligned}$$

In `lem-inv-basics.itp`, we show that `inv(P*)` is similar to `P*` and that `inv` can be defined over `zip` rather than `tie` as follows:

```
(lem inv-zip :
  A{P:Powerlist* ; Q:Powerlist*}
   ((sim?(P, Q)) = (true)
    => (inv(P zip Q)) = (inv(P) tie inv(Q))) .)
(cov* on (P zip Q) .)
```

This equivalence definition is used in `inv-inv.itp` to show that `inv` is a self inverse, i.e., `inv(inv(P)) = P`.

```
(goal inv-inv : POWERLIST-PERMUTATIONS-NAT
  |- A{P:Powerlist*}
    ((inv(inv(P))) = (P)) .)
load lem-sim-basics.itp
load lem-zip-sim.itp
load lem-inv-basics.itp
(cov* on inv(P*) .)
```

The lemmas in `lem-inv-basics.itp` are also used in conjunction with those in `lem-rev-basics.itp` to show in `inv-rev.itp` that `inv` and `rev` commute.

```
(goal inv-rev : POWERLIST-PERMUTATIONS-NAT
  |- A{P:Powerlist*}
    ((inv(rev(P))) = (rev(inv(P)))) .)
load lem-sim-basics.itp
load lem-zip-sim.itp
load lem-inv-basics.itp
load lem-rev-basics.itp
(cov* on rev(P*) .)
```

8.3 Fast Fourier transform

The Fast Fourier Transform is one of the most widely used algorithms in scientific computing. This is an efficient algorithm for computing the discrete Fourier transform — the evaluation of a polynomial P with n complex coefficients at the nth principal roots of 1. It was invented by Gauss, but rediscovered by James Cooley and John Tukey [37]. Misra showed in [113] how the Fast Fourier Transform (FFT) and inverse FFT can naturally be expressed as powerlist operators. He gave compact proofs for showing that the FFT defined over powerlists does indeed compute the discrete Fourier transform, and he also showed how to derive the inverse FFT directly from the definition of FFT.

Our goal in this section is to formally specify both the FFT and inverse FFT elegantly in Maude using Misra's approach, and also to develop proofs in the ITP for both the correctness of the FFT and inverse FFT. Before doing that, we first need to develop a reasonable approximation of complex numbers in Maude. We say an *approximation*, because there is no way to directly represent the uncountably many complex numbers as a term algebra in Maude.

Our approach is to represent a fragment of the complex numbers sufficient for defining the FFT operations. Specially, each complex number we represent can be expressed as the sum $(r_1, a_1) + \cdots + (r_n, a_n)$ of pairs of rational numbers. Each pair (r_j, a_j) in the sum represents the complex number

$$z_j = r_j \cdot (\cos(2\pi a_j) + i \cdot \sin(2\pi a_j)).$$

Essentially, the pair (r_j, a_j) is a complex number in polar coordinates where r_j is the *radius* and a_j corresponds to *revolutions* around the origin rather than *radians*. Not every complex number can be represented in this way. However, in our proofs we only rely on a few basic equational properties of these numbers, and these equational properties are also complex numbers.

In Figure 8.4, we present the Maude theory for defining complex numbers and a few basic operations. In order to define a valid domain for division, we introduce an additional sort `NzPolar` containing pairs (r, a) where r is a non-zero

```
fmod COMPLEX is protecting RAT .
  sort NzPolar Complex .
  subsort NzPolar < Complex .

  --- A complex number is represented as a sum of numbers in polar
  --- form (R, A) form where the angle A is measured in revolutions.
  --- The representation is normalized so that A is in [0,1/2) and
  --- addition satisfies certain basic identities.
  op ((_,_)) : NzRat Rat -> NzPolar [ctor].
  op ((_,_)) : Rat Rat -> Complex [ctor].
  op _+_ : Complex Complex -> Complex [ctor assoc comm prec 33].

  var R A R1 A1 R2 A2 : Rat .
  var NzR : NzRat .
  var C C1 C2 : Complex .
  var N : Nat .

  --- Basic identities.
  eq (0, NzR) = (0, 0) .
  ceq (R, A) = (R, A rem 1 + 1) if A < 0  .
  ceq (R, A) = (R, A rem 1) if 1 <= A .
  ceq (R, A) = (- R, A - 1/2) if 1/2 <= A /\ A <= 1 .
  eq (0, 0) + C = C .
  eq (R1, A) + (R2, A) = (R1 + R2, A) .

  --- Multiplication over complex numbers.
  op _*_ : Complex Complex -> Complex [assoc comm prec 31].
  eq (R1, A1) * (R2, A2) = (R1 * R2,  A1 + A2) .
  eq (C1 + C2) * C = C1 * C + C2 * C .
  --- Division operator over NzPolar numbers.
  op _/_ : Complex NzPolar -> Complex [prec 31].
  eq (R1, A1) / (NzR, A2) = (R1 / NzR,  _-_(A1, A2)) .
  eq (C1 + C2) / C = (C1 / C) + (C2 / C) .
  --- root(n) denotes the 2^nth principle root of 1.
  op root : Nat -> NzPolar .
  eq root(N) =  (1, 1 / 2 ^ N) [label def-root].
endfm
```

Figure 8.4: Complex numbers in Maude

rational number. This extra sort is used so that we can declare that division over `NzPolar` numbers is a total operation.

In `lem-fft-basics.itp`, we prove several lemmas about complex numbers that are useful for later proofs. These lemmas are proven by structural induction on the terms used to define our subset of complex numbers, but they are also true for all complex numbers. It turns out that the equation `def-root` which we use to define the `root` operation is rarely useful in later proofs. Instead, we define the following two lemmas in `lem-fft-basics.itp` and then disable `def-root`:

```
(lem root-sn :
  A{N:Nat}
    ((root(s N) * root(s N)) = (root(N))) .)
(auto .)
(lem root-1 :
  ((root(1)) = ((-1, 0))) .)
(auto .)
(disable def-root .)
```

The other lemmas about complex numbers are standard arithmetic facts, so we omit them here, but they may be found in Appendix B.

To instantiate `POWERLIST` over the complex numbers, we define the view:

```
view Complex from TRIV to COMPLEX is
  sort Elt to Complex .
endv
```

Powerlists can then be defined by instantiating `POWERLIST` with the `Complex` view. We do this in the module `POWERLIST-COMPLEX` in Figure 8.5. This module imports `POWERLIST`, and defines operators for element-wise addition, multiplication, division, and subtraction in the obvious way. To define a total domain of the division operator, we define a sort `NzPolarPowerlist` for representing powerlists over elements with sort `NzPolar`. In addition, Maude's requirement that every term has a least sort, requires us to define the sorts `NzPolarElt` for single element powerlists and `NsNzPolarPowerlist` for multiple element powerlists.

We prove several different elementary theorems about these operations in `lem-fft-basics.itp`. The results follow trivially by coverset induction, but they include: (1) basic facts about the similarity relation over the elementwise arithmetic operations, (2) various forms of associativity, distributivity, and identity axioms between addition, multiplication, and division, (3) alternative definitions of the elementwise arithmetic operations in terms of `zip` rather than tie. In order to avoid cluttering this chapter, we refer the interested reader to the Appendix B for the complete proof scripts.

Having defined basic operations over powerlists of complex numbers, we now turn our attention to defining the discrete Fourier transform. We first present a way of representing polynomials as powerlists. The basic idea in [113] is that

```
fmod POWERLIST-COMPLEX is
  protecting POWERLIST{Complex}
   * (sort          Elt{Complex} to Elt,
      sort       Scalar{Complex} to Scalar,
      sort  NsPowerlist{Complex} to NsPowerlist,
      sort    Powerlist{Complex} to Powerlist,
      sort NsPowerlist*{Complex} to NsPowerlist*,
      sort   Powerlist*{Complex} to Powerlist*) .

  sorts NzPolarElt NsNzPolarPowerlist NzPolarPowerlist .
  subsort NzPolarElt NsNzPolarPowerlist < NzPolarPowerlist .
  subsort NzPolarElt < Elt .
  subsort NsNzPolarPowerlist < NsPowerlist .
  subsort NzPolarPowerlist < Powerlist .

  var P P1 P2 Q Q1 Q2 : Powerlist .
  var C C' : Complex .
  var NzPol : NzPolar .
  var NzP NzQ : NzPolarPowerlist .

  op [_] : NzPolar -> NzPolarElt [ctor].
  cmb NzP tie NzQ : NsNzPolarPowerlist if sim?(NzP, NzQ) .

  --- Multiply elements in powerlist by scalar.
  op _*_ : Complex Powerlist -> Powerlist [prec 31].
  eq C * [ C' ] = [ C * C' ] .
  eq C * (P tie Q) = (C * P) tie (C * Q) .
  --- Elementwise multiplication.
  op _*_ : Powerlist Powerlist ~> Powerlist [prec 31].
  eq [ C ] * [ C' ] = [ C * C' ] .
  eq (P1 tie P2) * (Q1 tie Q2) = (P1 * Q1) tie (P2 * Q2) .
  --- Elementwise addition.
  op _+_ : Powerlist Powerlist ~> Powerlist [assoc comm prec 33].
  eq [ C ] + [ C' ] = [ C + C' ] .
  eq (P1 tie P2) + (Q1 tie Q2) = (P1 + Q1) tie (P2 + Q2) .
  --- Elementwise subtraction
  op _-_ : Powerlist Powerlist ~> Powerlist [prec 33].
  eq P - Q = P + (-1, 0) * Q .
  --- Elementwise division.
  op _/_  : Powerlist NzPolarPowerlist ~> Powerlist [prec 31].
  eq [ C ] / [ NzPol ] = [ C / NzPol ] .
  eq (P tie Q) / (NzP tie NzQ) = (P / NzP) tie (Q / NzQ) .
endfm
```

Figure 8.5: Powerlists over complex numbers in Maude

a polynomial with $m = 2^n$ coefficients

$$P = p_0 + p_1 x + \cdots + p_{m-1} x^{m-1}$$

can be represented by a powerlist $\texttt{P} = [\, p_0, p_1, \ldots, p_{m-1} \,]$.

For a complex number c, we can evaluate $P[c]$ using the operation `eval` defined below:

```
op eval : Powerlist Complex -> Complex .
eq eval([ C ], C') = C .
eq eval(P zip Q, C) = eval(P, C * C) + C * eval(Q, C * C) .
```

Furthermore, we can extend `eval` to evaluate `P` at a powerlist of points $Q = [\, q_1, \ldots, q_{m-1} \,]$ with the following operation extending `eval` to powerlists:

```
op eval : Powerlist Powerlist -> Powerlist .
eq eval(P, Q1 tie Q2) = eval(P, Q1) tie eval(P, Q2) .
eq eval(P, [ C ]) = [ eval(P, C) ] .
```

Finally, for each $n \in \mathbb{N}$ and complex number c, we let powers(n, c) be the powerlist

$$\text{powers}(n, c) = [\, c^0, c^1, \ldots c^{2^n - 1} \,]$$

This operation can be defined recursively in Maude as follows:

```
op powers : Nat Complex -> Powerlist .
eq powers(0, C) = [ (1, 0) ] .
eq powers(s N, C) = powers(N, C * C) zip (C * powers(N, C * C)) .
```

To define the discrete Fourier transform, we must evaluate a powerlist with length 2^n at the 2^nth principle roots of 1 which we denoted by root(n) in the module `COMPLEX` in Figure 8.4. For each $n \in \mathbb{N}$, we let $w(n)$ denote this list:

```
op w : Nat -> Powerlist .
eq w(N) = powers(N, root(N)) [label def-w].
```

The discrete Fourier transform can then be defined in Maude with the operation:

```
op ft : Powerlist -> Powerlist .
eq ft(P) = eval(P, w(lgl(P))) .
```

This definition is correct, but does not capture how the FFT is able to more efficiently compute the discrete Fourier transform. The FFT is a divide and conquer based algorithm which computes the discrete Fourier transform of `fft(P zip Q)` by first computing the FFT of `P` and `Q` separately, and then merging the results with some basic arithmetic operations. Before defining the FFT in Maude, we first must define the powerlist `u(N)` whose elements are the square root of `w(N)`. Specifically,

```
op u : Nat -> NzPolarPowerlist .
eq u(N) = powers(N, root(s N)) [label def-u].
```

```
(lem sim?-u :
  A{P:Powerlist ; Q:Powerlist}
    ((sim?(u(lgl(P)), Q)) = (sim?(P, Q))) .)
(auto .)

(lem sim?-w :
  A{P:Powerlist ; Q:Powerlist}
    ((sim?(w(lgl(P)), Q)) = (sim?(P, Q))) .)
(auto .)

(lem lgl-u :
  A{N:Nat} ((lgl(u(N))) = (N)) .)
(auto .)

(lem u-squared :
  A{N:Nat} ((u(N) * u(N)) = (w(N))) .)
(ind* on N .)

(lem w-tie :
  A{N:Nat} ((w(s N)) = (u(N) tie ((-1, 0) * u(N)))) .)
(ind* on N .)

(lem w-0 :
  (w(0)) = ([ (1, 0) ]) .)
(auto .)

(disable def-u .)
(disable def-w .)
```

Figure 8.6: Lemmas for `u(N)` and `w(N)` in `lem-fft-basics.itp`

For later results, the equations `def-w` and `def-u` for defining `w` and `u` actually make the proofs much more difficult. Our approach is to prove the lemmas shown in Figure 8.6, and then disable `def-w` and `def-u`.

Having defined `u`, we can define FFT recursively as follows:

```
op fft : Powerlist -> Powerlist .
eq fft([ C ]) = [ C ] .
eq fft(P zip Q)
 = (fft(P) + u(lgl(P)) * fft(Q)) tie (fft(P) - u(lgl(P)) * fft(Q)) .
```

Recall that the operation `lgl(P)` denote the log base 2 of the length of `P`.

Our first task with the ITP is to show that `fft` is the same function as `ft`. To show this, we first need to prove some basic facts about the function `eval` used to evaluate polynomials. First we show that `eval(P,Q)` is similar to `Q`.

```
(lem sim?-eval :
  A{P:Powerlist ; Q:Powerlist ; R:Powerlist}
   ((sim?(eval(P, Q), R)) = (sim?(Q, R))) .)
(cov-split* on eval(P, Q) split (sim?(Q, R)) .)
```

Next we need to define `eval` in terms of `zip` rather than `tie`. We need this because `fft` is defined in terms of `zip`. Due to the our previous lemmas, alternative definitions for `eval` in terms of `zip` are quite easy to show:

```
(lem eval-zip-right :
  A{P:Powerlist ; Q:Powerlist ; R:Powerlist}
   ((sim?(P, Q)) = (true)
    => (eval(R, P zip Q)) = (eval(R, P) zip eval(R, Q))) .)
(cov* on P + Q .)


(lem eval-zip-left :
  A{P:Powerlist ; Q:Powerlist ; R:Powerlist}
    ((sim?(P:Powerlist, Q:Powerlist)) = (true)
     => (eval(P zip Q, R)) = (eval(P, R * R) + R * eval(Q, R * R))) .)
(cov* on R * R .)
```

These results about `eval` allow us to show our goal quite easily.

```
(goal fft-ft : FFT
  |- A{P:Powerlist}
    ((fft(P)) = (ft(P))) .)
... Previously shown lemmas
--- Main theorem:
(cov* using zip on fft(P:Powerlist) .)
--- Induction case:
(a-inst lem-sim-lgl
  with (P:Powerlist <- V0#0*Powerlist) ;
       (Q:Powerlist <- V0#1*Powerlist) .)
(auto .)
```

The lemma `lem-sim-lgl` was shown in Section 8.2.3. It states that if `P` and `Q` are similar, then `lgl(P) = lgl(Q)`. The full proof may be found in `fft-ft.itp` in Appendix B.

The inverse FFT can also be expressed as a recursive operation in Maude:

```
op ift : Powerlist -> Powerlist .
eq ift([ C ]) = [ C ] .
eq ift(P tie Q)
 =     ift((1 / 2, 0) * (P + Q))
   zip ift((1 / 2, 0) * ((P - Q) / u(lgl(P)))) .
```

The fact that `fft` and `ift` are inverses follows directly from coverset induction with the previous lemmas:

```
(goal fft-ift : FFT
  |- A{P:Powerlist}
    ((fft(ift(P))) = (P)) .)
... Previously shown lemmas
--- Main theorem:
(cov* on ift(P) .)
(a-inst lem-sim-lgl
  with (P:Powerlist <- V0#0*Powerlist) ;
       (Q:Powerlist <- V0#1*Powerlist) .)
(auto .)

(goal ift-fft : FFT
  |- A{P:Powerlist}
    ((ift(fft(P))) = (P)) .)
... Previously shown lemmas
(cov* using zip on fft(P) .)
```

The full proofs are in `fft-ift.itp` and `ift-fft.itp`, and can be found in Appendix B.

8.4 Batcher sort

In this section, we develop a Maude specification for a parallel sorting algorithm developed by Batcher in [10]. The basic approach of this algorithm is to use a divide-and-conquer approach: a list is divided into two equal parts, which are sorted independently and then merged back together using a parallel merge algorithm called the Batcher merge.

The first challenge to defining a sorting algorithm on top of our Maude powerlist specification is that the elements in `TRIV` are unordered. Consequently, the elements are not directly comparable. To solve this problem, parameterized modules can be instantiated with a view from one theory to another. We can use this to instantiate `POWERLIST` to another parameterized module over a theory whose elements are ordered. The Maude prelude actually defines two such

```
fth TOTAL-PREORDER is
  protecting BOOL .
  including TRIV .
  op _<=_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
  eq X <= X = true [nonexec label reflexive] .
  ceq X <= Z = true if X <= Y /\ Y <= Z [nonexec label transitive] .
  eq X <= Y or Y <= X = true [nonexec label total] .
endfth

fth TOTAL-ORDER is
  including TOTAL-PREORDER .
  vars X Y : Elt .
  ceq X = Y if X <= Y /\ Y <= X  [nonexec label antisymmetric] .
endfth
```

Figure 8.7: Theories for total preorders and orders in Maude's prelude

theories, which appear in Figure 8.7. Both theories extend the trivial theory `TRIV` with an additional binary operator:

```
  op _<=_ : Elt Elt -> Bool .
```

The first theory, `TOTAL-PREORDER`, has non-executable equations that require `<=` to be a reflexive, transitive, total preordering. The second theory, `TOTAL-ORDER`, extends `TOTAL-PREORDER` with an additional non-executable equation requiring `<=` to be antisymmetric.

The prelude also defines a view with the overloaded name `TOTAL-PREORDER` from `TRIV` to `TOTAL-PREORDER`:

```
view TOTAL-PREORDER from TRIV to TOTAL-PREORDER is endv
```

If we instantiate `POWERLIST` with this view, we obtain a parameterized module `POWERLIST{TOTAL-PREORDER}` containing the powerlist module extended to have the parameter theory `TOTAL-PREORDER` rather than `TRIV`. For total orders, the prelude defines the view `TOTAL-ORDER` from `TOTAL-PREORDER` to `TOTAL-ORDER`.

```
view TOTAL-ORDER from TOTAL-PREORDER to TOTAL-ORDER is endv
```

The view `TOTAL-ORDER` can be used to map a parameterized module with the parameter `TOTAL-PREORDER` to the same parameterized module with `TOTAL-ORDER` as the parameter. Accordingly, `POWERLIST{TOTAL-PREORDER}{TOTAL-ORDER}` is the parameterized module with the `POWERLIST` definitions whose parameter theory is `TOTAL-ORDER`.

We define the Batcher merge and sort algorithms in the parameterized module `BATCHER-SORT` in Figure 8.8. This module captures the abstract nature of the sorting algorithm, and can be instantiated to sort powerlists over an arbitrary totally ordered set. The parametric sorts in `POWERLIST` are also instantiated with the views `TOTAL-PREORDER` and `TOTAL-ORDER`, so we use renaming

to reduce the clutter in their names. The key function is the Batcher merge operation `_bm_`. It merges two sorted powerlists `P1 zip P2` and `Q1 zip Q2` using divide-and-conquer. It recursively computes sorted lists $P =$ `P1` bm `Q2` and $Q =$ `P2` bm `Q1` that are mutually *interleaved*: That is, if we let $P = [\, p_1, \ldots, p_n \,]$ and $Q = [\, q_1, \ldots, q_n \,]$, we have that $\max(p_i, q_i) \leq \min(p_{i+1}, q_{i+1})$ for each $i \in [1, n)$. Batcher merge then returns the merger of P and Q, denoted by compare(P, Q). The correctness of this algorithm depends crucially on the property that P and Q are interleaved.

Maude's support for parameterized theories allows us to quite naturally express the Batcher sort algorithm. However, parameterized modules are not currently supported by the ITP, so there does not seem to be at present an easy way to mechanically prove that the Batcher's algorithm correctly sorts powerlists over an arbitrary totally order set. However, we are saved by one important observation made by Misra in [113]: Batcher sort is an example of a *compare and swap* sorting method, and such approaches to sorting are correct if they correctly sort lists only containing zero and one (see [95, Section 5.3.4]). To construct Maude powerlists over this type, we define the module `BIT` in Figure 8.9 with a sort `Bit` containing constants 0 and 1 and total ordering such that $0 \leq 1$.

One important feature of powerlists of bits is that there is an easy way to specify that a powerlist of bits is sorted using the zip operator. Specifically, a powerlist `P zip Q` is sorted if `P` and `Q` are sorted, and the number of zeros in `P` is either one less or equal to the number of zeros in `Q`. We define the sorted property in the module `POWERLIST-BIT-SORT` which appears in Figure 8.10.

This approach to defining whether a powerlist is sorted was also used in [113], and appears to be a key insight to elegantly showing the correctness of Batcher sort. We show in `sorted-bs.itp` the following theorem:

```
(goal sorted?-bs : POWERLIST-BIT-SORT
  |- A{P:Powerlist}
      ((sorted?(bs(P))) = (true)) .)
```

The lemmas in the ITP largely follow Misra's hand proof. We need several auxiliary lemmas (which we omit here) stating that for similar lists `P` and `Q`, the lists `min(P,Q)`, `max(P,Q)` and `bs(P)` are similar to `P`, and that `P bm Q` is similar to `P zip Q`. Most of these proofs follow automatically by coverset induction, although the proof that `P bm Q` is similar to `P zip Q` requires us to manually instantiate the congruence rule `tie-r-sim*` described in Section 8.2.1. In addition, we show several lemmas in `sorted-bs.itp` showing that for all sorted similar lists `P` and `Q`, both `P max Q` and `P min Q` are sorted, and

$$\texttt{zeros(P max Q)} = \texttt{min(zeros(P), zeros(Q))}$$
$$\texttt{zeros(P min Q)} = \texttt{max(zeros(P), zeros(Q))}$$

```
fmod BATCHER-SORT{X :: TOTAL-ORDER} is
  pr POWERLIST{TOTAL-PREORDER}{TOTAL-ORDER}{X} * (
   sort          Elt{TOTAL-PREORDER}{TOTAL-ORDER}{X} to          Elt{X},
   sort  NsPowerlist{TOTAL-PREORDER}{TOTAL-ORDER}{X} to  NsPowerlist{X},
   sort    Powerlist{TOTAL-PREORDER}{TOTAL-ORDER}{X} to    Powerlist{X},
   sort       Scalar{TOTAL-PREORDER}{TOTAL-ORDER}{X} to       Scalar{X},
   sort NsPowerlist*{TOTAL-PREORDER}{TOTAL-ORDER}{X} to NsPowerlist*{X},
   sort   Powerlist*{TOTAL-PREORDER}{TOTAL-ORDER}{X} to   Powerlist*{X}).

  var E E' : X$Elt .
  var P P1 P2 Q Q1 Q2 : Powerlist{X} .

  --- Take minimum of each element in arguments.
  op _min_ : Powerlist{X} Powerlist{X} ~> Powerlist{X} .
  eq [ E ] min [ E' ] = [ if E <= E' then E else E' fi ] .
  eq (P1 zip P2) min (Q1 zip Q2) = (P1 min Q1) zip (P2 min Q2) .

  --- Take maximum of each element in arguments.
  op _max_ : Powerlist{X} Powerlist{X} ~> Powerlist{X} .
  eq [ E ] max [ E' ] = [ if E <= E' then E' else E fi ] .
  eq (P1 zip P2) max (Q1 zip Q2) = (P1 max Q1) zip (P2 max Q2) .

  --- compare([p1 .. pn ], [q1 .. qn]) returns the powerlist:
  ---   [ min(p1, q1) max(p1,q1) .. min(pn, qn) max(pn,qn) ]
  op compare : Powerlist{X} Powerlist{X} ~> Powerlist{X} .
  eq compare(P, Q) = (P min Q) zip (P max Q) .

  --- Batcher merge
  op _bm_ : Powerlist{X} Powerlist{X} ~> Powerlist{X} .
  eq [ E ] bm [ E' ] = compare([ E ], [ E' ]) .
  eq (P1 zip P2) bm (Q1 zip Q2) = compare(P1 bm Q2, P2 bm Q1) .

  --- Sort using Batcher merge.
  op bs : Powerlist{X} -> Powerlist{X} .
  eq bs([ E ]) = [ E ] .
  eq bs(P zip Q) = bs(P) bm bs(Q) .
endfm
```

Figure 8.8: Batcher sort in Maude

```
fmod BIT is
  sort Bit .
  ops 0 1 : -> Bit [ctor].
  op _<=_ : Bit Bit -> Bool .
  eq B:Bit <= B:Bit = true .
  eq 0 <= 1 = true .
  eq 1 <= 0 = false .
endfm

view Bit<= from TOTAL-ORDER to BIT is
  sort Elt to Bit .
endv
```

Figure 8.9: Bits in Maude

```
fmod POWERLIST-BIT-SORT is
  pr BATCHER-SORT{Bit<=}
   * (sort          Elt{Bit<=} to   Elt,
      sort       Scalar{Bit<=} to   Scalar,
      sort  NsPowerlist{Bit<=} to NsPowerlist,
      sort    Powerlist{Bit<=} to   Powerlist,
      sort NsPowerlist*{Bit<=} to NsPowerlist*,
      sort   Powerlist*{Bit<=} to   Powerlist*) .
  var B : Bit .
  var P Q : Powerlist .

  --- Returns numbers of zeros in powerlist.
  op zeros : Powerlist -> Nat .
  eq zeros(P zip Q) = zeros(P) + zeros(Q) .
  eq zeros([ 0 ]) = 1 .
  eq zeros([ 1 ]) = 0 .

  op sorted? : Powerlist -> Bool .
  eq sorted?([ B ]) = true .
  eq sorted?(P zip Q)
   = sorted?(P) and
     sorted?(Q) and
     (zeros(Q) <=   zeros(P)) and
     (zeros(P) <= s zeros(Q)) .
endfm
```

Figure 8.10: Sorting bits in Maude

In Misra's work, a slightly different approach was used. His approach relied on an implicit assumption that `max` and `min` were commutative. We could add commutativity as an axiom to our operator declarations and replicate his approach, however our approach has a similar complexity and avoids the need to add additional axioms.

Unfortunately, as the ITP's decision procedure for linear arithmetic was incompatible with the other changes in Chapter 7, proving these lemmas was rather tedious. The proofs require some manual case splitting and instantiation of linear arithmetic facts. Collectively, the proofs total about 200 lines. We believe that the length of these proofs reflect just a current limitation in the implementation of the ITP, and not a major theoretical challenge. It appears that the manual quantifier instantiation and case splitting can be eliminated if an SMT-based theorem prover were to be integrated into the ITP as we propose in Chapter 7.

Together, these lemmas on `min` and `max` allow us to easily show that `bm` returns a sorted list given sorted lists `P` and `Q`, and that the number of zeros in `P bm Q` equals the total number of zeros in `P` and `Q`. The proof is fairly straightforward, but does require a few case splits.

```
(lem sorted?-bm :
  A{P:Powerlist ; Q:Powerlist}
   (   (sim?(P, Q)) = (true)
     & (sorted?(P)) = (true)
     & (sorted?(Q)) = (true)
    => (sorted?(P bm Q)) = (true)
     & (zeros(P bm Q)) = (zeros(P) + zeros(Q))) .)
(cov-split* using zip on P bm Q
     split (zeros(P)) ; (zeros(Q)) .)
--- First subcase
(split on zeros(V0#0*Powerlist bm V0#3*Powerlist)
       <= zeros(V0#2*Powerlist bm V0#1*Powerlist) .)
(auto .)
(a-inst lem-<=-false->true
   with (M:Nat <- (zeros(V0#2*Powerlist bm V0#1*Powerlist))) ;
        (N:Nat <- (zeros(V0#0*Powerlist bm V0#3*Powerlist))) .)
(auto .)
--- Second subcase
(split on zeros(V0#0*Powerlist bm V0#3*Powerlist)
       <= zeros(V0#2*Powerlist bm V0#1*Powerlist) .)
(auto .)
(auto .)
--- Third subcase
(split on zeros(V0#0*Powerlist bm V0#3*Powerlist)
       <= zeros(V0#2*Powerlist bm V0#1*Powerlist) .)
(auto .)
(auto .)
```

Given the previous lemmas, we are able to easily conclude with our main theorem:

```
(goal sorted?-bs : POWERLIST-BIT-SORT
  |- A{P:Powerlist}
      ((sorted?(bs(P))) = (true)) .)
... Previously shown lemmas
(cov* using zip on bs(P) .)
```

8.5 Ripple carry and carry lookahead adders

We conclude the powerlist case study with an examination of specifying adder circuits in Maude with powerlists. The basic idea is to represent the natural numbers in powerlists over bits using a binary representation, and then define addition over this representation. In this section we present two different addition algorithms: a *ripple carry adder* and a *carry lookahead adder*.

The approach we use to represent adders using powerlists comes from Adams in [1]. For powerlists with length n, each number less than 2^n may be uniquely encoded in a powerlist with length n over zero and one with the least-significant bit first. For example, the number 13 may be represented by the powerlist $[\,1, 0, 1, 1\,]$. The ripple carry adder adds elements one bit at a time. Let $Q = [\,q_0, \ldots, q_{n-1}\,]$ and $R = [\,r_0, \ldots, r_{n-1}\,]$ be powerlists that we wish to sum. This problem can be written out as follows:

$$\begin{array}{r} q_{n-1} \cdots q_1\ q_0 \\ r_{n-1} \cdots r_1\ r_0 \\ \hline ??? \end{array} \tag{8.1}$$

The ripple carry adder adds q_0 and r_0, computes the carry bit, and then procedures to add q_1 and r_1 with the previous carry bit. It adds each pair of bits sequentially until finishing with q_{n-1} and r_{n-1}.

The other type of adder we consider, the carry lookahead adder is able to be efficiently executed in parallel. The idea is that we can compute a powerlist containing each *carry bit* c_i for each pair of bits q_i and r_i using a *prefix computation* and then compute the sum for each following pair q_{i+1} and r_{i+1} independently by using the carry bit powerlist. To compute the carry bit, we observe that for each pair of bits q_i and r_i, one of three conditions will hold: (1) both q_i and r_i will be 0, and thus the carry bit c_i must be 0; (2) both q_i and r_i will be 1, and thus the carry bit c_i must be 1; or (3) q_i and r_i have different values 0 and 1, and thus the carry bit c_i will *propagate* the value of the previous carry bit c_{i-1}.

In the Maude module `CARRY` in Figure 8.5, we define a sort `Carry` for storing the three possible conditions that may hold for a given pair of bits q_i and r_i. The module uses 0 and 1 to represent that both q_i and r_i have the value 0 or 1 respectively, and uses the constant `p` to represent that they differ, and therefore

```
fmod CARRY is protecting BIT .
  sort Carry .
  subsort Bit < Carry .
  op p : -> Carry [ctor].

  var B : Bit .  var C : Carry .

  --- Compute carry for two bits.
  op carry : Bit Bit -> Carry .
  eq carry(B, B) = B .
  eq carry(0, 1) = p .
  eq carry(1, 0) = p .
  --- Combines carry bits together.
  op _!_ : Carry Carry -> Carry .
  eq C ! B = B .
  eq C ! p = C .
endfm
```

Figure 8.11: Maude module for carry values

propagate a carry value. The operator `carry` in Figure 8.5 computes the carry value c_i for given bits q_i and r_i. Carry values can be concatenated together. If we have two strings of bits $q_i q_{i+1}$ and $r_i r_{i+1}$ with carry values c_i and c_{i+1} respectively, then the carry value for the string $c_i c_{i+1}$, denoted $c_i ! c_{i+1}$, will be c_i if $c_{i+1} = \mathtt{p}$ and c_{i+1} otherwise. We prove in `rc-cl.itp` that the operator ! is associative, and that if c_i is 0 or 1, then $c_i ! c_{i+1}$ is 0 or 1.

The actual value for each carry bit used for adding q_i and r_i in the sum is the bit $v_i = 0 ! c_0 ! c_1 ! \ldots ! c_{i-1}$. We let V denote the powerlist $V = [\, v_0, \ldots, v_{n-1} \,]$. The powerlist V is called the *prefix sum* of the powerlist $C = [\, 0, c_0, \ldots, c_{n-2} \,]$ with respect to the associative operator !. The prefix sum can be computed sequentially by reading through the elements in C and accumulating each value v_i. We present a parameterized module for computing prefix sums over an arbitrary set and associative operator + in Figure 8.5. The definition we give in that figure is sequential; however, Misra showed in [113] how prefix sums can be computed in parallel over powerlists using the algorithm developed by Ladner and Fischer [97].

Due to some restrictions in Maude on overloaded operators imported in parameterized specifications, it is necessary to use a single import for powerlists over carry values that imports `PREFIX-SUM` for defining both powerlists over bits and carry values. We can then introduce a sorts for powerlists over bits with the appropriate constructors directly. This is done in the module `POWERLIST-NAT-ENCODING` in Figure 8.5, which also defines an operator `bv` for mapping natural numbers in the built-in representation to their corresponding powerlist representation, and an operator `eval` for evaluating the powerlists back into natural numbers.

In order to know the carry bit for the next inputs, the ripple carry adder

```
fth A-OP is
  inc TRIV .
  op _+_ : Elt Elt -> Elt .
  var X Y Z : Elt .
  eq X + (Y + Z) = (X + Y) + Z [nonexec].
endfth

view A-OP from TRIV to A-OP is endv

fmod PREFIX-SUM{X :: A-OP} is
  pr POWERLIST{A-OP}{X}
          * (sort          Elt{A-OP}{X} to          Elt{X},
             sort       Scalar{A-OP}{X} to       Scalar{X},
             sort  NsPowerlist{A-OP}{X} to  NsPowerlist{X},
             sort    Powerlist{A-OP}{X} to    Powerlist{X},
             sort NsPowerlist*{A-OP}{X} to NsPowerlist*{X},
             sort   Powerlist*{A-OP}{X} to   Powerlist*{X}) .

  var E E' : X$Elt .
  var P Q : Powerlist{X} .

  --- Apply + to all elements of powerlist.
  op _+_ : X$Elt Powerlist{X} -> Powerlist{X} .
  eq E + (P tie Q) = (E + P) tie (E + Q) .
  eq E + [ E' ] = [ E + E' ] .

  --- Prefix sum.
  op prefix : Powerlist{X} ~> Powerlist{X} .
  eq prefix([ E ])  = [ E ] .
  eq prefix(P tie Q) = prefix(P) tie (last(prefix(P)) + prefix(Q)) .
endfm
```

Figure 8.12: Parametrized prefix sum in Maude

```
fmod POWERLIST-NAT-ENCODING is
  protecting PREFIX-SUM{Carry}
   * (sort          Elt{Carry} to         Elt,
      sort       Scalar{Carry} to      Scalar,
      sort  NsPowerlist{Carry} to NsPowerlist,
      sort    Powerlist{Carry} to   Powerlist,
      sort NsPowerlist*{Carry} to NsPowerlist*,
      sort   Powerlist*{Carry} to   Powerlist*,
      op _+_ : Carry Powerlist{Carry} ~> Powerlist{Carry} to _!_) .

  sort Elt{Bit} NsPowerlist{Bit} Powerlist{Bit} .
  subsort Elt{Bit} NsPowerlist{Bit} < Powerlist{Bit} .
  subsort Elt{Bit} < Elt .
  subsort NsPowerlist{Bit} < NsPowerlist .
  subsort Powerlist{Bit} < Powerlist .

  var P Q : Powerlist{Bit} .
  var N Val : Nat .

  op [_] : Bit -> Elt{Bit} [ctor ditto].
  cmb P tie Q : NsPowerlist{Bit} if sim?(P, Q) .

  --- Returns least-significant bit of number.
  op lsb : Nat -> Bit .
  eq lsb(0) = 0 .
  eq lsb(1) = 1 .
  eq lsb(s s Val) = lsb(Val) .
  --- bv(N, Val) returns representation of Val in powerlist with lgl N.
  op bv : Nat Nat -> Powerlist{Bit} .
  eq bv(0, Val) = [ lsb(Val) ] .
  eq bv(s N,   Val) = bv(N, Val) tie bv(N, Val >> 2 ^ N) .
  --- Expands powerlist into natural number.
  op eval : Powerlist{Bit} -> Nat .
  eq eval([ 0 ]) = 0 .
  eq eval([ 1 ]) = 1 .
  eq eval(P tie Q) = eval(P) + (eval(Q) << (2 ^ lgl(P))) .
endfm
```

Figure 8.13: Representation of natural numbers as powerlists

outputs both the current sum and output carry bit. In the Maude module `ADDER`, we define the following sort for storing both these values:

```
sort AdderOutput .
op <_;_> : Bit Powerlist{Bit} -> AdderOutput [ctor].
```

We also define operators `carry` and `list`, which return the first and second values in a pair $< B ; P >$.

```
op carry : AdderOutput -> Bit .
eq carry(< B ; P >) = B .
op list : AdderOutput -> Powerlist{Bit} .
eq list(< B ; P >) = P .
```

Our ripple carry adder takes the previous carry bit and two powerlists to be added as inputs, and returns both the next carry bit and the sum of the inputs.

```
op rc : Bit Powerlist{Bit} Powerlist{Bit} ~> AdderOutput .
eq rc(0, [ 0 ], [ 0 ]) = < 0 ; [ 0 ] > .
eq rc(0, [ 0 ], [ 1 ]) = < 0 ; [ 1 ] > .
eq rc(0, [ 1 ], [ 0 ]) = < 0 ; [ 1 ] > .
eq rc(0, [ 1 ], [ 1 ]) = < 1 ; [ 0 ] > .
eq rc(1, [ 0 ], [ 0 ]) = < 0 ; [ 1 ] > .
eq rc(1, [ 0 ], [ 1 ]) = < 1 ; [ 0 ] > .
eq rc(1, [ 1 ], [ 0 ]) = < 1 ; [ 0 ] > .
eq rc(1, [ 1 ], [ 1 ]) = < 1 ; [ 1 ] > .
eq rc(B, P1 tie P2, Q1 tie Q2)
 = < carry(rc(carry(rc(B, P1, Q1)), P2, Q2)) ;
     list(rc(B, P1, Q1)) tie list(rc(carry(rc(B, P1, Q1)), P2, Q2)) > .
```

In `rc.itp`, we show that the ripple carry adder correctly implements modular addition.

```
(goal rc : ADDER
  |- A{B:Bit ; P:Powerlist‘{Bit‘} ; Q:Powerlist‘{Bit‘}}
      ((sim?(P, Q)) = (true)
       => (rc(B, P, Q))
        = (< lsb(eval([B]) + eval(P) + eval(Q) >> 2 ^ lgl(P)) ;
             bv(lgl(P), eval([B]) + eval(P) + eval(Q))>)) .)
```

This formula is complicated, due to the fact that `rc` implements addition modulo $n = 2^{2^{\mathrm{lgl}(P)}}$. If we let $m =$ `eval([b]) + eval(P) + eval(Q)`, then this theorem states that the carry bit returned by `rc` is the least-significant bit of m divided by n, while the bit-vector returned is the powerlist with length $2^{\mathrm{lgl}(P)}$ obtained by evaluating m.

We first need some of the previous lemmas about similarity, results about similarity of the powerlists returned by `bv` and `rc`, and basic arithmetic facts about the `lsb`, `bv`, and `eval` functions. We omit these lemmas and their proofs, because they turn out to be quite straightforward to show by coverset induction. The full details may be seen in Appendix D. The main goal, `rc`, follows by coverset induction, but requires two manual instantiations of `lem-sim-lgl`:

```
... Previous lemmas:
--- Main theorem
(cov* on rc(B, P, Q) .)
--- Last goal:
(a-inst lem-sim-lgl
  with (P:Powerlist <- V0#2*Powerlist‘{Bit‘}) ;
       (Q:Powerlist <- V0#1*Powerlist‘{Bit‘}) .)
(a-inst lem-sim-lgl
  with (P:Powerlist <- V0#3*Powerlist‘{Bit‘}) ;
       (Q:Powerlist <- V0#1*Powerlist‘{Bit‘}) .)
(auto .)
```

Our final result in this chapter is to define an operation that describes a carry lookahead adder and prove that it computes the same function as the ripple carry adder. We first extend the `carry` operation defined in `CARRY` to powerlists.

```
op carry : Powerlist{Bit} Powerlist{Bit} ~> Powerlist .
eq carry([ B ], [ B' ]) = [ carry(B, B') ] .
eq carry(P1 tie P2, Q1 tie Q2) = carry(P1, Q1) tie carry(P2, Q2) .
```

To compute the prefix sum over the carry bits, we define the operation `rsh` below to shift the elements off carry to the right by one position and insert a new value `C` as the left-most element.

```
op rsh : Carry Powerlist ~> Powerlist .
eq rsh(C, [ C' ]) = [ C ] .
eq rsh(C, PC tie QC) = rsh(C, PC) tie rsh(last(PC), QC) .
```

The prefix sum V used in summation is defined by

$$V = \texttt{prefix(rsh(0, carry(P, Q))))}.$$

After obtaining the prefix sum, we can compute the sum of similar powerlists P and Q by a function which accepts the prefix sum and the carry values `carry(P,Q)`. The idea is that for bits q and r, $\text{carry}(q,r) = 0$ iff $q + r = 0$, $\text{carry}(q,r) = \texttt{p}$ iff $q + r = 1$, and $\text{carry}(q,r) = 1$ iff $q + r = 2$. We use this to define the following function `sum`, which takes `prefix(rsh(0, carry(P, Q))))` and `carry(P,Q)` to compute the final sum.

```
op sum : Powerlist{Bit} Powerlist ~> Powerlist{Bit} .
eq sum(P tie Q, PC tie QC) = sum(P, PC) tie sum(Q, QC) .
eq sum([ B ], [ B' ]) = [ B ] .
eq sum([ 0 ], [ p  ]) = [ 1 ] .
eq sum([ 1 ], [ p  ]) = [ 0 ] .
```

The carry lookahead adder is then defined as follows:

```
op cl : Bit Powerlist{Bit} Powerlist{Bit} ~> AdderOutput .
eq cl(B, P, Q)
 = < last(prefix(rsh(B, carry(P, Q)))) ! last(carry(P, Q)) ;
     sum( prefix(rsh(B, carry(P, Q))),        carry(P, Q)) > .
```

Finally, we show that the ripple carry adder `rc` and carry lookahead adder `cl` implement the same function.

```
(goal rc-cl : ADDER
  |- A{B:Bit ; P:Powerlist‘{Bit‘} ; Q:Powerlist‘{Bit‘}}
      ((sim?(P,Q)) = (true)
       => (rc(B,P,Q)) = (cl(B,P,Q))) .)
```

The proof of this is non-trivial, but essentially involves showing that the two ways of computing the carry bits are equivalent. The proof requires several similarity and typing lemmas which we omit. It also requires that we show the associativity of `!` as well as a lemma stating that `!` commutes with `last`.

```
(lem !-left-assoc :
  A{C1:Carry ; C2:Carry ; C3:Carry}
   ((C1 ! (C2 ! C3)) = ((C1 ! C2) ! C3)) .)
(eq-split* on C2 ! C3 .)

(lem !-left-assoc-list :
  A{C:Carry ; C':Carry ; P:Powerlist}
   ((C ! (C' ! P)) = ((C ! C') ! P)) .)
(cov-split* on C' ! P split (C ! C') .)

(lem last-! :
  A{C:Carry ; P:Powerlist}
   ((last(C ! P)) = (C ! last(P))) .)
(cov* on C ! P .)
```

We also need to show how to move a carry concatenation operation inside of a prefix sum. Fortunately, the proofs are quite simple:

```
(lem last-prefix-rsh :
  A{C:Carry ; C':Carry ; P:Powerlist}
   ((C ! last(prefix(rsh(C', P)))) = (last(prefix(rsh(C ! C', P))))) .)
(cov* on C ! P .)

(lem prefix-rsh :
  A{C:Carry ; C':Carry ; P:Powerlist}
   ((C ! prefix(rsh(C', P))) = (prefix(rsh(C ! C', P)))) .)
(cov* on C ! P .)
```

Given the previous lemmas, showing the goal `rc-cl` involves a single coverset induction command.

```
(cov* on rc(B, P, Q) .)
```

8.6 Conclusions and related work

We have presented a detailed case study of Maude and the Maude ITP. Our experience so far suggests that membership equational logic is an excellent logic

for specifying powerlists, and that the ITP — with the enhancements discussed in the previous chapter — can provide much of the reasoning support to automate many of Misra's and Adam's elegant hand-proofs for powerlist algorithms. We have used the ITP to formally prove many of the main theorems and lemmas in both [1, 113], including basic powerlist lemmas, the correctness of the Fast Fourier Transform and inverse FFT, the correctness of Batcher sort, and the correctness of ripple carry and carry lookahead adder circuits.

There are already a number of papers on automated reasoning about powerlists. For the RRL theorem prover, Kapur and Subramaniam have proven results on the correctness of Batcher merge sort and hypercube embedding in [85]. They have also proven results on prefix sum, the ripple carry adder, and carry lookahead adder in [87]. For the ACL2 theorem prover, Gamboa has proven many of the same basic theorems we have, as well as the correctness of merge sort and Batcher sort for powerlists and results on prefix sums in [56, 57]. He has also proven many of the same results that we have proved for discrete Fourier transforms in [55]. Our primary research goal in this work is to not just to mechanically prove these results, but to do so in a *natural way* that reflects the elegance of the informal proofs in [1, 96, 113]. We believe that this work represents a major step towards that goal.

Although our results so far seem promising, there are several different research directions that seem worth pursuing. One research direction, that we mentioned at the end of Chapter 7, is to find ways to integrate the Maude ITP with existing theorem provers based on satisfiability modulo theories (SMT). These theorem provers handle disjunction quite well by case splitting, and support many different decisions procedures such as linear arithmetic which could be used to simplify the proofs in Section 8.4. Related to this would be to add non-standard analysis to the Maude ITP in order to fully model complex numbers. This was done by Gamboa and Cowles in ACL2 [58]. A third research direction would be to extend the work on syntactic rewriting modulo multiple equivalence relations in [22] for the ACL2 theorem prover to membership equational logic and the ITP. By implementing their ideas we could have the congruence equations such as `tie-r-sim*` automatically execute without manual instantiating the lemma as the ITP currently requires.

Another important research direction would be to find ways to automatically prove many of the various similarity lemmas that we had to prove. Although these lemmas were not difficult to prove, their large number meant that simply stating them often distracted from the fundamental results we wanted to prove. One approach to eliminating these lemmas may lay in extending the idea of *context-preserving rewriting* in [84] to membership equational logic. Another interesting approach may be to extend the decidability results in [34] on *Visibly Tree Automata with Memory* to develop a decidable type system capable of expressing the similarity constraints on powerlists.

Appendix A

Basic powerlist scripts

A.1 powerlist.maude

```
fmod POWERLIST{X :: TRIV} is protecting NAT .
  --- Sorts for unnested powerlists.
  sorts Elt{X} NsPowerlist{X} Powerlist{X} .
  subsort Elt{X} NsPowerlist{X} < Powerlist{X} .
  --- Sorts for nested powerlists.
  sorts Scalar{X} NsPowerlist*{X} Powerlist*{X} .
  subsort Scalar{X} NsPowerlist*{X} < Powerlist*{X} .
  --- Subsorts relating unnested and nested powerlists.
  subsort Elt{X} < Scalar{X} .
  subsort NsPowerlist{X} < NsPowerlist*{X} .
  subsort Powerlist{X} < Powerlist*{X} .

  --- Basic constructors.
  op [_] : X$Elt -> Elt{X} [ctor].
  op <_> : Powerlist*{X} -> Scalar{X} [ctor].
  op _tie_ : Powerlist*{X} Powerlist*{X} ~> Powerlist*{X} [prec 35].

  var E E' : X$Elt .
  var S S' : Scalar{X} .
  var P Q : Powerlist{X} .
  var P* P1* P2* Q* Q1* Q2* : Powerlist*{X} .
  var NsP* NsQ* : NsPowerlist*{X} .

  --- Similarity predicate.
  op sim? : Powerlist*{X} Powerlist*{X} -> Bool [comm].
  eq sim?([ E ], [ E' ])  = true .
  eq sim?(< P* >, [ E ]) = false .
  eq sim?(< P* >, < Q* >) = sim?(P*, Q*) .
  eq sim?(NsP*, S) = false .
  eq sim?(P1* tie P2*, Q1* tie Q2*) = sim?(P1*, Q1*) .

  --- Constructor memberships.
  cmb P tie Q : NsPowerlist{X} if sim?(P, Q) .
  cmb P* tie Q* : NsPowerlist*{X} if sim?(P*, Q*) .
```

```
  --- Definition of zip.
  op _zip_ : Powerlist{X} Powerlist{X} ~> Powerlist{X} .
  eq S zip S' = S tie S' [label def-zip-1].
  eq (P1* tie P2*) zip (Q1* tie Q2*)
   = (P1* zip Q1*) tie (P2* zip Q2*) [label def-zip-2].

  --- Returns the left powerlist obtained by unziping the argument.
  op unzip-l : NsPowerlist*{X} -> Powerlist*{X} .
  eq unzip-l(S tie S') = S .
  eq unzip-l(NsP* tie NsQ*) = unzip-l(NsP*) tie unzip-l(NsQ*) .

  --- Returns the right powerlist obtained by unziping the argument.
  op unzip-r : NsPowerlist*{X} -> Powerlist*{X} .
  eq unzip-r(S tie S') = S' .
  eq unzip-r(NsP* tie NsQ*) = unzip-r(NsP*) tie unzip-r(NsQ*) .

  --- Logarithm of length of Powerlist.
  op lgl : Powerlist*{X} -> Nat .
  eq lgl(S) = 0 .
  eq lgl(P* tie Q*) = s lgl(P*) .

  --- Returns first element in powerlist.
  op first : Powerlist{X} -> X$Elt .
  eq first([ E ]) = E .
  eq first(P tie Q) = first(P) .

  --- Returns last element in powerlist.
  op last : Powerlist{X} -> X$Elt .
  eq last([ E ]) = E .
  eq last(P tie Q) = last(Q) .
endfm

---(
Theorems:
 1. [lem-rev-basics.itp]
    rev(P* zip Q*)  = rev(Q*) zip rev(P*)
 2. [lem-rev-rev.itp]
    rev(rev(P*)) = P
 3. [lem-zip-ctor.itp]
    NsP* = unzip-l(NsP*) zip unzip-r(NsP*)
 4. [unzip-l-zip.itp]
    unzip-l(P1* zip P2*) == P1*.
 5. [unzip-r-zip.itp]
    unzip-r(P1* zip P2*) == P2*.
)
```

```
---  Functions over Powerlists.
fmod POWERLIST-PERMUTATIONS{X :: TRIV} is
  protecting POWERLIST{X} .

  var N : Nat .
  var NsP* : NsPowerlist*{X} .
  var P* Q* : Powerlist*{X} .
  var S : Scalar{X} .

  --- Rotates elements in powerlist right by one.
  op rr : Powerlist*{X} -> Powerlist*{X} .
  eq rr(S) = S .
  eq rr(P* zip Q*) = rr(Q*) zip P* .

  --- Rotates elements in powerlist left by one.
  op rl : Powerlist*{X} -> Powerlist*{X} .
  eq rl(S) = S .
  eq rl(P* zip Q*) = Q* zip rl(P*) .

  --- Rotates elements in powerlist right by an arbitrary value.
  op grr : Nat Powerlist*{X} -> Powerlist*{X} .
  eq grr(N, S) = S .
  eq grr(N, P* zip Q*)
   = if 2 divides N then
       grr(N >> 1, P*) zip grr(N >> 1, Q*)
     else
       grr((N >> 1) + 1, Q*) zip grr(N >> 1, P*)
     fi .

  op grl : Nat Powerlist*{X} -> Powerlist*{X} .
  eq grl(N, S) = S .
  eq grl(N, P* zip Q*)
   = if 2 divides N then
       grl(N >> 1, P*) zip grl(N >> 1, Q*)
     else
       grl(N >> 1, Q*) zip grl((N >> 1) + 1, P*)
     fi .

  op rs : Powerlist*{X} -> Powerlist*{X} .
  eq rs(S) = S .
  eq rs(P* zip Q*) = P* tie Q* .

  op ls : Powerlist*{X} -> Powerlist*{X} .
  eq ls(S) = S .
  eq ls(P* tie Q*) = P* zip Q* .

  --- Reverse powerlist.
```

```
  op rev : Powerlist*{X} -> Powerlist*{X} .
  eq rev(S) = S .
  eq rev(P* tie Q*) = rev(Q*) tie rev(P*) .

  op inv : Powerlist*{X} -> Powerlist*{X} .
  eq inv(S) = S .
  eq inv(P* tie Q*) = inv(P*) zip inv(Q*) .
endfm

---(Theorems:
--- rr and rl theorems:
 1. [rr-rl.itp]
    rr(rl(P)) = P
 2. [rl-rr.itp]
    rl(rr(P)) = P
 3. [rev-rr-rev-rr.itp]
    rev(rr(rev(rr(P)))) = P
--- rs and ls theorems:
 4. [rs-ls.itp]
    rs(ls(P)) = P
 5. [ls-rs.itp]
    ls(rs(P)) = P
--- inv theorems:
 6. [lem-inv-basics.itp]
    inv(P zip Q) = inv(P) tie inv(Q)
 7. [inv-inv.itp]
    inv(inv(P)) = P .
 8. [inv-rev.itp]
    inv(rev(P)) = rev(inv(P))
)
```

A.2 powerlist-nat.maude

```
load powerlist

fmod POWERLIST-NAT is
  protecting POWERLIST{Nat}
   * (sort          Elt{Nat} to   Elt,
      sort       Scalar{Nat} to   Scalar,
      sort  NsPowerlist{Nat} to NsPowerlist,
      sort    Powerlist{Nat} to   Powerlist,
      sort NsPowerlist*{Nat} to NsPowerlist*,
      sort   Powerlist*{Nat} to   Powerlist*) .
endfm

fmod POWERLIST-PERMUTATIONS-NAT is
```

```
  protecting POWERLIST-PERMUTATIONS{Nat}
   * (sort          Elt{Nat} to   Elt,
      sort       Scalar{Nat} to   Scalar,
      sort  NsPowerlist{Nat} to NsPowerlist,
      sort    Powerlist{Nat} to   Powerlist,
      sort NsPowerlist*{Nat} to NsPowerlist*,
      sort   Powerlist*{Nat} to   Powerlist*) .
endfm
```

A.3 lem-sim-basics.itp

```
(ctor-def plist* :
  A{P:Powerlist*}
   ((P) : NsPowerlist* V (P) : Scalar) .)
(eq-split* on (sim?(P, P)) .)
(set-default-ctor plist* .)

(ctor-def nsplist* :
  A{NsP:NsPowerlist*}
    (E{P:Powerlist* ; Q:Powerlist*}
      ((P tie Q) = (NsP)
     & (sim?(P, Q)) = (true))) .)
(eq-split on (sim?(NsP, NsP)) .)
(cns .)
(imp .)
(e-inst
  with ((P:Powerlist* <- (V0#0*Powerlist*)) ;
        (Q:Powerlist* <- (V0#1*Powerlist*))) .)
(auto .)
(set-default-ctor nsplist* .)

--- Define sym to be an equivalence relation on Powerlist*.
(defequiv sim? on Powerlist* .)
--- Reflexitivity goal:
(cov* on sim?(V0#0, V0#0) .)
--- Transitivity goal:
(cov-split* on sim?(V0#0, V0#2)
    split (sim?(V0#0, V0#1)) ; (sim?(V0#1, V0#2)) .)
--- Nesting subggoal.
(a-inst hyp-2 with (V0#1:Powerlist* <- V1#2*Powerlist*) .)
(auto .)
--- Concatenation subgoal.
(a-inst hyp-5 with (V0#1:Powerlist* <- V1#2*Powerlist*) .)
(auto .)

(lem tie-r-sim* :
```

```
  A{P:Powerlist* ; Q1:Powerlist* ; Q2:Powerlist* ; R:Powerlist*}
   ((sim?(P, Q2)) = (true)
    & (sim?(Q1, Q2)) = (true)
     => (sim?(P tie Q1, R)) = (sim?(P tie Q2, R))) .)
(eq-split* on sim?(P tie Q1, R) .)
```

A.4 lem-zip-ctor.itp

```
--- Requires lem-sim-basics.itp

(lem zip-unzip :
  A{NsP*:NsPowerlist*}
   ((unzip-l(NsP*) zip unzip-r(NsP*)) = (NsP*)) .)
(cov* on (unzip-l(NsP*)) .)

(lem sim-unzip-l-unzip-r-1 :
  A{NsP*:NsPowerlist*}
   ((sim?(unzip-l(NsP*), unzip-r(NsP*))) = (true)) .)
(cov* on (unzip-l(NsP*)) .)

(ctor-def zip* :
  A{NsP*:NsPowerlist*}
   (E{P*:Powerlist* ; Q*:Powerlist*}
     ((P* zip Q*) = (NsP*)
    & (sim?(P*, Q*)) = (true))) .)
(cns .)
(e-inst
  with ((P*:Powerlist* <- (unzip-l(NsP**NsPowerlist*))) ;
        (Q*:Powerlist* <- (unzip-r(NsP**NsPowerlist*)))) .)
(auto .)

(ctor-def zip :
  A{NsP:NsPowerlist}
   (E{P:Powerlist ; Q:Powerlist}
     ((P zip Q) = (NsP) & (sim?(P, Q)) = (true))) .)
(cns .)
(e-inst
  with ((P:Powerlist <- (unzip-l(NsP*NsPowerlist))) ;
        (Q:Powerlist <- (unzip-r(NsP*NsPowerlist)))) .)
(auto .)
```

A.5 lem-zip-sim.itp

```
--- Requires lem-sim-basics.itp

--- Reduce zip to tie inside sim?
```

```
(lem zip-sim* :
  A{P:Powerlist* ; Q:Powerlist* ; R:Powerlist*}
   ((sim?(P, Q)) = (true)
    => (P zip Q) : NsPowerlist*
     & ((sim?(P zip Q, R)) = (sim?(P tie Q, R)))) .)
(cov-split* on P zip Q split (sim?(P tie Q, R)) .)
--- Induction case 8.2:
(a-inst lem-tie-r-sim*
  with (Q1:Powerlist* <- V0#1*Powerlist*) ;
       (Q2:Powerlist* <- V0#5*Powerlist*) .)
(auto .)

(lem zip-sim :
  A{P:Powerlist ; Q:Powerlist}
   ((sim?(P, Q)) = (true) => (P zip Q) : NsPowerlist) .)
(cov* on P zip Q .)
```

A.6 unzip-l-zip.itp

```
load powerlist-nat

select ITP-TOOL .
loop init-itp(uiuc) .

(goal unzip-l-zip : POWERLIST-NAT
  |- A{P:Powerlist* ; Q:Powerlist*}
      ((sim?(P, Q)) = (true)
        => (unzip-l(P zip Q)) = (P)) .)

load lem-sim-basics.itp
load lem-zip-sim.itp
(cov* on (P zip Q) .)
```

A.7 unzip-r-zip.itp

```
load powerlist-nat

select ITP-TOOL .
loop init-itp(uiuc) .

(goal unzip-r-zip : POWERLIST-NAT
  |- A{P:Powerlist* ; Q:Powerlist*}
      ((sim?(P, Q)) = (true)
        => (unzip-r(P zip Q)) = (Q)) .)

load lem-sim-basics.itp
```

```
load lem-zip-sim.itp
(cov* on (P zip Q) .)
```

A.8 rl-rr.itp

```
load powerlist-nat

select ITP-TOOL .
loop init-itp(uiuc) .

(goal rl-rr : POWERLIST-PERMUTATIONS-NAT
  |- A{P*:Powerlist*}
    ((rl(rr(P*))) = (P*)) .)

load lem-sim-basics.itp
load lem-zip-ctor.itp

(cov* using zip* on rr(P*) .)
```

A.9 rr-rl.itp

```
load powerlist-nat

select ITP-TOOL .
loop init-itp(uiuc) .

(goal rr-rl : POWERLIST-PERMUTATIONS-NAT
  |- A{P*:Powerlist*}
    ((rr(rl(P*))) = (P*)) .)

load lem-sim-basics.itp
load lem-zip-ctor.itp

(cov* using zip* on rl(P*) .)
```

A.10 lem-rev-basics.itp

```
---- Requires lem-sim-basics.itp and lem-zip-sim.itp
(lem rev-zip :
  A{P*:Powerlist* ; Q*:Powerlist*}
   ((sim?(P*, Q*)) = (true)
     => (rev(P* zip Q*)) = (rev(Q*) zip rev(P*))) .)
(cov* on (P* zip Q*) .)

(lem sim?-rev :
```

```
  A{P*:Powerlist* ; Q*:Powerlist*}
    ((sim?(rev(P*), Q*)) = (sim?(P*, Q*))) .)
(cov-split* on rev(P*) split (sim?(P*, Q*)) .)
--- Induction case 7.1:
(split on (sim?(V0#0*Powerlist*,V0#3*Powerlist*)) .)
(auto .)
(split on (sim?(V0#0*Powerlist*,V0#1*Powerlist*)) .)
(auto .)
(auto .)
```

A.11 lem-rev-rev.itp

```
(lem rev-rev :
  A{P*:Powerlist*} ((rev(rev(P*))) = (P*)) .)
(cov* on (rev(P*:Powerlist*)) .)
```

A.12 rev-rr-rev-rr.itp

```
load powerlist-nat

select ITP-TOOL .
loop init-itp(uiuc) .

(goal rev-rr-rev-rr : POWERLIST-PERMUTATIONS-NAT
  |- A{P*:Powerlist*}
    ((rev(rr(rev(rr(P*))))) = (P*)) .)

load lem-sim-basics.itp
load lem-zip-ctor.itp
load lem-zip-sim.itp
load lem-rev-basics.itp
load lem-rev-rev.itp

(lem sim?-rr :
  A{P*:Powerlist* ; Q*:Powerlist*}
   ((sim?(rr(P*), Q*))
     = (sim?(P*, Q*))) .)
(cov-split* using zip* on rr(P*) split (rr(Q*)) .)
(split on (sim?(V0#0*Powerlist*,V0#1*Powerlist*)) .)
(auto .)
(split on (sim?(V0#1*Powerlist*,V0#2*Powerlist*)) .)
(auto .)
(auto .)

--- Goal:
(cov* using zip* on rr(P*) .)
```

A.13 ls-rs.itp

```
load powerlist-nat

select ITP-TOOL .
loop init-itp(uiuc) .

(goal ls-rs : POWERLIST-PERMUTATIONS-NAT
  |- A{P*:Powerlist*} ((ls(rs(P*))) = (P*)) .)

load lem-sim-basics.itp
load lem-zip-ctor.itp

(cov* using zip* on rs(P*:Powerlist*) .)
```

A.14 rs-ls.itp

```
load powerlist-nat

select ITP-TOOL .
loop init-itp(uiuc) .

(goal rs-ls : POWERLIST-PERMUTATIONS-NAT
  |- A{P*:Powerlist*}
    ((rs(ls(P*))) = (P*)) .)

(cov* on ls(P*:Powerlist*) .)
```

A.15 lem-inv-basics.itp

```
--- Requires lem-sim-basics.itp and lem-zip-sim.itp

(lem inv-zip :
  A{P*:Powerlist* ; Q*:Powerlist*}
   ((sim?(P*, Q*)) = (true)
    => (inv(P* zip Q*)) = (inv(P*) tie inv(Q*))) .)
(cov* on (P* zip Q*) .)

(lem sim?-inv :
  A{P*:Powerlist* ; R*:Powerlist*}
   ((sim?(inv(P*), R*)) = (sim?(P*, R*))) .)
(cov-split* on inv(P*)
     split (sim?(P*, R*)) .)
```

A.16 inv-inv.itp

```
load powerlist-nat

select ITP-TOOL .
loop init-itp(uiuc) .

(goal inv-inv : POWERLIST-PERMUTATIONS-NAT
  |- A{P*:Powerlist*}
    ((inv(inv(P*))) = (P*)) .)

load lem-sim-basics.itp
load lem-zip-sim.itp
load lem-inv-basics.itp

(cov* on inv(P*) .)
```

A.17 inv-rev.itp

```
load powerlist-nat

select ITP-TOOL .
loop init-itp(uiuc) .

(goal inv-rev : POWERLIST-PERMUTATIONS-NAT
  |- A{P*:Powerlist*}
    ((inv(rev(P*))) = (rev(inv(P*)))) .)

load lem-sim-basics.itp
load lem-zip-sim.itp
load lem-inv-basics.itp
load lem-rev-basics.itp

(cov* on rev(P*) .)
```

A.18 lem-lgl.itp

```
--- Requires lem-sim-basics.itp and lem-zip-sim.itp

(lem sim-lgl :
  A{P:Powerlist ; Q:Powerlist}
    ((sim?(P, Q)) = (true)
     => (lgl(P)) = (lgl(Q))) .)
(cov* on sim?(P, Q) .)

(lem lgl-zip :
```

```
  A{P:Powerlist* ; Q:Powerlist*}
   ((sim?(P, Q)) = (true)
    => (lgl(P zip Q)) = (s lgl(P))) .)
(cov* on P zip Q .)
```

Appendix B

Fast Fourier transform scripts

B.1 powerlist-fft.maude

```
load powerlist

fmod COMPLEX is protecting RAT .
  sort NzPolar Complex .
  subsort NzPolar < Complex .

  --- A complex number is represented as a sum of numbers in polar form
  --- (R, A) --- where the angle A is measured in revolutions.
  --- The representation is normalized so that A is in [0,1/2) and
  --- addition satisfies certain basic identites.
  op ((_,_)) : NzRat Rat -> NzPolar [ctor].
  op ((_,_)) : Rat Rat -> Complex [ctor].
  op _+_ : Complex Complex -> Complex [ctor assoc comm prec 33].

  var R A R1 A1 R2 A2 : Rat .
  var NzR : NzRat .
  var C C1 C2 : Complex .
  var N : Nat .

  --- Basic identities.
  eq (0, NzR) = (0, 0) .
  ceq (R, A) = (R, A rem 1 + 1) if A < 0  .
  ceq (R, A) = (R, A rem 1) if 1 <= A .
  ceq (R, A) = (- R, A - 1/2) if 1/2 <= A /\ A <= 1 .
  eq (0, 0) + C = C .
  eq (R1, A) + (R2, A) = (R1 + R2, A) .

  --- Multiplication over complex numbers.
  op _*_ : Complex Complex -> Complex [assoc comm prec 31].
  eq (R1, A1) * (R2, A2) = (R1 * R2,  A1 + A2) .
  eq (C1 + C2) * C = C1 * C + C2 * C .

  --- A partial division operator over complex numbers.
  op _/_ : Complex NzPolar -> Complex [prec 31].
```

```
  eq (R1, A1) / (NzR, A2) = (R1 / NzR,  _-_(A1, A2)) .
  eq (C1 + C2) / C = (C1 / C) + (C2 / C) .

  --- root(n) denotes the 2^nth principle root of 1.
  op root : Nat -> NzPolar .
  eq root(N) =  (1, 1 / 2 ^ N) [label def-root].
endfm

view Complex from TRIV to COMPLEX is
  sort Elt to Complex .
endv

fmod POWERLIST-COMPLEX is
  protecting POWERLIST{Complex}
   * (sort          Elt{Complex} to Elt,
      sort       Scalar{Complex} to Scalar,
      sort  NsPowerlist{Complex} to NsPowerlist,
      sort    Powerlist{Complex} to Powerlist,
      sort NsPowerlist*{Complex} to NsPowerlist*,
      sort   Powerlist*{Complex} to Powerlist*) .

  sorts NzPolarElt NsNzPolarPowerlist NzPolarPowerlist .
  subsort NzPolarElt NsNzPolarPowerlist < NzPolarPowerlist .
  subsort NzPolarElt < Elt .
  subsort NsNzPolarPowerlist < NsPowerlist .
  subsort NzPolarPowerlist < Powerlist .

  var P P1 P2 Q Q1 Q2 : Powerlist .
  var C C' : Complex .
  var NzPol : NzPolar .
  var NzP NzQ : NzPolarPowerlist .

  op [_] : NzPolar -> NzPolarElt [ctor].
  cmb NzP tie NzQ : NsNzPolarPowerlist if sim?(NzP, NzQ) .

  --- Multiply elements in powerlist by scalar.
  op _*_ : Complex Powerlist -> Powerlist [prec 31].
  eq C * [ C' ] = [ C * C' ] .
  eq C * (P tie Q) = (C * P) tie (C * Q) .
  --- Elementwise multiplication.
  op _*_ : Powerlist Powerlist ~> Powerlist [prec 31].
  eq [ C ] * [ C' ] = [ C * C' ] .
  eq (P1 tie P2) * (Q1 tie Q2) = (P1 * Q1) tie (P2 * Q2) .
  --- Elementwise addition.
  op _+_ : Powerlist Powerlist ~> Powerlist [assoc comm prec 33].
  eq [ C ] + [ C' ] = [ C + C' ] .
  eq (P1 tie P2) + (Q1 tie Q2) = (P1 + Q1) tie (P2 + Q2) .
```

```
  --- Elementwise subtraction
  op _-_ : Powerlist Powerlist ~> Powerlist [prec 33].
  eq P - Q = P + (-1, 0) * Q .
  --- Elementwise division.
  op _/_  : Powerlist NzPolarPowerlist ~> Powerlist [prec 31].
  eq [ C ] / [ NzPol ] = [ C / NzPol ] .
  eq (P tie Q) / (NzP tie NzQ) = (P / NzP) tie (Q / NzQ) .
endfm

fmod FFT is
  protecting POWERLIST-COMPLEX .

  var N : Nat .
  var C C' : Complex .
  var P Q Q1 Q2 : Powerlist .

  --- Evaluate powerlist representing a polynomial at a specific point.
  op eval : Powerlist Complex -> Complex .
  eq eval([ C ], C') = C .
  eq eval(P zip Q, C) = eval(P, C * C) + C * eval(Q, C * C) .

  --- Extend eval to evaluate polynomial at each point in powerlist.
  op eval : Powerlist Powerlist -> Powerlist .
  eq eval(P, Q1 tie Q2) = eval(P, Q1) tie eval(P, Q2) .
  eq eval(P, [ C ]) = [ eval(P, C) ] .

  --- powers(N,C) return powerlist [0 C ... C^{2^n-1}]
  op powers : Nat NzPolar -> NzPolarPowerlist .
  op powers : Nat Complex -> Powerlist .
  eq powers(0, C) = [ (1, 0) ] .
  eq powers(s N, C) = powers(N, C * C) zip (C * powers(N, C * C)).

  --- u(N) returns powerlist with length N and where the value of w(N) at
  --- position i times the 2^{n+1} principle root of 1.
  op u : Nat -> NzPolarPowerlist .
  eq u(N) = powers(N, root(s N)) [label def-u].

  --- w(N) returns powerlist with length N and where the value of w(N) at
  --- position i times the 2^n principle root of 1.
  op w : Nat -> Powerlist .
  eq w(N) = powers(N, root(N)) [label def-w].

  --- Discrete fourier transform.
  op ft : Powerlist -> Powerlist .
  eq ft(P) = eval(P, w(lgl(P))) .

  --- FFT on powerlist.
```

```
  op fft : Powerlist -> Powerlist .
  eq fft([ C ]) = [ C ] .
  eq fft(P zip Q)
   =     (fft(P) + u(lgl(P)) * fft(Q))
     tie (fft(P) - u(lgl(P)) * fft(Q)) .

  --- Inverse FFT
  op ift : Powerlist -> Powerlist .
  eq ift([ C ]) = [ C ] .
  eq ift(P tie Q)
   =     ift((1 / 2, 0) * (P + Q))
     zip ift((1 / 2, 0) * ((P - Q) / u(lgl(P)))) .
endfm

---(Theorems:
Theorems:
 1. [fft-ft.itp]
    fft(P) = ft(P)
 2. [ift-fft.itp]
    ift(fft(P)) = P
 3. [fft-ift.itp]
    fft(ift(P)) = P
)
```

B.2 lem-fft-basics.itp

```
--- Requires lem-sim-basics.itp, lem-zip-sim.itp, and lem-zip-ctor.itp

--- Declare constructor for Powerlist to discard
--- NzPolarPowerlist subsort.
(ctor-def plist :
  A{P:Powerlist}
   ((P) : NsPowerlist V (P) : Elt) .)
(eq-split* on (sim?(P, P)) .)
(set-default-ctor plist .)

(lem complex-*-0 :
  A{C:Complex}
    ((C * (0, 0)) = ((0, 0))) .)
(cov on (C * (0,0)) .)
(auto .)
(eq-split* on (0, V0#1) .)
(sort-ctor-split on (V1#0*Zero) .)
(auto .)

(lem complex-*-1 :
```

```
  A{C:Complex}
    ((C * (1, 0)) = (C)) .)
(cov* on (C * (1,0)) .)

(lem complex-*-2 :
  A{C:Complex}
    ((C * (2, 0)) = (C + C)) .)
(cov* on (C * (2,0)) .)

(lem collect-/-lem :
  A{C1:Complex ; C2:Complex ; NzR:NzRat ; A:Rat}
   ((C1 * (C2 / (NzR, A))) = ((C1 * C2) / (NzR, A))) .)
(cov* on (C1 * C2) .)

(lem collect-/ :
  A{C1:Complex ; C2:Complex ; NzP:NzPolar}
   ((C1 * (C2 / NzP)) = ((C1 * C2) / NzP)) .)
(eq-split* on NzP / NzP .)

(lem cancel-/ :
  A{C:Complex ; NzP:NzPolar}
   (((NzP * C) / NzP) = (C)) .)
(cov* on C / NzP .)

(lem complex-/-1 :
  A{C:Complex}
    ((C / (1, 0)) = (C)) .)
(cov* on (C / (1,0)) .)

------------------------------------------------------------------------
--- root
------------------------------------------------------------------------
(lem root-sn :
  A{N:Nat}
    ((root(s N) * root(s N)) = (root(N))) .)
(auto .)

(lem root-1 :
  ((root(1)) = ((-1, 0))) .)
(auto .)

(disable def-root .)

------------------------------------------------------------------------
--- +
------------------------------------------------------------------------
(lem sim-+ :
```

```
  A{P:Powerlist ; Q:Powerlist ; R:Powerlist}
   ((sim?(P, Q)) = (true)
    =>  (P + Q) : Powerlist
     & ((sim?(P + Q, R)) = (sim?(P, R)))) .)
(cov-split* on P + Q split (sim?(P, R)) .)

(lem cancel-+ :
  A{P:Powerlist}
   ((P + P) = ((2,0) * P)) .)
(cov* on (2, 0) * P .)

------------------------------------------------------------------------
--- *
------------------------------------------------------------------------
(lem sim-* :
  A{P:Powerlist ; Q:Powerlist ; R:Powerlist}
   ((sim?(P, Q)) = (true)
    =>  (P * Q) : Powerlist
     & ((sim?(P * Q, R)) = (sim?(P, R)))) .)
(cov-split* on P * Q split  (sim?(P, R)) .)

(lem *-zip :
  A{P1:Powerlist ; P2:Powerlist ; Q1:Powerlist ; Q2:Powerlist}
   (  (sim?(P1, P2)) = (true)
    & (sim?(Q1, Q2)) = (true)
    & (sim?(P1, Q1)) = (true)
     => ((P1 zip P2) * (Q1 zip Q2)) = ((P1 * Q1) zip (P2 * Q2))) .)
(cov-split* on P1 * Q1 split (P2 * Q2) .)

------------------------------------------------------------------------
--- scalar*
------------------------------------------------------------------------
(lem sim?-scalar* :
  A{P:Powerlist ; E:Complex ; R:Powerlist}
    ((sim?(E * P, R)) = (sim?(P, R))) .)
(cov-split* on E * P split (sim?(P, R)) .)

(lem scalar*-*-left :
  A{C:Complex ; P:Powerlist ; Q:Powerlist}
   ((sim?(P, Q)) = (true)
    => ((C * P) * Q) = (C * (P * Q))) .)
(cov* on (P * Q) .)

(lem scalar*-*-right :
  A{C:Complex ; P:Powerlist ; Q:Powerlist}
   ((sim?(P, Q)) = (true)
    => (P * (C * Q)) = (C * (P * Q))) .)
```

```
(cov* on (P * Q) .)

(lem scalar*-scalar* :
  A{C1:Complex ; C2:Complex ; P:Powerlist}
   ((C1 * (C2 * P)) = ((C1 * C2) * P)) .)
(cov* on C2 * P .)

(lem scalar-1 :
  A{P:Powerlist}
    (((1, 0) * P) = (P)) .)
(cov* on (1, 0) * P .)

(lem scalar*-zip :
  A{C:Complex ; P:Powerlist ; Q:Powerlist}
   ((sim?(P, Q)) = (true)
    => (C * (P zip Q)) = ((C * P) zip (C * Q))) .)
(cov* on (P * Q) .)

(lem scalar*-+-dist :
  A{C:Complex ; P:Powerlist ; Q:Powerlist}
   ((sim?(P, Q)) = (true)
    => (C * (P + Q)) = (C * P + C * Q)) .)
(cov* on P + Q .)

(lem scalar*-+-collect :
  A{C1:Complex ; C2:Complex ; P:Powerlist}
   (((C1 * P) + (C2 * P)) = ((C1 + C2) * P)) .)
(cov* on C1 * P .)

(lem scalar*-+-zero :
  A{P:Powerlist ; Q:Powerlist}
   ((sim?(P, Q)) = (true)
    => (P + ((0,0) * Q)) = (P)) .)
(cov* on P + Q .)

(lem cancel-scalar-+ :
  A{P:Powerlist ; C:Complex}
   ((P + (C * P)) = (((1,0) + C) * P)) .)
(cov* on C * P .)

------------------------------------------------------------------------
--- /
------------------------------------------------------------------------
(lem sim-/ :
  A{P:Powerlist ; Q:NzPolarPowerlist ; R:Powerlist}
   ((sim?(P, Q)) = (true)
    =>  (P / Q) : Powerlist
```

```
     & ((sim?(P / Q, R)) = (sim?(P, R)))) .)
(cov-split* on P / Q split (sim?(P, R)) .)

(lem collect-plist/ :
  A{P:Powerlist ; Q:Powerlist ; R:NzPolarPowerlist}
   ((  (sim?(P, Q)) = (true)
     & (sim?(Q, R)) = (true))
    =>  ((P * (Q / R)) = ((P * Q) / R))) .)
(cov-split* on Q / R split (sim?(P, Q)) .)

(lem scalar-/ :
  A{C:Complex ; P:Powerlist ; Q:NzPolarPowerlist}
   ((sim?(P, Q)) = (true)
    => ((C * P) / Q) = (C * (P / Q))) .)
(cov* on P / Q .)

(lem cancel-*-/ :
  A{P:NzPolarPowerlist ; Q:Powerlist}
   ((sim?(P, Q)) = (true)
    => ((P * Q) / P) = (Q)) .)
(cov* on Q / P .)

------------------------------------------------------------------------
--- lgl
------------------------------------------------------------------------
(lem lgl-+ :
  A{P:Powerlist ; Q:Powerlist}
   ((sim?(P, Q)) = (true)
    => (lgl(P + Q)) = (lgl(P))) .)
(cov* on P + Q .)

(lem lgl-* :
  A{P:Powerlist ; Q:Powerlist}
   ((sim?(P, Q)) = (true)
    => (lgl(P * Q)) = (lgl(P))) .)
(cov* on P * Q .)

(lem lgl-scalar :
  A{C:Complex ; P:Powerlist}
   ((lgl(C * P)) = (lgl(P))) .)
(cov* on C * P .)

------------------------------------------------------------------------
--- powers
------------------------------------------------------------------------
(lem sim?-powers :
  A{N:Nat ; C1:Complex ; C2:Complex}
```

```
    ((sim?(powers(N, C1), powers(N, C2))) = (true)) .)
(cov* on powers(N, C1) .)

(lem sim?-powers-lgl :
  A{P:Powerlist ; Q:Powerlist ; C:Complex}
    ((sim?(powers(lgl(P), C), Q)) = (sim?(P, Q))) .)
(cov-split* on lgl(P) split (sim?(P, Q)).)

(lem powers-* :
  A{N:Nat ; C1:Complex ; C2:Complex}
   ((powers(N, C1) * powers(N, C2)) = (powers(N, C1 * C2))) .)
(cov* on powers(N, C1) .)

(lem lgl-powers :
  A{N:Nat ; C:Complex}
   ((lgl(powers(N, C))) = (N)) .)
(cov* on (powers(N, C)) .)

------------------------------------------------------------------------
--- u and w
------------------------------------------------------------------------
(lem sim?-u :
  A{P:Powerlist ; Q:Powerlist}
    ((sim?(u(lgl(P)), Q)) = (sim?(P, Q))) .)
(auto .)

(lem lgl-u :
  A{N:Nat}
   ((lgl(u(N))) = (N)) .)
(auto .)

(lem sim?-w :
  A{P:Powerlist ; Q:Powerlist}
    ((sim?(w(lgl(P)), Q)) = (sim?(P, Q))) .)
(auto .)

(lem u-squared :
  A{N:Nat}
    ((u(N) * u(N)) = (w(N))) .)
(ind* on N .)

(lem w-tie :
  A{N:Nat}
    ((w(s N)) = (u(N) tie ((-1, 0) * u(N)))) .)
(ind* on N .)

(lem w-0 :
```

```
  (w(0)) = ([ (1, 0) ]) .)
(auto .)

(disable def-u .)
(disable def-w .)

------------------------------------------------------------------------
--- fft
------------------------------------------------------------------------
(lem sim?-fft :
  A{P:Powerlist ; Q:Powerlist}
    ((sim?(fft(P), Q)) = (sim?(P, Q))) .)
(cov-split* using zip on fft(P) split (fft(Q)) .)

------------------------------------------------------------------------
--- ift
------------------------------------------------------------------------
(lem sim?-ift :
  A{P:Powerlist ; Q:Powerlist}
    ((sim?(ift(P), Q)) = (sim?(P, Q))) .)
(cov-split* on ift(P) split (sim?(P, Q)) .)
--- Induction case:
(split on (sim?(V0#0*Powerlist, V0#1*Powerlist)) .)
(auto .)
(split on (sim?(V0#0*Powerlist, V0#3*Powerlist)) .)
(auto .)
(auto .)

(lem lgl-ift :
  A{P:Powerlist}
    ((lgl(ift(P))) = (lgl(P))) .)
(a-inst lem-sim-lgl
  with (P:Powerlist <- (ift(P:Powerlist))) ;
       (Q:Powerlist <- P:Powerlist) .)
(auto .)
```

B.3 fft-ft.itp

```
load powerlist-fft

select ITP-TOOL .
loop init-itp(uiuc) .

(goal fft-ft : FFT
  |- A{P:Powerlist}
    ((fft(P)) = (ft(P))) .)
```

```
load lem-sim-basics.itp
load lem-zip-sim.itp
load lem-zip-ctor.itp
load lem-lgl.itp
load lem-fft-basics.itp

--------------------------------------------------------------------
--- eval
--------------------------------------------------------------------
(lem sim?-eval :
  A{P:Powerlist ; Q:Powerlist ; R:Powerlist}
   ((sim?(eval(P, Q), R)) = (sim?(Q, R))) .)
(cov-split* on eval(P, Q) split (sim?(Q, R)) .)

(lem eval-zip-right :
  A{P:Powerlist ; Q:Powerlist ; R:Powerlist}
   ((sim?(P, Q)) = (true)
    => (eval(R, P zip Q)) = (eval(R, P) zip eval(R, Q))) .)
(cov* on P + Q .)

(lem eval-zip-left :
  A{P:Powerlist ; Q:Powerlist ; R:Powerlist}
    ((sim?(P:Powerlist, Q:Powerlist)) = (true)
     => (eval(P zip Q, R)) = (eval(P, R * R) + R * eval(Q, R * R))) .)
(cov* on R * R .)

--- Main theorem
(cov* using zip on fft(P:Powerlist) .)
--- Induction case:
(a-inst lem-sim-lgl
  with (P:Powerlist <- V0#0*Powerlist) ;
       (Q:Powerlist <- V0#1*Powerlist) .)
(auto .)
```

B.4 fft-ift.itp

```
load powerlist-fft

select ITP-TOOL .
loop init-itp(uiuc) .

(goal fft-ift : FFT
  |- A{P:Powerlist}
    ((fft(ift(P))) = (P)) .)
```

```
load lem-sim-basics.itp
load lem-zip-sim.itp
load lem-zip-ctor.itp
load lem-lgl.itp
load lem-fft-basics.itp

(cov on ift(P) .)
(a-inst lem-sim-lgl
  with (P:Powerlist <- V0#0*Powerlist) ;
       (Q:Powerlist <- V0#1*Powerlist) .)
(auto .)
```

B.5 ift-fft.itp

```
load powerlist-fft

select ITP-TOOL .
loop init-itp(uiuc) .

(goal ift-fft : FFT
  |- A{P:Powerlist}
    ((ift(fft(P))) = (P)) .)

load lem-sim-basics.itp
load lem-zip-sim.itp
load lem-zip-ctor.itp
load lem-lgl.itp
load lem-fft-basics.itp

(cov* using zip on fft(P) .)
```

Appendix C

Batcher sort scripts

C.1 bit.maude

```
fmod BIT is
  sort Bit .
  ops 0 1 : -> Bit [ctor].

  op _<=_ : Bit Bit -> Bool .
  eq B:Bit <= B:Bit = true .
  eq 0 <= 1 = true .
  eq 1 <= 0 = false .
endfm

view Bit from TRIV to BIT is
  sort Elt to Bit .
endv

view Bit<= from TOTAL-ORDER to BIT is
  sort Elt to Bit .
endv
```

C.2 powerlist-sort.maude

```
load powerlist

fmod BATCHER-SORT{X :: TOTAL-ORDER} is
  protecting POWERLIST{TOTAL-PREORDER}{TOTAL-ORDER}{X} * (
    sort          Elt{TOTAL-PREORDER}{TOTAL-ORDER}{X} to          Elt{X},
    sort       Scalar{TOTAL-PREORDER}{TOTAL-ORDER}{X} to       Scalar{X},
    sort  NsPowerlist{TOTAL-PREORDER}{TOTAL-ORDER}{X} to  NsPowerlist{X},
    sort    Powerlist{TOTAL-PREORDER}{TOTAL-ORDER}{X} to    Powerlist{X},
    sort NsPowerlist*{TOTAL-PREORDER}{TOTAL-ORDER}{X} to NsPowerlist*{X},
    sort   Powerlist*{TOTAL-PREORDER}{TOTAL-ORDER}{X} to  Powerlist*{X}).

  var E E' : X$Elt .
  var P P1 P2 Q Q1 Q2 : Powerlist{X} .
```

```
  --- Take minimum of each element in arguments.
  op _min_ : Powerlist{X} Powerlist{X} ~> Powerlist{X} .
  eq [ E ] min [ E' ] = [ if E <= E' then E else E' fi ] .
  eq (P1 zip P2) min (Q1 zip Q2) = (P1 min Q1) zip (P2 min Q2) .

  --- Take maximum of each element in arguments.
  op _max_ : Powerlist{X} Powerlist{X} ~> Powerlist{X} .
  eq [ E ] max [ E' ] = [ if E <= E' then E' else E fi ] .
  eq (P1 zip P2) max (Q1 zip Q2) = (P1 max Q1) zip (P2 max Q2) .

  --- compare([p1 .. pn ], [q1 .. qn]) returns the powerlist:
  ---   [ min(p1, q1) max(p1,q1) .. min(pn, qn) max(pn,qn) ]
  op compare : Powerlist{X} Powerlist{X} ~> Powerlist{X} .
  eq compare(P, Q) = (P min Q) zip (P max Q) .

  --- Batcher merge
  op _bm_ : Powerlist{X} Powerlist{X} ~> Powerlist{X} .
  eq [ E ] bm [ E' ] = compare([ E ], [ E' ]) .
  eq (P1 zip P2) bm (Q1 zip Q2) = compare(P1 bm Q2, P2 bm Q1) .

  --- Sort using batcher merge.
  op bs : Powerlist{X} -> Powerlist{X} .
  eq bs([ E ]) = [ E ] .
  eq bs(P zip Q) = bs(P) bm bs(Q) .
endfm

---(Theorems about bs bi and bm [sorted-bs.itp]
  1. sim?(P, Q) & sorted(P) & sorted(Q) => sorted(P bm Q).
  2. sorted(bs(P))
  3. elts(P) = elts(bs(P))

Reference:
Section 5.3.4, Theorem Z (Zero-One Principle) in The Art of Computer
Programming Volume 3 discusses why one can show a particular type of
sorting routine sorts arbitrary types iff it sorts Booleans correctly.
)

load bit

fmod POWERLIST-BIT-SORT is
  protecting BATCHER-SORT{Bit<=}
   * (sort          Elt{Bit<=} to   Elt,
      sort       Scalar{Bit<=} to   Scalar,
      sort  NsPowerlist{Bit<=} to NsPowerlist,
      sort    Powerlist{Bit<=} to   Powerlist,
      sort NsPowerlist*{Bit<=} to NsPowerlist*,
      sort   Powerlist*{Bit<=} to   Powerlist*) .
```

```
  var B : Bit .
  var P Q : Powerlist .

  --- Returns numbers of zeros in powerlist.
  op zeros : Powerlist -> Nat .
  eq zeros(P zip Q) = zeros(P) + zeros(Q) .
  eq zeros([ 0 ]) = 1 .
  eq zeros([ 1 ]) = 0 .

  op sorted? : Powerlist -> Bool .
  eq sorted?([ B ]) = true .
  eq sorted?(P zip Q)
   = sorted?(P) and
     sorted?(Q) and
     (zeros(Q) <=   zeros(P)) and
     (zeros(P) <= s zeros(Q)) .
endfm
```

C.3 sorted-bs.itp

```
load powerlist-sort
select ITP-TOOL .
loop init-itp(uiuc) .

(goal sorted?-bs : POWERLIST-BIT-SORT
  |- A{P:Powerlist}
      ((sorted?(bs(P))) = (true)) .)

load lem-sim-basics.itp
load lem-zip-sim.itp
load lem-zip-ctor.itp

(lem sim?-cong :
  A{P*:Powerlist* ; Q*:Powerlist* ; R*:Powerlist*}
   ((sim?(P*, Q*)) = (true)
    => (sim?(P*, R*)) = (sim?(Q*, R*))) .)
(cov-split* on sim?(P*, R*) split (sim?(P*, Q*)) .)
--- 18.0
(a-inst hyp-2
    with (Q*:Powerlist*  <- V0#1*Powerlist*) .)
(auto .)
--- 22.0
(a-inst hyp-5
    with (Q*:Powerlist*  <- V0#1*Powerlist*) .)
(auto .)
```

```
(disable def-zip-1 .)
(disable def-zip-2 .)

(lem sim?-min :
  A{P:Powerlist ; Q:Powerlist ; R:Powerlist}
   ((sim?(P, Q)) = (true)
    => (P min Q) : Powerlist
     & (sim?(P min Q, R)) = (sim?(P , R))) .)
(cov-split* using zip on P min Q
     split (bs(R)) ; (zeros(P)) ; (zeros(Q)) .)

(lem sim?-max :
  A{P:Powerlist ; Q:Powerlist ; R:Powerlist}
   ((sim?(P, Q)) = (true)
    => (P max Q) : Powerlist
     & (sim?(P max Q, R)) = (sim?(P , R))) .)
(cov-split* using zip on P max Q
     split (bs(R)) ; (zeros(P)) ; (zeros(Q)) .)

(lem sim?-bm :
  A{P:Powerlist ; Q:Powerlist ; R:Powerlist}
   ((sim?(P, Q)) = (true)
    => (P bm Q) : Powerlist
     & (sim?(P bm Q, R)) = (sim?(P zip Q, R))) .)
(cov-split* using zip on P bm Q
     split (bs(R)) ; (zeros(P)) ; (zeros(Q)) .)
--- 7.2.0
(a-inst lem-sim?-cong
   with (P*:Powerlist* <- ([ 0 ])) ;
        (Q*:Powerlist* <- ([ 1 ])) .)
(auto .)
--- 7.2.0
(a-inst lem-tie-r-sim*
    with (P:Powerlist*  <- (V0#0*Powerlist)) ;
         (Q1:Powerlist* <- (V0#5*Powerlist)) ;
         (Q2:Powerlist* <- (V0#3*Powerlist)) ;
         (R:Powerlist*  <- (V0#1*Powerlist)) .)
(auto .)

(lem sim?-bs :
  A{P:Powerlist ; R:Powerlist}
   ((sim?(bs(P), R)) = (sim?(P, R))) .)
(cov-split* using zip on bs(P) split (bs(R)) .)

(lem zeros-max-sorted? :
  A{P:Powerlist ; Q:Powerlist}
```

```
   (  (sim?(P, Q)) = (true)
    & (sorted?(P)) = (true)
    & (sorted?(Q)) = (true)
    => (zeros(P max Q)) = (min(zeros(P), zeros(Q)))) .)
(cov-split* using zip on P max Q
     split (zeros(P)) ; (zeros(Q)) .)
--- 9.0
(split on zeros(V0#0*Powerlist) <= zeros(V0#1*Powerlist) .)
--- 9.1.0
(split on zeros(V0#2*Powerlist) <= zeros(V0#3*Powerlist) .)
(auto .)
(a-inst lem-<=-false->true-succ
   with (M:Nat <- (zeros(V0#3*Powerlist))) ;
        (N:Nat <- (zeros(V0#2*Powerlist))) .)
(a-inst lem-<=-trans
   with (M:Nat <- (  zeros(V0#1*Powerlist))) ;
        (N:Nat <- (s zeros(V0#3*Powerlist))) ;
        (P:Nat <- (  zeros(V0#2*Powerlist))) .)
(a-inst lem-<=-trans
   with (M:Nat <- (  zeros(V0#1*Powerlist))) ;
        (N:Nat <- (  zeros(V0#2*Powerlist))) ;
        (P:Nat <- (  zeros(V0#0*Powerlist))) .)
(a-inst lem-<=-two-+
   with (M:Nat <- (  zeros(V0#0*Powerlist))) ;
        (N:Nat <- (  zeros(V0#1*Powerlist))) .)
(auto .)
--- 9.2.0
(a-inst lem-<=-false->true-succ
   with (M:Nat <- (zeros(V0#1*Powerlist))) ;
        (N:Nat <- (zeros(V0#0*Powerlist))) .)
(a-inst lem-<=-trans
   with (M:Nat <- (s zeros(V0#1*Powerlist))) ;
        (N:Nat <- (  zeros(V0#0*Powerlist))) ;
        (P:Nat <- (s zeros(V0#2*Powerlist))) .)
(a-inst lem-<=-trans
   with (M:Nat <- (  zeros(V0#3*Powerlist))) ;
        (N:Nat <- (  zeros(V0#1*Powerlist))) ;
        (P:Nat <- (  zeros(V0#2*Powerlist))) .)
(a-inst lem-<=-two-+
   with (M:Nat <- (  zeros(V0#2*Powerlist))) ;
        (N:Nat <- (  zeros(V0#3*Powerlist))) .)
(split on zeros(V0#2*Powerlist) <= zeros(V0#3*Powerlist) .)
(auto .)
(auto .)

(lem zeros-min-sorted? :
  A{P:Powerlist ; Q:Powerlist}
```

```
   (  (sim?(P, Q)) = (true)
    & (sorted?(P)) = (true)
    & (sorted?(Q)) = (true)
    => (zeros(P min Q)) = (max(zeros(P), zeros(Q)))) .)
(cov-split* using zip on P min Q
     split (zeros(P)) ; (zeros(Q)) .)
--- 9.0
(split on zeros(V0#0*Powerlist) <= zeros(V0#1*Powerlist) .)
--- 9.1.0
(split on zeros(V0#2*Powerlist) <= zeros(V0#3*Powerlist) .)
(auto .)
(a-inst lem-<=-false->true-succ
   with (M:Nat <- (zeros(V0#3*Powerlist))) ;
        (N:Nat <- (zeros(V0#2*Powerlist))) .)
(a-inst lem-<=-trans
   with (M:Nat <- (  zeros(V0#1*Powerlist))) ;
        (N:Nat <- (s zeros(V0#3*Powerlist))) ;
        (P:Nat <- (  zeros(V0#2*Powerlist))) .)
(a-inst lem-<=-trans
   with (M:Nat <- (  zeros(V0#1*Powerlist))) ;
        (N:Nat <- (  zeros(V0#2*Powerlist))) ;
        (P:Nat <- (  zeros(V0#0*Powerlist))) .)
(a-inst lem-<=-two-+
   with (M:Nat <- (  zeros(V0#0*Powerlist))) ;
        (N:Nat <- (  zeros(V0#1*Powerlist))) ;
        (P:Nat <- (  zeros(V0#2*Powerlist))) .)
(auto .)
--- 9.2.0
(a-inst lem-<=-false->true-succ
   with (M:Nat <- (zeros(V0#1*Powerlist))) ;
        (N:Nat <- (zeros(V0#0*Powerlist))) .)
(a-inst lem-<=-trans
   with (M:Nat <- (s zeros(V0#1*Powerlist))) ;
        (N:Nat <- (  zeros(V0#0*Powerlist))) ;
        (P:Nat <- (s zeros(V0#2*Powerlist))) .)
(a-inst lem-<=-trans
   with (M:Nat <- (  zeros(V0#3*Powerlist))) ;
        (N:Nat <- (  zeros(V0#1*Powerlist))) ;
        (P:Nat <- (  zeros(V0#2*Powerlist))) .)
(split on zeros(V0#2*Powerlist) <= zeros(V0#3*Powerlist) .)
(a-inst lem-<=-two-+
   with (M:Nat <- (  zeros(V0#3*Powerlist))) ;
        (N:Nat <- (  zeros(V0#2*Powerlist))) .)
(auto .)
(auto .)

(lem sorted?-max :
```

```
  A{P:Powerlist ; Q:Powerlist}
   (   (sim?(P, Q)) = (true)
     & (sorted?(P)) = (true)
     & (sorted?(Q)) = (true)
    => (sorted?(P max Q)) = (true)) .)
(cov-split* using zip on P max Q
     split (zeros(P)) ; (zeros(Q)) .)
--- 9.1.0
(split on zeros(V0#0*Powerlist) <= zeros(V0#1*Powerlist) .)
--- 9.1.1.0
(split on zeros(V0#2*Powerlist) <= zeros(V0#3*Powerlist) .)
(auto .)
(a-inst lem-<=-trans
   with (M:Nat <- (  zeros(V0#0*Powerlist))) ;
        (N:Nat <- (  zeros(V0#1*Powerlist))) ;
        (P:Nat <- (s zeros(V0#3*Powerlist))) .)
(auto .)
--- 9.1.2.0
(split on zeros(V0#2*Powerlist) <= zeros(V0#3*Powerlist) .)
(a-inst lem-<=-false->true
   with (M:Nat <- (zeros(V0#1*Powerlist))) ;
        (N:Nat <- (zeros(V0#0*Powerlist))) .)
(a-inst lem-<=-trans
   with (M:Nat <- (  zeros(V0#1*Powerlist))) ;
        (N:Nat <- (  zeros(V0#0*Powerlist))) ;
        (P:Nat <- (s zeros(V0#2*Powerlist))) .)
(auto .)
(auto .)
--- 9.2.0
(split on zeros(V0#0*Powerlist) <= zeros(V0#1*Powerlist) .)
--- 9.2.1.0
(split on zeros(V0#2*Powerlist) <= zeros(V0#3*Powerlist) .)
(auto .)
(a-inst lem-<=-false->true
   with (M:Nat <- (zeros(V0#3*Powerlist))) ;
        (N:Nat <- (zeros(V0#2*Powerlist))) .)
(a-inst lem-<=-trans
   with (M:Nat <- (zeros(V0#3*Powerlist))) ;
        (N:Nat <- (zeros(V0#2*Powerlist))) ;
        (P:Nat <- (zeros(V0#0*Powerlist))) .)
(auto .)
--- 9.2.2.0
(split on zeros(V0#2*Powerlist) <= zeros(V0#3*Powerlist) .)
(a-inst lem-<=-trans
   with (M:Nat <- (zeros(V0#2*Powerlist))) ;
        (N:Nat <- (zeros(V0#3*Powerlist))) ;
        (P:Nat <- (zeros(V0#1*Powerlist))) .)
```

```
(auto .)
(auto .)

(lem sorted?-min :
  A{P:Powerlist ; Q:Powerlist}
   (   (sim?(P, Q)) = (true)
     & (sorted?(P)) = (true)
     & (sorted?(Q)) = (true)
    => (sorted?(P min Q)) = (true)) .)
(cov-split* using zip on P min Q
     split (zeros(P)) ; (zeros(Q)) .)
--- 9.1.0
(split on zeros(V0#0*Powerlist) <= zeros(V0#1*Powerlist) .)
--- 9.1.1.0
(split on zeros(V0#2*Powerlist) <= zeros(V0#3*Powerlist) .)
(auto .)
(a-inst lem-<=-false->true
   with (M:Nat <- (zeros(V0#3*Powerlist))) ;
        (N:Nat <- (zeros(V0#2*Powerlist))) .)
(a-inst lem-<=-trans
   with (M:Nat <- (  zeros(V0#1*Powerlist))) ;
        (N:Nat <- (s zeros(V0#3*Powerlist))) ;
        (P:Nat <- (s zeros(V0#2*Powerlist))) .)
(auto .)
--- 9.1.2.0
(split on zeros(V0#2*Powerlist) <= zeros(V0#3*Powerlist) .)
(a-inst lem-<=-trans
   with (M:Nat <- (  zeros(V0#0*Powerlist))) ;
        (N:Nat <- (s zeros(V0#2*Powerlist))) ;
        (P:Nat <- (s zeros(V0#3*Powerlist))) .)
(auto .)
(auto .)
--- 9.2.0
(split on zeros(V0#0*Powerlist) <= zeros(V0#1*Powerlist) .)
--- 9.2.1.0
(split on zeros(V0#2*Powerlist) <= zeros(V0#3*Powerlist) .)
(auto .)
(a-inst lem-<=-trans
   with (M:Nat <- (zeros(V0#2*Powerlist))) ;
        (N:Nat <- (zeros(V0#0*Powerlist))) ;
        (P:Nat <- (zeros(V0#1*Powerlist))) .)
(auto .)
--- 9.2.2.0
(split on zeros(V0#2*Powerlist) <= zeros(V0#3*Powerlist) .)
(a-inst lem-<=-false->true
   with (M:Nat <- (zeros(V0#1*Powerlist))) ;
        (N:Nat <- (zeros(V0#0*Powerlist))) .)
```

```
(a-inst lem-<=-trans
   with (M:Nat <- (zeros(V0#3*Powerlist))) ;
        (N:Nat <- (zeros(V0#1*Powerlist))) ;
        (P:Nat <- (zeros(V0#0*Powerlist))) .)
(auto .)
(auto .)

(lem sorted?-bm :
  A{P:Powerlist ; Q:Powerlist}
   (   (sim?(P, Q)) = (true)
     & (sorted?(P)) = (true)
     & (sorted?(Q)) = (true)
    => (sorted?(P bm Q)) = (true)
     & (zeros(P bm Q)) = (zeros(P) + zeros(Q))) .)
(cov-split* using zip on P bm Q
     split (zeros(P)) ; (zeros(Q)) .)
--- First subcase
(split on
     zeros(V0#0*Powerlist bm V0#3*Powerlist)
  <= zeros(V0#2*Powerlist bm V0#1*Powerlist) .)
(auto .)
(a-inst lem-<=-false->true
   with (M:Nat <- (zeros(V0#2*Powerlist bm V0#1*Powerlist))) ;
        (N:Nat <- (zeros(V0#0*Powerlist bm V0#3*Powerlist))) .)
(auto .)
--- Second subcase
(split on
     zeros(V0#0*Powerlist bm V0#3*Powerlist)
  <= zeros(V0#2*Powerlist bm V0#1*Powerlist) .)
(auto .)
(auto .)
--- Third subcase
(split on
     zeros(V0#0*Powerlist bm V0#3*Powerlist)
  <= zeros(V0#2*Powerlist bm V0#1*Powerlist) .)
(auto .)
(auto .)

--- Main goal.
(cov* using zip on bs(P) .)
```

Appendix D

Adder scripts

D.1 powerlist-adder.maude

```
load powerlist
load bit

fth A-OP is
  inc TRIV .
  op _+_ : Elt Elt -> Elt .
  var X Y Z : Elt .
  eq X + (Y + Z) = (X + Y) + Z [nonexec].
endfth

view A-OP from TRIV to A-OP is endv

fmod PREFIX-SUM{X :: A-OP} is
  protecting POWERLIST{A-OP}{X}
          * (sort          Elt{A-OP}{X} to          Elt{X},
             sort       Scalar{A-OP}{X} to       Scalar{X},
             sort  NsPowerlist{A-OP}{X} to  NsPowerlist{X},
             sort    Powerlist{A-OP}{X} to    Powerlist{X},
             sort NsPowerlist*{A-OP}{X} to NsPowerlist*{X},
             sort   Powerlist*{A-OP}{X} to   Powerlist*{X}) .

  var E E' : X$Elt .
  var P Q : Powerlist{X} .

  --- Apply + to all elements of powerlist.
  op _+_ : X$Elt Powerlist{X} -> Powerlist{X} .
  eq E + (P tie Q) = (E + P) tie (E + Q) .
  eq E + [ E' ] = [ E + E' ] .

  --- Prefix sum.
  op prefix : Powerlist{X} ~> Powerlist{X} .
  eq prefix([ E ])  = [ E ] .
  eq prefix(P tie Q) = prefix(P) tie (last(prefix(P)) + prefix(Q)) .
endfm
```

```
fmod CARRY is
  protecting BIT .

  sort Carry .
  subsort Bit < Carry .
  op p : -> Carry [ctor].

  var B : Bit .
  var C : Carry .

  --- Compute carry for two bits.
  op carry : Bit Bit -> Carry .
  eq carry(B, B) = B .
  eq carry(0, 1) = p .
  eq carry(1, 0) = p .

  --- Combines carry bits together.
  op _!_ : Carry Carry -> Carry .
  eq C ! B = B .
  eq C ! p = C .
endfm

view Carry from A-OP to CARRY is
  sort Elt to Carry .
  op _+_ to _!_ .
endv

--- Operations common to both ripple carry and carry lookahead
fmod POWERLIST-NAT-ENCODING is
  protecting PREFIX-SUM{Carry}
   * (sort          Elt{Carry} to         Elt,
      sort       Scalar{Carry} to      Scalar,
      sort  NsPowerlist{Carry} to NsPowerlist,
      sort    Powerlist{Carry} to   Powerlist,
      sort NsPowerlist*{Carry} to NsPowerlist*,
      sort   Powerlist*{Carry} to   Powerlist*,
      op _+_ : Carry Powerlist{Carry} ~> Powerlist{Carry} to _!_) .

  sort Elt{Bit} NsPowerlist{Bit} Powerlist{Bit} .
  subsort Elt{Bit} NsPowerlist{Bit} < Powerlist{Bit} .
  subsort Elt{Bit} < Elt .
  subsort NsPowerlist{Bit} < NsPowerlist .
  subsort Powerlist{Bit} < Powerlist .

  var P Q : Powerlist{Bit} .
```

```
  var N Val : Nat .

  op [_] : Bit -> Elt{Bit} [ctor ditto].
  cmb P tie Q : NsPowerlist{Bit} if sim?(P, Q) .

  --- Returns least-significant bit of number.
  op lsb : Nat -> Bit .
  eq lsb(0) = 0 .
  eq lsb(1) = 1 .
  eq lsb(s s Val) = lsb(Val) .
  --- bv(N, Val) returns representation of Val in powerlist with lgl N.
  op bv : Nat Nat -> Powerlist{Bit} .
  eq bv(0, Val) = [ lsb(Val) ] .
  eq bv(s N,   Val) = bv(N, Val) tie bv(N, Val >> 2 ^ N) .
  --- Expands powerlist into natural number.
  op eval : Powerlist{Bit} -> Nat .
  eq eval([ 0 ]) = 0 .
  eq eval([ 1 ]) = 1 .
  eq eval(P tie Q) = eval(P) + (eval(Q) << (2 ^ lgl(P))) .
endfm

fmod ADDER is
  protecting POWERLIST-NAT-ENCODING .

  sort AdderOutput .
  op <_;_> : Bit Powerlist{Bit} -> AdderOutput [ctor].

  var B B' : Bit .
  var C C' : Carry .
  var P Q P1 P2 Q1 Q2 : Powerlist{Bit} .
  var NsPC : NsPowerlist .
  var PC QC : Powerlist .

  op carry : AdderOutput -> Bit .
  eq carry(< B ; P >) = B .
  op list : AdderOutput -> Powerlist{Bit} .
  eq list(< B ; P >) = P .

  --- Convert adder output to natural number.
  op eval : AdderOutput -> Nat .
  eq eval(< B ; P >) = (eval([ B ]) << 2 ^ lgl(P)) + eval(P) .

  --- Ripple-carry adder.
  op rc : Bit Powerlist{Bit} Powerlist{Bit} ~> AdderOutput .
  eq rc(0, [ 0 ], [ 0 ]) = < 0 ; [ 0 ] > .
  eq rc(0, [ 0 ], [ 1 ]) = < 0 ; [ 1 ] > .
  eq rc(0, [ 1 ], [ 0 ]) = < 0 ; [ 1 ] > .
```

```
  eq rc(0, [ 1 ], [ 1 ]) = < 1 ; [ 0 ] > .
  eq rc(1, [ 0 ], [ 0 ]) = < 0 ; [ 1 ] > .
  eq rc(1, [ 0 ], [ 1 ]) = < 1 ; [ 0 ] > .
  eq rc(1, [ 1 ], [ 0 ]) = < 1 ; [ 0 ] > .
  eq rc(1, [ 1 ], [ 1 ]) = < 1 ; [ 1 ] > .
  eq rc(B, P1 tie P2, Q1 tie Q2)
   = < carry(rc(carry(rc(B, P1, Q1)), P2, Q2)) ;
       list(rc(B, P1, Q1)) tie list(rc(carry(rc(B, P1, Q1)), P2, Q2)) > .

  --- Computes carry values for two powerlists.
  --- carry = \bullet in adam's paper.
  op carry : Powerlist{Bit} Powerlist{Bit} ~> Powerlist .
  eq carry([ B ], [ B' ]) = [ carry(B, B') ] .
  eq carry(P1 tie P2, Q1 tie Q2) = carry(P1, Q1) tie carry(P2, Q2) .

  --- Sums powerlist.
  op sum : Powerlist{Bit} Powerlist ~> Powerlist{Bit} .
  eq sum(P tie Q, PC tie QC) = sum(P, PC) tie sum(Q, QC) .
  eq sum([ B ], [ B' ]) = [ B ] .
  eq sum([ 0 ], [ p  ]) = [ 1 ] .
  eq sum([ 1 ], [ p  ]) = [ 0 ] .

  --- Shifts powerlist to right by one by inserting carry.
  op rsh : Carry Powerlist ~> Powerlist .
  eq rsh(C, [ C' ]) = [ C ] .
  eq rsh(C, PC tie QC) = rsh(C, PC) tie rsh(last(PC), QC) .

  --- Carry lookahead adder.
  op cl : Bit Powerlist{Bit} Powerlist{Bit} ~> AdderOutput .
  eq cl(B, P, Q)
   = < last(prefix(rsh(B, carry(P, Q)))) ! last(carry(P, Q)) ;
       sum( prefix(rsh(B, carry(P, Q))),        carry(P, Q)) > .
endfm

---(Theorems:
 1. [lem-eval-bv.itp]
    eval(bv(N, Val)) = Val rem 2 ^ (2 ^ N).
 2. [rc.itp]
    sim?(P,Q)
    => rc(B,P,Q) = < bv(0, eval(B) + eval(P) + eval(Q) >> 2 ^ lgl(P)) ;
                    bv(lgl(P), eval(B) + eval(P) + eval(Q))>
 3: [rc-cl.itp]
    rc(B,P,Q) = cl(B,P,C)
)
```

D.2 lem-adder.itp

```
(lem lgl-bv-powerlist :
  A{N:Nat ; Val:Nat}
    ((lgl(bv(N, Val))) = (N)) .)
(cov* on bv(N, Val) .)

(lem eval-lsb :
  A{V:Nat}
     ((eval([ lsb(V) ])) = (V rem 2)) .)
(cov* on lsb(V) .)

(lem eval-bv :
  A{N:Nat ; Val:Nat}
     ((eval(bv(N, Val))) = (Val rem 2 ^ (2 ^ N))) .)
(cov* on bv(N, Val) .)

(lem rc-powerlist :
  A{B:Bit ; P:Powerlist‘{Bit‘} ; Q:Powerlist‘{Bit‘}}
   ((sim?(P, Q)) = (true)
     => (rc(B, P, Q)) : AdderOutput
      & (sim?(list(rc(B, P, Q)), P)) = (true)) .)
(cov* on rc(B, P, Q) .)
```

D.3 rc.itp

```
load powerlist-adder
select ITP-TOOL .
loop init-itp(uiuc) .

(goal rc : ADDER
  |- A{B:Bit ; P:Powerlist‘{Bit‘} ; Q:Powerlist‘{Bit‘}}
      ((sim?(P, Q)) = (true)
       => (rc(B, P, Q))
        = (< lsb(eval([ B ]) + eval(P) + eval(Q) >> 2 ^ lgl(P)) ;
             bv(lgl(P), eval([ B ]) + eval(P) + eval(Q))>)) .)

load lem-sim-basics.itp
load lem-zip-sim.itp
load lem-lgl.itp
load lem-adder.itp

-----------------------------------------------------------------------
--- lsb
-----------------------------------------------------------------------
(lem lsb-2N :
```

```
  A{N:Nat}
   ((lsb(N + N)) = (O)) .)
(ind* on N .)

(lem lsb-s-2N :
  A{N:Nat}
   ((lsb(s(N + N))) = (1)) .)
(ind* on N .)

(lem lsb-sd-ss :
  A{M:Nat ; N:Nat}
   ((lsb(sd(M, s s N))) = (lsb(sd(M, N)))) .)
(cov-split* on sd(M, N) split (lsb(M)) ; (lsb(N)) .)

(lem lsb-sd-+2 :
  A{M:Nat ; N:Nat}
   ((lsb(sd(M, N + N))) = (lsb(M))) .)
(ind* on N .)

(lem lsb-M-N+N :
  A{M:Nat ; N:Nat}
   ((lsb(M + N + N)) = (lsb(M))) .)
(cov* on lsb(M) .)

------------------------------------------------------------------------
--- bv
------------------------------------------------------------------------
(lem bv-N-sd-2^2^P :
  A{M:NzNat ; N:Nat ; P:Nat}
   ((M * 2 ^ (2 ^ P) <= N) = (true)
    => (bv(P, sd(N, M * 2 ^ (2 ^ P)))) = (bv(P, N))) .)
(cov* on bv(P, N) .)

(lem bv-N-shl-2^P :
  A{M:Nat ; N:Nat ; P:Nat}
   ((bv(P, M + (N << 2 ^ P))) = (bv(P, M))) .)
(cov* on bv(P, M) .)

------------------------------------------------------------------------
--- eval
------------------------------------------------------------------------
(lem eval-leq-bit :
  A{B:Bit}
   ((2 <= eval([ B ])) = (false)) .)
(cov* on eval([ B ]) .)

(lem eval-leq-p :
```

```
  A{P:Powerlist‘{Bit‘}}
   ((2 ^ (2 ^ lgl(P)) <= eval(P)) = (false)) .)
(cov* on eval(P) .)
--- Induction case 3:
(a-inst lem-sim-lgl
  with (P:Powerlist <- V0#0*Powerlist‘{Bit‘}) ;
       (Q:Powerlist <- V0#1*Powerlist‘{Bit‘}) .)
(auto .)

(lem eval-leq-p-general :
  A{P:Powerlist‘{Bit‘} ; Q:Powerlist‘{Bit‘}}
   ((sim?(P, Q)) = (true)
    => (2 ^ (2 ^ lgl(Q)) <= eval(P)) = (false)) .)
(cns .)
(a-inst lem-sim-lgl
  with (P:Powerlist <- Q*Powerlist‘{Bit‘}) ;
       (Q:Powerlist <- P*Powerlist‘{Bit‘}) .)
(auto .)

--- Main theorem
(cov* on rc(B, P, Q) .)
--- Last goal:
(a-inst lem-sim-lgl
  with (P:Powerlist <- V0#2*Powerlist‘{Bit‘}) ;
       (Q:Powerlist <- V0#1*Powerlist‘{Bit‘}) .)
(a-inst lem-sim-lgl
  with (P:Powerlist <- V0#3*Powerlist‘{Bit‘}) ;
       (Q:Powerlist <- V0#1*Powerlist‘{Bit‘}) .)
(auto .)
```

D.4 rc-cl.itp

```
load powerlist-adder
select ITP-TOOL .
loop init-itp(uiuc) .

(goal rc-cl : ADDER
  |- A{B:Bit ; P:Powerlist‘{Bit‘} ; Q:Powerlist‘{Bit‘}}
      ((sim?(P,Q)) = (true)
       => (rc(B,P,Q)) = (cl(B,P,Q))) .)

load lem-sim-basics.itp
load lem-adder.itp

(lem last-bit :
  A{P:Powerlist‘{Bit‘}}
```

```
   ((last(P)) : Bit) .)
(cov* on last(P) .)

------------------------------------------------------------------------
--- !
------------------------------------------------------------------------
(lem !-left-assoc :
  A{C1:Carry ; C2:Carry ; C3:Carry}
   ((C1 ! (C2 ! C3)) = ((C1 ! C2) ! C3)) .)
(eq-split* on C2 ! C3 .)

(lem B!-bit :
  A{B:Bit ; C:Carry}
   ((B ! C) : Bit) .)
(eq-split* on B ! C .)

------------------------------------------------------------------------
--- carry
------------------------------------------------------------------------
(lem carry-powerlist :
  A{P:Powerlist‘{Bit‘} ; Q:Powerlist‘{Bit‘} ; R:Powerlist}
   ((sim?(P, Q)) = (true)
     => (carry(P, Q)) : Powerlist
      & (sim?(carry(P, Q), R)) = (sim?(P, R))) .)
(cov-split* on carry(P, Q) split (sim?(P, R)) .)

------------------------------------------------------------------------
--- sum
------------------------------------------------------------------------
(lem sum-powerlist :
  A{P:Powerlist‘{Bit‘} ; Q:Powerlist ; R:Powerlist}
   ((sim?(P, Q)) = (true)
     => (sum(P, Q)) : Powerlist‘{Bit‘}
      & (sim?(sum(P, Q), R)) = (sim?(P, R))) .)
(cov-split* on sum(P, Q) split (sim?(P, R)) .)

------------------------------------------------------------------------
--- ! list
------------------------------------------------------------------------
(lem elt-list-!-sim? :
  A{C:Carry ; P:Powerlist ; R:Powerlist}
   (  (C ! P) : Powerlist
    & (sim?(C ! P, R)) = (sim?(P, R))) .)
(cov-split* on C ! P split (sim?(P, R)) .)

(lem B!-powerlist :
  A{B:Bit ; P:Powerlist}
```

```
   ((B ! P) : Powerlist‘{Bit‘}) .)
(cov* on B ! P .)

(lem !-left-assoc-list :
  A{C:Carry ; C’:Carry ; P:Powerlist}
   ((C ! (C’ ! P)) = ((C ! C’) ! P)) .)
(cov-split* on C’ ! P split (C ! C’) .)

(lem last-! :
  A{C:Carry ; P:Powerlist}
   ((last(C ! P)) = (C ! last(P))) .)
(cov* on C ! P .)

-------------------------------------------------------------------
--- prefix and rsh
-------------------------------------------------------------------
(lem prefix-sim? :
  A{P:Powerlist ; R:Powerlist}
   (  (prefix(P)) : Powerlist
    & (sim?(prefix(P), R)) = (sim?(P, R))) .)
(cov-split* on prefix(P) split (sim?(P, R)) .)

(lem rsh-sim? :
  A{C:Carry ; P:Powerlist ; R:Powerlist}
   (  (rsh(C, P)) : Powerlist
    & (sim?(rsh(C, P), R)) = (sim?(P, R))) .)
(cov-split* on rsh(C, P) split (sim?(P, R)) .)

(lem lf-rsh-powerlist :
  A{B:Bit ; P:Powerlist}
   ((prefix(rsh(B, P))) : Powerlist‘{Bit‘}) .)
(cov* on prefix(P) .)

(lem last-prefix-rsh :
  A{C:Carry ; C’:Carry ; P:Powerlist}
   ((C ! last(prefix(rsh(C’, P)))) = (last(prefix(rsh(C ! C’, P))))) .)
(cov* on C ! P .)

(lem prefix-rsh :
  A{C:Carry ; C’:Carry ; P:Powerlist}
   ((C ! prefix(rsh(C’, P))) = (prefix(rsh(C ! C’, P)))) .)
(cov* on C ! P .)

--- Main lemma
(cov* on rc(B, P, Q) .)
```

References

[1] William Adams. Verifying adder circuits using powerlists. Technical report, University of Texas at Austin, Department of Computer Science, Austin, TX, USA, 1994.

[2] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986.

[3] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[4] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In Kousha Etessami and Sriram K. Rajamani, editors, *Proc. of CAV'05*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285. Springer, 2005.

[5] Jean-Michel Autebert, Jean Berstel, and Luc Boasson. Context-free languages and pushdown automata. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 1, Word Language Grammar, chapter 3, pages 111–174. Springer, 1997.

[6] Franz Baader and Tobias Nipkow. *Term rewriting and All That*. Cambridge University Press, 1998.

[7] Franz Baader and Wayne Snyder. Unification theory. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier, 2001.

[8] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking rewriting on Java. In Franz Baader, editor, *Proc. of RTA'07*, volume 4533 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 2007.

[9] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proc. CAV'07*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer, 2007.

[10] Kenneth E. Batcher. Sorting networks and their applications. In *Spring Joint Computer Conference, AFIPS Proc.*, volume 32, pages 307–314, 1968.

[11] Narjes Berregeb, Adel Bouhoula, and Michaël Rusinowitch. SPIKE-AC: A system for proofs by induction in associative-commutative theories. In Harald Ganzinger, editor, *Proc. of RTA-96*, volume 1103 of *Lecture Notes in Computer Science*, pages 428–431. Springer, 1996.

[12] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004. URL `http://www.labri.fr/publications/l3a/2004/BC04`.

[13] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. The Barcelogic SMT solver. In Aarti Gupta and Sharad Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 294–298. Springer, 2008.

[14] Yohan Boichut, Pierre-Cyrille Héam, and Olga Kouchnarenko. Automatic verification of security protocols using approximations. technical report RR-5727, INRIA, October 2005.

[15] Peter Borovanský, Claude Kirchner, Hélène Kirchner, and Pierre-Etienne Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285(2):155–185, 2002.

[16] Alexandre Boudet. Unification in order-sorted algebras with overloading. In Deepak Kapur, editor, *Proc. of CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1992.

[17] Gérard Boudol. Computational semantics of term rewriting systems. *Algebraic methods in semantics*, pages 169–236, 1986.

[18] Adel Bouhoula and Florent Jacquemard. Automatic verification of sufficient completeness for conditional constrained term rewriting systems. Technical Report LSC-05-17, ENS de Cachan, 2006. Available at: `http://www.lsv.ens-cachan.fr/Publis/`.

[19] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[20] Robert Stephen Boyer and J Strother Moore. *A Computational Logic*. Academic Press, 1979.

[21] Mark van den Brand, Arie van Deursen, Jan Heering, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The ASF+SDF meta-environment: a component-based language development environment. *Electronic Notes Theoretical Computer Science*, 44(2), 2001.

[22] Bishop Brock, Matt Kaufmann, and J Strother Moore. Rewriting with equivalence relations in ACL2. *Journal of Automated Reasoning*, 40 (4):293–306, 2008. ISSN 0168–7433. doi: http://dx.doi.org/10.1007/s10817-007-9095-9.

[23] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT solver. In Aarti Gupta and Sharad Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 299–303. Springer, 2008.

[24] Anne-Cécile Caron, Hubert Comon, Jean-Luc Coquidé, Max Dauchet, and Florent Jacquemard. Pumping, cleaning and symbolic constraints solving. In Serge Abiteboul and Eli Shamir, editors, *Proc. of ICALP*, volume 820 of *Lecture Notes in Computer Science*, pages 436–449. Springer, 1994.

[25] Manuel Clavel, Francisco Durán, Steven Eker, José Meseguer, and Mark-Oliver Stehr. Maude as a formal meta-tool. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *World Congress on Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1684–1703. Springer, 1999.

[26] Manuel Clavel, José Meseguer, and Miguel Palomino. Reflection in membership equational logic, many-sorted equational logic, horn logic with equality, and rewriting logic. *Electronic Notes Theoretical Computer Science*, 71, 2002.

[27] Manuel Clavel, Miguel Palomino, and Adrián Riesco. Introducing the ITP tool: a tutorial. *Journal of Universal Computer Science*, 12(11): 1618–1650, 2006.

[28] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[29] Manuel Clavel, Francisco Durán, Joe Hendrix, Salvador Lucas, José Meseguer, and Peter Csaba Ölveczky. The Maude formal tool environment. In Till Mossakowski, Ugo Montanari, and Magne Haveraaen, editors, *Proc. of CALCO*, volume 4624 of *Lecture Notes in Computer Science*, pages 173–178. Springer, 2007.

[30] Hubert Comon. Completion of rewrite systems with membership constraints. part I: Deduction rules. *Journal of Symbolic Computation*, 25(4): 397–419, 1998.

[31] Hubert Comon. Completion of rewrite systems with membership constraints. part II: Constraint solving. *Journal of Symbolic Computation*, 25 (4):421–453, 1998.

[32] Hubert Comon and Florent Jacquemard. Ground reducibility is EXPTIME-complete. *Information and Computation*, 187(1):123–153, 2003.

[33] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. Available at: `http://www.grappa.univ-lille3.fr/tata`, 1997.

[34] Hubert Comon-Lundh, Florent Jacquemard, and Nicolas Perrin. Tree automata with memory, visibility and structural constraints. In Helmut Seidl, editor, *Proc. of FoSSaCS 2007*, volume 4423 of *Lecture Notes in Computer Science*, pages 168–182. Springer, 2007.

[35] Evelyne Contejean and Claude Marché. CiME: Completion modulo E. In Harald Ganzinger, editor, *Proc. of RTA-96*, volume 1103 of *Lecture Notes in Computer Science*, pages 416–419. Springer, 1996.

[36] Evelyne Contejean, Claude Marché, Benjamin Monate, and Xavier Urbain. The CiME 2 system, 2000. Available at `http://cime.lri.fr/`.

[37] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.*, 19:297–301, 1965.

[38] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Sematics, pages 243–320. Elsevier and MIT Press, 1990.

[39] Nachum Dershowitz and David A. Plaisted. Rewriting. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 9, pages 535–610. Elsevier, 2001.

[40] Nachum Dershowitz, Stéphane Kaplan, and David A. Plaisted. Rewrite, rewrite, rewrite, rewrite, ... *Theoretical Computer Science*, 83 (1):71–96, 1991.

[41] Philippe Devienne, Jean-Marc Talbot, and Sophie Tison. Solving classes of set constraints with tree automata. In Gert Smolka, editor, *Proc. of CP97*, volume 1330 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 1997.

[42] Francisco Durán and José Meseguer. The Maude specification of Full Maude. Technical report, SRI International, 1999. Available at: `http://maude.cs.uiuc.edu/papers/`.

[43] Francisco Durán, Salvador Lucas, José Meseguer, Claude Marché, and Xavier Urbain. Proving termination of membership equational programs. In Nevin Heintze and Peter Sestoft, editors, *Proc. of PEPM*, pages 147–158. ACM, 2004.

[44] Francisco Durán, Salvador Lucas, Claude Marché, José Meseguer, , and Xavier Urbain. Proving operational termination of membership equational programs. *Higher-Order and Symbolic Computation*, 21(1–2):59–88, 2008.

[45] Bruno Dutertre and Leonardo Mendonça de Moura. System description: Yices 1.0. Submitted to SMT-COMP 2006, 2006. Available at `http://yices.csl.sri.com/`.

[46] Santiago Escobar. Refining weakly outermost-needed rewriting and narrowing. In *Proc. of PPDP*, pages 113–123. ACM, 2003.

[47] Santiago Escobar. *Strategies and Analysis Techniques for Functional Program Optimization.* PhD thesis, Departamento de Sistemas Informaticos y Computación, Universidad Politécnica de Valencia, Valencia, Spain, Oct 2003.

[48] Santiago Escobar, Catherine Meadows, and José Meseguer. A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties. *Theoretical Computer Science*, 367(1–2):162–202, 2006.

[49] Santiago Escobar, Joe Hendrix, Catherine Meadows, and José Meseguer. Diffie-Hellman cryptographic reasoning in the Maude-NRL protocol analyzer. In *Proc. of SecRet 2007*, 2007.

[50] Guillaume Feuillade, Thomas Genet, and Valérie Viet Triem Tong. Reachability analysis over term rewriting systems. *J. Autom. Reasoning*, 33 (3–4):341–383, 2004.

[51] Kokichi Futatsugi and Răzvan Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998.

[52] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian Reid, editor, *Proc. of POPL 1985*, pages 52–66. ACM Press, 1985.

[53] John P. Gallagher and Germán Puebla. Abstract interpretation over nondeterministic finite tree automata for set-based analysis of logic programs. In Shriram Krishnamurthi and C. R. Ramakrishnan, editors, *Proc. of PADL 2002*, volume 2257 of *Lecture Notes in Computer Science*, pages 243–261. Springer, 2002.

[54] Jean Gallier and Tomas Isakowitz. Order-sorted rigid E-unification. Technical Report STERN IS-91-40, Stern School of Business at New York University, 1991. Available at `http://hdl.handle.net/2451/14397`.

[55] Ruben Gamboa. The correctness of the fast fourier transform: A structured proof in ACL2. *Formal Methods in System Design*, 20(1):91–106, 2002.

[56] Ruben Gamboa. A formalization of powerlist algebra in ACL2. *Journal of Automated Reasoning*, 2008. Submitted.

[57] Ruben Gamboa. Defthms about zip and tie: Reasoning about powerlists in ACL2. Technical Report TR87-02, University of Texas Computer Science, 1997.

[58] Ruben Gamboa and John R. Cowles. Theory extension in ACL2(r). *Journal of Automated Reasoning*, 38(4):273–301, 2007.

[59] Thomas Genet and Francis Klay. Rewriting for cryptographic protocol verification. In David A. McAllester, editor, *Proc. of CADE-17*, volume 1831 of *Lecture Notes in Computer Science*, pages 271–290. Springer, 2000.

[60] Jürgen Giesl and Aart Middeldorp. Transformation techniques for context-sensitive rewrite systems. *Journal of Functional Programming*, 14(4):379–427, 2004.

[61] Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In Ulrich Furbach and Natarajan Shankar, editors, *Proc. of IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 281–286. Springer, 2006.

[62] Seymore Ginsburg and Edwin H. Spanier. Semiground, Presburger formulas and languages. *Pacific Journal of Mathematics*, 16:285–296, 1966.

[63] Joseph Goguen and Răzvan Diaconescu. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4(3):363–392, 1994.

[64] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

[65] Michael J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993. URL `http://www.dcs.glasgow.ac.uk/~tfm/HOLbook.html`.

[66] John V. Guttag. *The Specification and Application to Programming of Abstract Data Types*. PhD thesis, University of Toronto, 1975. Computer Science Department, Report CSRG-59.

[67] John V. Guttag and James J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[68] John V. Guttag, James J. Horning, Stephen J. Garland, and K.D. Jones. *Larch: Languages and Tools for Formal Specification*. Springer, 1993.

[69] Joe Hendrix. CETA: A library for equational tree automata, 2008. Software available under GPL license at `http://texas.cs.uiuc.edu/ceta/`.

[70] Joe Hendrix and José Meseguer. On the completeness of context-sensitive order-sorted specifications. In Franz Baader, editor, *Proc. of RTA'07*, volume 4533 of *Lecture Notes in Computer Science*, pages 229–245. Springer, 2007.

[71] Joe Hendrix and José Meseguer. Order-sorted equational unification revisited. In Günter Kniesel and Jorge Sousa Pinto, editors, *Preproceedings of RULE'08*, To appear in Electronic Notes in Theoretical Computer Science. Elsevier, 2008.

[72] Joe Hendrix and Hitoshi Ohsaki. Combining equational tree automata over AC and ACI theories. In Andrei Voronkov, editor, *Proc. of RTA'08*, volume 5117 of *Lecture Notes in Computer Science*, pages 142–156. Springer, 2008.

[73] Joe Hendrix, Manuel Clavel, and José Meseguer. A sufficient completeness reasoning tool for partial specifications. In Jürgen Giesl, editor, *Proc. of RTA'05*, volume 3467 of *Lecture Notes in Computer Science*, pages 165–174. Springer, 2005.

[74] Joe Hendrix, Hitoshi Ohsaki, and José Meseguer. Sufficient completeness checking with propositional tree automata. Technical Report UIUCDCS-R-2005-2635, University of Illinois, 2005. Available at: `http://maude.cs.uiuc.edu/tools/scc/`.

[75] Joe Hendrix, José Meseguer, and Hitoshi Ohsaki. A sufficient completeness checker for linear order-sorted specifications modulo axioms. In Ulrich Furbach and Natarajan Shankar, editors, *Proc. of IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 151–155. Springer, 2006.

[76] Joe Hendrix, Hitoshi Ohsaki, and Mahesh Viswanathan. Propositional tree automata. In Frank Pfenning, editor, *Proc. of RTA'06*, volume 4098 of *Lecture Notes in Computer Science*, pages 165–174. Springer, 2006.

[77] John Edward Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[78] Mark W. Hopkins and Dexter Kozen. Parikh's theorem in commutative Kleene algebra. In *Proc. of LICS 1999*, pages 394–401. IEEE Computer Society, 1999.

[79] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In Martin Odersky and Philip Wadler, editors, *Proc. of ICFP*, pages 11–22, New York, NY, USA, 2000. ACM Press.

[80] Florent Jacquemard, Michaël Rusinowitch, and Laurent Vigneron. Tree automata with equality constraints modulo equational theories. In Ulrich Furbach and Natarajan Shankar, editors, *Proc. of IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 557–571. Springer, 2006.

[81] Jean-Pierre Jouannaud and Hélène Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal on Computing*, 15(4):1155–1194, 1986.

[82] Jean-Pierre Jouannaud and Emmanuel Kounalis. Automatic proofs by induction in theories without constructors. *Information and Computation*, 82(1):1–33, 1989.

[83] Deepak Kapur. An automated tool for analyzing completeness of equational specifications. In *ISSTA*, pages 28–43, 1994.

[84] Deepak Kapur. Constructors can be partial too. In Robert Veroff, editor, *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*, pages 177–210. MIT Press, 1997.

[85] Deepak Kapur and Mahadevan Subramaniam. Automated reasoning about parallel algorithms using powerlists. In Vangalur S. Alagar and Maurice Nivat, editors, *Proc. of AMAST'95*, volume 936 of *Lecture Notes in Computer Science*, pages 416–430. Springer, 1995.

[86] Deepak Kapur and Mahadevan Subramaniam. New uses of linear arithmetic in automated theorem proving by induction. *Journal of Automated Reasoning*, 16(1–2):39–78, 1996.

[87] Deepak Kapur and Mahadevan Subramaniam. Mechanical verification of adder circuits using Rewrite Rule Laboratory. *Formal Methods in System Design*, 13(2):127–158, 1998.

[88] Deepak Kapur and Hantao Zhang. An overview of Rewrite Rule Laboratory (RRL). *Journal of Computer and Mathematics with Applications*, 29 (2):91–114, 1995.

[89] Deepak Kapur, Paliath Narendran, and Hantao Zhang. On sufficient-completeness and related properties of term rewriting systems. *Acta Informatica*, 24(4):395–415, 1987.

[90] Deepak Kapur, Paliath Narendran, Daniel Rosenkrantz, and Hantao Zhang. Sufficient-completeness, ground-reducibility and their complexity. *Acta Informatica*, 28(4):311–350, 1991.

[91] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, 2000.

[92] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

[93] Claude Kirchner. Order-sorted equational unification. Presented at the fifth International Conference on Logic Programming (Seattle, USA), August 1988. Also as rapport de recherche INRIA 954, Dec. 88.

[94] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, University of Aarhus, January 2001. Notes Series NS-01-1. Available from `http://www.brics.dk/mona/`. Revision of BRICS NS-98-3.

[95] Donald Ervin Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley Professional, 1997.

[96] Jacob Kornerup. *Data Structures for Parallel Recursion*. PhD thesis, The University of Texas at Austin, December 1997.

[97] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[98] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proc. of POPL 2006*, pages 42–54. ACM, 2006. ISBN 1-59593-027-2.

[99] Hanbing Liu and J Strother Moore. Executable JVM model for analytical reasoning: A study. *Sci. Comput. Program.*, 57(3):253–274, 2005.

[100] Salvador Lucas. Transfinite rewriting semantics for term rewriting systems. In Aart Middeldorp, editor, *Proc. of RTA'01*, volume 2051 of *Lecture Notes in Computer Science*, pages 216–230. Springer, 2001.

[101] Salvador Lucas. Context-sensitive rewriting strategies. *Information and Computation*, 178(1):294–343, 2002.

[102] Salvador Lucas. Proving termination of context-sensitive rewriting by transformation. *Information and Computation*, 204(12):1782–1846, 2006.

[103] Salvador Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998 (1), 1998.

[104] Salvador Lucas. MU-TERM: A tool for proving termination of context-sensitive rewriting. In Vincent van Oostrom, editor, *Proc. of RTA'04*, volume 3091 of *Lecture Notes in Computer Science*, pages 200–209. Springer, 2004.

[105] Denis Lugiez. Multitree automata that count. *Theoretical Computer Science*, 333(1–2):225–263, 2005.

[106] Denis Lugiez and J. L. Moysset. Tree automata help one to solve equational formulae in AC-theories. *Journal of Symbolic Computation*, 18(4): 297–318, 1994.

[107] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[108] Theodore McCombs. Maude 2.0 primer. Available at: `http://maude.cs.uiuc.edu/primer/`, August 2003.

[109] José Meseguer. Conditioned rewriting logic as a united model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[110] José Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Proc. of Marktoberdorf 1997*, volume 165 of *NATO ASI Series F: Computer and Systems Sciences*, pages 347–398. NATO Advanced Study Institute, Springer-Verlag, 1998.

[111] José Meseguer. Membership algebra as a logical framework for equational specification. In Francesco Parisi-Presicce, editor, *Proc. of WADT'97*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1997.

[112] José Meseguer, Joseph A. Goguen, and Gert Smolka. Order-sorted unification. *Journal of Symbolic Computation*, 8:383–413, 1989.

[113] Jayadev Misra. Powerlist: a structure for parallel recursion. *ACM Transactions on Programming Languages and Systems*, 16(6):1737–1767, 1994.

[114] Leonardo Mendonça de Moura and Nikolaj Bjørner. Engineering DPLL(T) + saturation. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Proc. of IJCAR*, volume 5195 of *Lecture Notes in Computer Science*. Springer, 2008. To appear.

[115] Leonardo Mendonça de Moura and Nikolaj Bjørner. Efficient E-matching for SMT solvers. In Frank Pfenning, editor, *Proc. of CADE-21*, volume 4603 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2007.

[116] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[117] Makoto Murata, Akihiko Tozawa, Michiharu Kudo, and Satoshi Hada. XML access control using static analysis. *ACM Trans. Inf. Syst. Secur.*, 9(3):292–324, 2006.

[118] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[119] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier, 2001.

[120] Tobias Nipkow and Gerhard Weikum. A decidability result about sufficient-completeness of axiomatically specified abstract data types. In Armin B. Cremers and Hans-Peter Kriegel, editors, *Proc. of Theoretical Computer Science*, volume 145 of *Lecture Notes in Computer Science*, pages 257–268. Springer, 1982.

[121] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[122] Enno Ohlebusch. *Advanced topics in term rewriting*. Springer-Verlag, London, UK, 2002.

[123] Hitoshi Ohsaki. Beyond regularity: Equational tree automata for associative and commutative theories. In Laurent Fribourg, editor, *Proc. of CSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 539–553. Springer, 2001.

[124] Hitoshi Ohsaki and Hiroyuki Seki. Languages modulo normalization. In Jaime G. Carbonell and J org Siekmann, editors, *Proc. of FroCoS 2007*, volume 4720 of *Lecture Notes in Computer Science*, pages 221–236. Springer, 2007.

[125] Hitoshi Ohsaki and Toshinori Takai. Decidability and closure properties of equational tree languages. In Sophie Tison, editor, *Proc. of RTA'02*, volume 2378 of *Lecture Notes in Computer Science*, pages 114–128. Springer, 2002.

[126] Hitoshi Ohsaki and Toshinori Takai. ACTAS : A system design for associative and commutative tree automata theory. *Electronic Notes Theoretical Computer Science*, 124(1):97–111, 2005.

[127] Hitoshi Ohsaki, Jean-Marc Talbot, Sophie Tison, and Yves Roos. Monotone AC-tree automata. In Geoff Sutcliffe and Andrei Voronkov, editors, *Proc. of LPAR 2005*, volume 3835 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2005.

[128] Sam Owre, John Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Proc. of CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

[129] Rohit J. Parikh. On context-free languages. *Journal of the ACM*, 13(4): 570–581, 1966. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321356.321364.

[130] Gordon Plotkin. Building-in equational theories. *Machine Intelligence*, 7: 73–90, 1972.

[131] David M. Rusinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7$^{\text{TM}}$ processor. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998.

[132] Ralf Sasse and José Meseguer. Java+ITP: A verification tool based on hoare logic and algebraic semantics. *Electronic Notes Theoretical Computer Science*, 176(4):29–46, 2007.

[133] Helmut Seidl, Thomas Schwentick, and Anca Muscholl. Numerical document queries. In *Proc. of PODS*, pages 155–166. ACM Press, 2003.

[134] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, 1984.

[135] Giora Slutzki. Alternating tree automata. *Theoretical Computer Science*, 41:305–318, 1985.

[136] Kumar Neeraj Verma. Two-way equational tree automata for AC-like theories: Decidability and closure properties. In Robert Nieuwenhuis, editor, *Proc. of RTA'03*, volume 2706 of *Lecture Notes in Computer Science*, pages 180–196. Springer, 2003.

[137] Kumar Neeraj Verma. On closure under complementation of equational tree automata for theories extending AC. In Moshe Y. Vardi and Andrei Voronkov, editors, *Proc. of LPAR 2003*, volume 2850 of *Lecture Notes in Computer Science*, pages 183–197. Springer, 2003.

[138] Kumar Neeraj Verma and Jean Goubault-Larrecq. Alternating two-way AC-tree automata. *Information and Computation*, 205(6):817–869, 2007.

[139] Kumar Neeraj Verma, Helmut Seidl, and Thomas Schwentick. On the complexity of equational horn clauses. In Robert Nieuwenhuis, editor, *Proc. of CADE-20*, volume 3632 of *Lecture Notes in Computer Science*, pages 337–352. Springer, 2005.

[140] Patrick Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285(2):487–517, 2002.

[141] Christoph Walther. Many-sorted unification. *Journal of the ACM*, 35 (1):1–17, 1988. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/42267.45071.

[142] Christoph Weidenbach. Unification in sort theories and its applications. *Annals of Mathematics and Artificial Intelligence*, 18:261–293, 1996.

[143] Freek Wiedijk, editor. *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Computer Science*, 2006. Springer.

[144] Isao Yagi, Yoshiaki Takata, and Hiroyuki Seki. A static analysis using tree automata for XML access control. In Doron Peled and Yih-Kuen Tsay, editors, *Proc. of ATVA 2005*, volume 3707 of *Lecture Notes in Computer Science*, pages 234–247. Springer, 2005.

[145] Hans Zantema. Termination of context-sensitive rewriting. In Hubert Comon, editor, *Proc. of RTA-97*, volume 1232 of *Lecture Notes in Computer Science*, pages 172–186. Springer, 1997.

[146] Hantao Zhang, Deepak Kapur, and Mukkai S. Krishnamoorthy. A mechanizable induction principle for equational specifications. In Ewing Lusk and Ross Overbeek, editors, *Proc. of CADE-9*, volume 310 of *Lecture Notes in Computer Science*, pages 162–181. Springer, 1988.

Author's Biography

Joe Hendrix was born in Houston, Texas in 1978. He graduated from the University of Texas with bachelor degrees in Plan II Honors and Computer Science in December 2000. After working for various companies, he enrolled in the Ph.D. program at the University of Illinois in 2002. During his six years at Illinois, he has had internships as NASA Ames and Microsoft Research, and a visiting position in Amagasaki, Japan.