Peter Csaba Ölveczky

Formal Modeling and Analysis of Distributed Systems

An introduction based on executable modeling in Maude

January 26, 2015

Springer
Preface

The two main goals of this book are to:

1. provide an introduction to formal modeling and analysis of both data types and, in particular, distributed systems; and
2. provide an introduction to distributed computer systems and the challenges of designing and analyzing such systems.

The book is meant to be a first introduction to formal methods and therefore does not assume any previous knowledge about formal methods or distributed systems; it is based on a third-year course at the University of Oslo, but can equally well be taught at the second-year level. Some exposition to programming could be useful; likewise, previous experience with simple recursive functions is useful but not strictly necessary. There are no prerequisites on the mathematical side.

A distinguishing feature of this book is the significant use of the rewriting-logic-based Maude language and simulation and model checking tool for formally modeling both data types and distributed systems. Data types are specified using term rewriting, that is, they are specified using a functional programming style that traditionally appeals to students; indeed, a valuable side effect of studying this book is training in writing (first-order) recursive programs. For formally modeling distributed systems, Maude provides a simple yet intuitive and expressive modeling formalism that is particularly suitable for modeling distributed systems in an object-oriented way. The Maude system is by now a mature and well established tool that has been developed at leading research groups at the University of Illinois at Urbana-Champaign and at SRI International and is now increasingly used around the world.

About the Content

As mentioned above, one main goal of this book is to gently introduce students to a wide range of concepts in formal methods, including:
• verifying properties about programs and (models of) systems; e.g., proving that a specification/program terminates for all possible inputs, and using equational logic to prove semantics properties;
• logics and inference systems; and
• automated model checking techniques to analyze properties for some—but not all—possible inputs/systems.

This book is divided into two parts. The first part deals with specifying the data types needed to model complex distributed systems. This part introduces classical algebraic specification and term rewriting theory, including reasoning about termination and confluence and inductive equational theorems.

The second part deals with formally modeling and analyzing distributed systems in rewriting logic using Maude. This part introduces rewriting logic, and the object-oriented modeling of distributed systems; to express powerful properties, it also introduces temporal logic. The analysis of such models will happen through Maude simulations, reachability analysis, and LTL model checking, thereby also giving the students a hands-on experience of the state space explosion problem for distributed systems. As mentioned above, a second main goal of this work is introduce the students to the problems of designing and analyzing distributed systems. Instead of giving theoretical explanations of these issues, the book tries to convey intuition about distributed systems and their design challenges through a range of examples/case studies in different domains, including: the classic dining philosophers problem, communication protocols like the alternating bit protocol and the sliding window protocol, distributed algorithms such as the distributed two-phase protocol for distributed or replicated database systems, distributed mutual exclusion and leader election algorithms, and the NSPK cryptography protocol.

The book is based on a course that has been given at the University of Oslo for more than 10 years, which implies that the book contains a wealth of exercises, both smaller ones and larger ones suitable for course projects, etc.

Oslo, January 2015

Peter Csaba Ölveczky
## Contents

1 Introduction ................................................... 1
  1.1 Modeling .................................................. 2
    1.1.1 Models of Distributed Systems ....................... 3
    1.1.2 From Model to System ................................. 4
  1.2 The Maude Modeling Language and Analysis Tool .......... 4
  1.3 Why Maude? ................................................ 5
  1.4 Content of the Book ....................................... 7
    1.4.1 Part I: Algebraic Specification and Term Rewriting ... 7
    1.4.2 Part II: Dynamic Systems ............................. 7

Part I Equational Specifications and Their Analysis

2 Equational Specifications in Maude .......................... 11
  2.1 Hello World: Our First Maude Specifications .......... 12
    2.1.1 Natural Numbers with Addition ....................... 13
    2.1.2 The Boolean Values and Functions .................... 15
    2.1.3 Natural Numbers and Booleans: Less Than ........... 15
  2.2 Many-Sorted Equational Specifications .................. 16
  2.3 Requirements of Equational Specifications .............. 20
    2.3.1 One-to-one Constructor Basis ......................... 20
    2.3.2 Termination (No Infinite Looping) ................. 20
    2.3.3 Uniqueness of the “Result” .......................... 21
    2.3.4 Definedness: The Result Should be a Constructor Term .. 22
    2.3.5 Maude and the Requirements ......................... 22
  2.4 Many-Sorted Specification of Data Types ................ 23
    2.4.1 Defining Functions: Getting Started ............... 23
    2.4.2 Expressiveness of Many-Sorted Equational Specifications .. 24
    2.4.3 Maude Specifications of Some Data Types .......... 25
  2.5 Order-Sorted Equational Specifications ................ 30
    2.5.1 Examples of Order-Sorted Equational Specifications ... 32
  2.6 Membership Equational Logic Specifications ............. 35
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.7</td>
<td>Built-in Data Types</td>
<td>37</td>
</tr>
<tr>
<td>2.7.1</td>
<td>The Booleans</td>
<td>37</td>
</tr>
<tr>
<td>2.7.2</td>
<td>The Natural Numbers</td>
<td>38</td>
</tr>
<tr>
<td>2.7.3</td>
<td>The Integers</td>
<td>40</td>
</tr>
<tr>
<td>2.7.4</td>
<td>The Rational Numbers</td>
<td>40</td>
</tr>
<tr>
<td>2.7.5</td>
<td>The Floating-Point Numbers</td>
<td>41</td>
</tr>
<tr>
<td>2.7.6</td>
<td>Strings</td>
<td>41</td>
</tr>
<tr>
<td>2.8</td>
<td>Associativity and Commutativity: Lists and Multisets</td>
<td>43</td>
</tr>
<tr>
<td>2.8.1</td>
<td>Commutativity, Associativity, and Identity</td>
<td>43</td>
</tr>
<tr>
<td>2.8.2</td>
<td>Associativity and Identity: Lists</td>
<td>45</td>
</tr>
<tr>
<td>2.8.3</td>
<td>Associativity, Commutativity, and Identity: Multisets and Sets</td>
<td>46</td>
</tr>
<tr>
<td>2.9</td>
<td>Examples: Quicksort and Mergesort</td>
<td>48</td>
</tr>
<tr>
<td>2.9.1</td>
<td>Quicksort</td>
<td>49</td>
</tr>
<tr>
<td>2.9.2</td>
<td>Mergesort</td>
<td>50</td>
</tr>
<tr>
<td>2.10</td>
<td>* Some Other Maude Features</td>
<td>51</td>
</tr>
<tr>
<td>2.10.1</td>
<td>Parameterized Modules</td>
<td>51</td>
</tr>
<tr>
<td>2.10.2</td>
<td>Telling Maude how to Evaluate an Expression</td>
<td>53</td>
</tr>
<tr>
<td>2.10.3</td>
<td>Other Features</td>
<td>54</td>
</tr>
<tr>
<td>3</td>
<td>Operational Semantics of Equational Specifications</td>
<td>55</td>
</tr>
<tr>
<td>3.1</td>
<td>The Reduction Relation</td>
<td>55</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Basic Definitions</td>
<td>56</td>
</tr>
<tr>
<td>3.1.2</td>
<td>The Reduction Relation</td>
<td>58</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Some Derived Relations</td>
<td>58</td>
</tr>
<tr>
<td>3.2</td>
<td>Operational Properties</td>
<td>59</td>
</tr>
<tr>
<td>3.3</td>
<td>Conditional Equations and Matching with assoc/comm</td>
<td>60</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Conditional Equations</td>
<td>60</td>
</tr>
<tr>
<td>3.3.2</td>
<td>* A-, C-, and AC-matching is NP-hard</td>
<td>61</td>
</tr>
<tr>
<td>4</td>
<td>Termination</td>
<td>63</td>
</tr>
<tr>
<td>4.1</td>
<td>Undecidability of Termination</td>
<td>64</td>
</tr>
<tr>
<td>4.2</td>
<td>Nontermination</td>
<td>67</td>
</tr>
<tr>
<td>4.3</td>
<td>Proving Termination Using “Weight Functions”</td>
<td>68</td>
</tr>
<tr>
<td>4.4</td>
<td>Simplification Orders</td>
<td>71</td>
</tr>
<tr>
<td>4.4.1</td>
<td>The Lexicographic Path Order</td>
<td>74</td>
</tr>
<tr>
<td>4.4.2</td>
<td>The Multiset Path Order and Other Variations of Ipo</td>
<td>75</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Comparing Weight Functions and Simplification Orders</td>
<td>76</td>
</tr>
<tr>
<td>5</td>
<td>Confluence</td>
<td>79</td>
</tr>
<tr>
<td>5.1</td>
<td>Unification</td>
<td>81</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Checking Local Confluence</td>
<td>83</td>
</tr>
</tbody>
</table>
Contents

6 Equational Logic ........................................... 87
   6.1 Equational Logic ........................................ 88
      6.1.1 * Knuth-Bendix Completion ......................... 94
   6.2 Inductive Theorems ..................................... 96
      6.2.1 Proving Inductive Theorems for Nat .............. 97
      6.2.2 Inductive Theorems for Other Data Types ....... 100

7 Algebraic Semantics .................................. 105

8 Reasoning About Imperative Programs ............... 107

Part II Specification and Analysis of Distributed Systems in Maude

9 Modeling Dynamic and Distributed Systems in Rewriting Logic .... 111
   9.1 Dynamic Systems ....................................... 111
      9.1.1 Properties of Dynamic and Distributed Systems .... 112
      9.1.2 Behaviors of Distributed Systems .................. 113
   9.2 Modeling Dynamic Systems in Rewriting Logic ........ 114
      9.2.1 Rewrite Rules ..................................... 114
      9.2.2 Rewriting Logic Specifications .................... 115
      9.2.3 Nondeterminism and Nontermination ............... 116
      9.2.4 Examples ........................................... 116
   9.3 Concurrency ............................................ 119
      9.3.1 Concurrent steps .................................. 119
      9.3.2 Nested Concurrency ................................. 122
   9.4 Deduction in Rewriting Logic ......................... 124
      9.4.1 Concurrent Steps ................................... 125
      9.4.2 Termination and Confluence ....................... 127
   9.5 * Frozen Operators .................................... 128
   9.6 * Denotational Semantics .............................. 129

10 Executing Rewriting Logic Specifications in Maude .......... 131
   10.1 Executing One Sequential Rewrite Step ............... 131
   10.2 Simulating Single Behaviors .......................... 134
   10.3 Search .................................................. 136

11 Concurrent Objects in Maude ............................. 143
   11.1 Modeling Concurrent Objects in Maude ............... 143
      11.1.1 Rewrite Rules .................................... 144
   11.2 Concurrent Objects in Full Maude ..................... 151
      11.2.1 Using Full Maude .................................. 151
      11.2.2 Object-Oriented Modules in Full Maude ......... 152
      11.2.3 Subclasses ........................................ 154
      11.2.4 Search in Full Maude ............................... 157
      11.2.5 Using Full Maude: Repetition ..................... 158
   11.3 Example: The Dining Philosophers .................... 159
11.3.1 Problem Description .............................................. 159
11.3.2 Modeling the Dining Philosophers ......................... 160
11.3.3 Deadlock and Livelock .................................... 161
11.3.4 Fairness Issues ............................................. 162
11.3.5 Version 2: A Deadlock-Free Solution ..................... 162
11.3.6 Version 3: A Deadlock-Free and Livelock-Free Solution 162

12 Modeling Communication in Maude ...................... 167
12.1 Synchronous Communication ................................ 168
12.2 Asynchronous Communication by Unordered Message Passing . 169
  12.2.1 Unordered Unicast ...................................... 169
  12.2.2 Multicast ............................................... 173
  12.2.3 Broadcast ............................................... 174
  12.2.4 Wireless Broadcast .................................. 174
  12.2.5 Modeling Unreliable Communication .................... 175
12.3 Ordered Asynchronous Communication using Links ........ 177
  12.3.1 Links and Concurrency ................................ 179
  12.3.2 Unreliable Links ...................................... 180
  12.3.3 Links with Limited Capacity ......................... 181
12.4 Asynchronous Communication Using Shared Variables .... 182

13 Distributed Algorithms ........................................ 185

14 Examples: Transport Protocols .......................... 187
14.1 Reliable Communication Through Sequence Numbers ........ 187
  14.1.1 Maude Modeling ...................................... 189
  14.1.2 Formal Analysis ...................................... 191
14.2 The Alternating Bit Protocol .............................. 193
14.3 The Sliding Window Protocol ............................. 195
  14.3.1 Sliding Window with Links ......................... 200
14.4 Atomicity of Distributed Transactions: The Two-Phase Commit Protocol ........................................ 202
  14.4.1 The Two-Phase Commit Protocol ...................... 203
  14.4.2 Abstraction ........................................... 204
  14.4.3 Assumptions .......................................... 205
  14.4.4 Specification and Analysis of 2PC in Maude ......... 205
14.5 Distributed Mutual Exclusion ......................... 214
14.6 Distributed Leader Election ............................... 219
  14.6.1 A Ring-based Leader Election Algorithm ............ 219
  14.6.2 A Spanning-Tree-based Algorithm for Wireless Networks 220
14.7 Consensus Algorithms ...................................... 225
Chapter 1
Introduction

Our modern society is becoming increasingly dependent on large and complex computer systems. Our cars, airplanes, banks, power plants, social interactions, shopping activities, etc., are all controlled to a large extent by computer systems. Most computer systems these days are distributed systems consisting of multiple computers, or processors, of various kinds that collaborate to perform some desired actions. This applies not only to social networks or, say, buying plane tickets—which involves multiple airlines, customers, financial institutions, etc.—but also airplanes and cars that contain large numbers of processors. There is also a trend towards integrating medical devices into networks, and so on.

Unfortunately, distributed systems are quite complex and significantly harder to get right than single-threaded sequential programs, because:

- any component in the system may perform an action at any time,
- it may be hard to know whether, or when, a sent message will be delivered, and
- it may be hard to predict the behavior of other components in the system.

Example 1.1. A prerequisite for banking is of course that (i) you know that you are communicating with your bank and not with some impostor, and (ii) the bank knows that the person pretending to be you actually is you. These two essential properties are called mutual authentication. In a physical bank, you know that you are in your bank by the imposing building, and the bank clerk asks you to show some photo identification to be sure that you are who you claim to be. In online banking and commerce, mutual authentication protocols (“programs for distributed systems”) are used to ensure mutual authentication. One of the most well-established mutual authentication protocols is the so-called Needham-Schroeder public key protocol (NSPK) that was developed in 1978 by some of the best experts in the field. It is typically written as follows:

\[
\begin{align*}
\text{Message 1.} & \quad A \rightarrow B : \quad A.B.\{N_a.A\}_{PK(B)} \\
\text{Message 2.} & \quad B \rightarrow A : \quad B.A.\{N_a.N_b\}_{PK(A)} \\
\text{Message 3.} & \quad A \rightarrow B : \quad A.B.\{N_b\}_{PK(B)}
\end{align*}
\]
Chapter ?? explains what all this means; essentially, A and B are the agents that want to establish mutual authentication (e.g., the bank and you), and the system consists of sending three messages: first one message \((A.B\{N_a.A\}_{PK(B)})\) is sent from A to B; then B responds by sending a message \((B.A\{N_a.N_b\}_{PK(A)})\) back to A; finally, A sends a message \((A.B\{N_b\}_{PK(B)})\) to B. After these three messages have been sent and received, A should know that it communicates with B, and vice versa.

This protocol was heavily studied, used, and assumed correct until 1995, when Gavin Lowe used techniques very similar to those in this book to break the protocol.

This example shows that even a three-line distributed “program” can be really hard to get right. However, our lives and economy depend crucially on the correctness of complex distributed systems. How can we develop correct distributed systems and ensure that they indeed are correct?

1.1 Modeling

Let us consider an analogy. Thousands of years ago, building a hut for yourself was pretty easy and could be done right away without much elaboration. If the hut collapsed, you could rebuild it in half an hour or so. Just like you could start coding the basic programs in your introductory programming course without further ado. However, buildings have become much more complex the last 1000 years. How are complex buildings constructed these days? One typically does not start building a large building with only a faint idea of what you want. You first build (or draw) a model of your building. A first model may be quite rough, but can be quickly developed and allows the architect and the person commissioning the building to get an idea of whether this is what they want. Once the main design is agreed upon, a more detailed model should be used to deduce properties of the model: will the bridge collapse? can the proposed skyscraper withstand the strong winds of the area, floods, and earthquakes? The point is of course that:

1. such models are developed reasonably cheaply and quickly before starting to build the building; and
2. one should be able to use the model and the laws of physics to predict whether certain desired properties will hold in the building to be built.

It may be hard to compute by hand whether your skyscraper will withstand the winds/earthquakes/floods in the region. It is therefore desirable to let computers analyze whether those properties will hold.

When advanced models have been developed and analyzed, impressive modern-day engineering technology can “easily” construct the building from the models. It may not be a coincidence that we know Gustave Eiffel, Oscar Niemeyer, and Frank Gehry, but have absolutely no clue about who actually built the Eiffel Tower, the Museum of Contemporary Art in Niteroi, and the Guggenheim Museum in Bilbao.
1.1 Modeling

1.1.1 Models of Distributed Systems

In the same way, one needs models of distributed systems before implementing them: you do not want to implement your new avionics system directly on an Airbus A380 and have one plane crash for each mistake in your code, or to deploy some new e-commerce algorithm before you are really confident that your design is correct. The model should be reasonably quick to develop and should focus on the “essence” of the design (whatever that may be) and abstract from other details. For example, a model of a distributed algorithm could focus on what happens when a message is successfully received or is lost in transmission, but should probably abstract from details on how a packet is sent from one computer to another.

Of course, a model you can only look at is not very useful. What we really want is to be able to deduce properties from the model: can the flight control system deadlock? can your authentication protocol be broken by malicious agents? does the e-commerce protocol also function well if a crucial server goes down? Just like the architect should be able to use the laws of physics to predict properties about the building yet to be built, so should a system designer be able to infer desirable properties of his model of the distributed system.

To obtain consequences of the design, the model must have a clear and precise meaning, and there must be some laws/rules that allows us to deduce consequences of the model. Therefore, the model should be a mathematical object with precise, mathematical, rules of how one can deduce properties of the design. Such a mathematical model of a computer system is called a formal model.

In general, models, or specifications, of systems can roughly be classified to be either system (or operational) specifications or requirement (or property) specifications. System models specify the system, which means the computations performed by the system, whereas a requirement specification specifies the requirements that a system should satisfy. For example, in an authentication protocol like NSPK, the three lines in Example 1.1 defines the system model: it defines the computation steps that the participants should perform (namely, sending messages, either to start a session or as a response to receiving a message). The corresponding requirement specification states the overall requirements that the system should satisfy: “when an agent $A$ thinks that it has established an authenticated connection with an agent $B$, then it indeed has a connection with $B$ and not some other agent”, or “an agent $A$ that wants to establish an authenticated connection with an agent $B$ using the NSPK protocol will eventually be able to do so.”

The main goal is to prove that all possible behaviors of the system (model) satisfy its requirement specification. Therefore, both the system model and the requirements model must be mathematical objects, and there must be mathematical rules that allow us to deduce whether or not a system satisfies its requirements. Furthermore, just like the architect may want to use computer simulations and analysis to analyze consequences of her design, it would indeed be very advantageous if we could let the computer do the analysis for us.

The formal system model should preferably be executable; that is, the model can directly be executed. This would allow for a range of automated computer analyses,
for example by simulating single behaviors of the system being modeled, or by model checking analyses that analyze many, or all, possible behaviors of the system.

This book focuses on developing and analyzing—by computer and by hand—executable formal models of distributed computer systems. It also deals with formalizing requirements of distributed systems using temporal logic. One can focus on many different aspects of a system, including the efficiency of the system. This book focuses on modeling and analyzing the functionality of the system.

1.1.2 From Model to System

The ultimate goal is not to have a nice model for its own sake, but to develop a correct system implementations. However, just like modern engineering technology and companies are very good at constructing even very large and complex buildings from correct models, modern programmers and programming environments and methodologies are very good at implementing systems from correct specifications. In addition, there are commercial code generation tools that can automatically generate code from higher-level models.

Therefore, developing correct models is a very crucial task in the system development process. When the task at hand is well understood, the actual implementation in, say, a real aircraft is “just” programming and hardware engineering. In an early example illustrating the importance of developing correct system models, it turned out that of the 197 critical defects identified during integration and system testing of the Voyager and Galileo spacecrafts, only three were due to coding errors [? , ?]. Most faults arose in requirements and difficult design problems related to distribution [?]. Furthermore, not only are defects most likely to be introduced in the early stage of software development; it is also much cheaper to correct errors early in the software development process.

1.2 The Maude Modeling Language and Analysis Tool

This book uses the Maude [?] modeling language to define the executable formal models of the distributed system, and uses the Maude analysis tool to formally analyze the models. In Maude, a distributed system is formalized as a rewriting logic [? , ?] theory. Maude and rewriting logic were both developed by José Meseguer and his research group at the Computer Science Laboratory at SRI International. (Meseguer now works at the University of Illinois at Urbana-Champaign.)

In rewriting logic, the data types of the system are defined algebraically by equations. In essence, defining data types amounts to define functions in a recursive, functional programming style. The dynamic behavior of a system is then defined by rewrite rules which describe how a part of the state can change in one step.
Maude supports object-oriented programming, including multiple inheritance, and asynchronous communication through message passing, in a natural way.

The Maude interpreter evaluates an expression in an *equational* Maude program by applying the equations “from left to right” until no equation can be applied, thereby computing the *normal form* (or “value”) of the expression. The interpreter executes *rewrite* programs by “arbitrarily” applying rewrite rules (also “from left to right”) on the given initial expression/state, either until no rule can be applied or until a user-given upper bound on the number of rewrites has been reached. (The equations are applied to reduce each intermediate state to its normal form before a rewrite rule is applied.)

Since rewriting logic theories model distributed systems, they are typically *non-deterministic*, meaning that there may be many completely different behaviors from the same initial state of a system. A first form of analysis provided by Maude of such system is to *simulate one* of those behaviors by *rewriting*, which “arbitrarily” applies rewrite rules (also “from left to right”) on the given initial state, either until no rule can be applied or until a user-given upper bound on the number of rewrites has been reached. (The equations are applied to reduce each intermediate state to its normal form before a rewrite rule is applied.) To analyze all possible behaviors *from a given initial state* one can use Maude’s high-performance *search* capabilities to investigate whether certain (un)desired states can be reached from the initial state.

As mentioned above, we typically have *two* levels of specification: a *system specification* describing which actions the system can perform, and a *requirement specification* describing the requirements that the system should satisfy. Not only can we specify the system in Maude; we can also define the requirements the system should satisfy in Maude as linear temporal logic formulas. Maude’s high-performance *model checker* [?], which is comparable in efficiency with specialized state-of-the-art model checkers, can then be used to decide whether all possible behaviors from a given initial state satisfy the requirements, *provided* that the set of states reachable from the initial state is a finite set.

The Maude system, including a user manual, the source code, etc., is available free of charge at http://maude.cs.illinois.edu for various UNIX/Linux platforms, as well as for Mac (and Windows under Cygwin, see http://maude.cs.uiuc.edu/download/windows.html).

### 1.3 Why Maude?

There are a number of reasons why I think that Maude is a good choice for an introduction to formal modeling and analysis of distributed systems:

*Simple and intuitive formalism.* Maude models basically consist of equations that define functions recursively, and rewrite rules that specify how the states evolve dynamically. That’s all! There are no tricky constructs for concurrency or communication. This functional programming style tends to appeal to students.
Expressive formalism. The modeling formalism is fairly expressive, which makes it easy and convenient to define models of complex systems. This is in contrast to simpler, e.g., automaton-based, approaches which require a significant amount of work to specify larger systems, if at all possible. Furthermore, Maude provides a natural and simple model of concurrent objects, which is ideal for modeling distributed systems. Together, this implies that we can easily and naturally model a wide range of distributed systems, as illustrated in this book.

Active area of research. A number of leading research groups perform research on rewriting logic and apply Maude to state-of-the-art systems. A recent survey [?] lists about 1000 published scientific papers involving rewriting logic and Maude. Some applications of Maude includes:

- Researchers at Microsoft and the University of Illinois at Urbana-Champaign (UIUC) modeled aspects of web browsers and their interface in Maude, and used Maude search to discover many previously unknown address bar and status bar spoofing attacks that can potentially be used in phishing attacks in web browsers [?]. Maude is now also used to formally specify and analyze a new secure web browser being developed at UIUC [?].
- Modeling and analysis of a number of complex security and network communication protocols, including new 50-page multicast protocols, new protocols developed by the IETF, etc. (see, e.g., [?], [?], [?], [?], [?], [?]).
- Most modeling and programming languages do not have a well-defined precise meaning (or semantics); the meaning of a model may be unclear or ambiguous, and the meaning of a program may depend on the compiler being used. This is of course unacceptable for safety-critical systems. Furthermore, the lack of a formal meaning makes it impossible to deduce properties about such models, and hence to build tools for such analysis. Due to its expressiveness and simplicity, Maude is suitable to define the mathematical semantics/meaning of a model or a program, and has been used to define the semantics to a wide range of modeling and programming languages, including subsets of the avionics (aircraft software) industrial modeling standard AADL [?], the PLEXIL language developed at NASA for spacecraft operations [?], the most complete formal definition of the C language [?], subsets of the graphical Ptolemy II modeling language used in industry [?], and so on. Such a formalization not only defines a precise mathematical meaning to a modeling/programming language, but also means that models/programs in such languages can be analyzed using the Maude tool. Apart from analyzing PLEXIL; Ptolemy II, AADL models, there is also an efficient tool for analyzing multi-threaded Java programs [?].
- Finding several bugs in embedded software used by major car makers.
- Programs developed at NASA to determine the position of objects in space.
- Modeling of cell biology to simulate and analyze biological reactions [?], [?].

The recent overview paper [?] gives an overview of some applications of Maude.

Mature and efficient. Maude is a fairly mature, robust, and very efficient tool, publicly released in 1998, and is still under active development. It is also open-source and easy to install.
1.4 Content of the Book

A model of a distributed system can be seen to consist of (at least) two parts: (i) the definition of the data types (integers, Booleans, lists, sets, and so on) needed to define the states; and (ii) the definition of the dynamic behavior of the system. This is reflected in the structure of this book, which is divided into two parts.

Part I deals with defining data types by equational specifications, and to analyze both the meaning and the operational properties of such equational specifications. Part II deals with defining the dynamics of a distributed system using rewriting logic, and of manually and automatically analyzing such models. Since another objective of this book is to introduce distributed systems, Part II introduces examples of such systems from different domains, including communication protocols, distributed algorithms, and cryptographic (or “security”) protocols.

1.4.1 Part I: Algebraic Specification and Term Rewriting

This part covers basic classic topics in algebraic specification and term rewriting.

In particular, Chapter 2 introduces equational specification in Maude; we define and execute in Maude the usual data types: natural numbers, integers, lists, binary trees, and multisets. We define the usual functions on these data types, including the quicksort and mergesort functions on lists.

Chapter 3 introduces some operational properties that equational specifications should satisfy. To exemplify how to formally reason about specifications and programs, I have chosen to focus on reasoning about termination. Chapter 4 provides some intuition and more concrete techniques to prove that your specification does not contain an infinite loop for any input. We study the theoretical basis for the concept of simplification orders, and use the standard path orders to prove termination. Chapter 5 shows how to verify that specifications are confluent, that is, that the result of computing an expression is independent of the order in which Maude chooses to apply the equations.

Chapter 6 shows how to use equational logic to reason about the “meaning” of a specification. In particular, we focus on how induction techniques can be used to prove that certain desired properties “follow logically” from a specification.

1.4.2 Part II: Dynamic Systems

Chapter 9 introduces rewriting logic and explains how rewrite rules can be used to specify the possible concurrent behaviors of a system.

Chapter 10 explains how rewriting logic models can be analyzed in Maude by simulating one possible behavior of the system and by searching for (un)desired states. Chapter 11 then introduces Maude’s model of concurrent objects; all the
larger examples will be modeled in an object-oriented style. Chapters 9 to 11 illustrate the concepts on simple examples, such as various small “games” and modeling the “lives” of persons, and end with the well known dining philosophers problem.

Chapter 12 shows how different forms of communication (unicast/multicast/broadcast; unordered/ordered message delivery; message loss; etc.) can be modeled at a high level of abstraction in Maude. These techniques are used to model a TCP-like transport protocol that uses sequence numbers to achieve reliable and ordered message communication over a network infrastructure that is unreliable and that only provides unordered message delivery. We then modify this protocol to the classic alternating bit protocol when we can assume ordered but unreliable links in the network. These two protocols are then generalized to two versions of the sliding window protocol, which is supposedly the most frequently used protocol in networking [1].

We are then ready for some larger examples. Chapter 13 deals with modeling and analyzing a number of classic distributed algorithms, including the two-phase commit protocol for distributed and/or database systems, distributed mutual exclusion algorithms, and distributed leader election and consensus algorithms.

Chapter ?? shows how Maude can be used to model and analyze the aforementioned well-known security protocol (the Needham-Schroeder public-key authentication protocol [2]), whose goal is to let Alice and Bob establish a communication between them so that Alice can be sure she’s communicating with Bob and not with the malicious intruder Walker. Is the well-known security protocol up to this task, or can Maude show that Walker can impersonate Bob?

Chapter ?? introduces invariance and other kinds of requirements that our systems may have satisfy, and discusses both how Maude can be used to analyze such system properties, and how they may be analyzed “by hand.”

These kinds of requirements are then formalized in Chapter ??, which introduces temporal logic to formally define requirements that our models should satisfy, and by explains how Maude’s model checker can be used to check whether our systems models satisfy their requirements.

Finally, Chapter ?? by briefly mentions the following two kinds of extensions of dynamic systems can be modeled and analyzed in (extensions of) Maude:

1. **Real-time systems**, where the amount of time of/between events play a crucial role and must be taken into account in the model.
2. **Probabilistic systems**, where certain events may happen with higher probabilities than others.
Part I

Equational Specifications and Their Analysis
Chapter 2
Equational Specification in Maude

This chapter describes how data types can be defined in Maude as equational specifications. In particular, Section 2.1 introduces specification and execution in Maude with some simple “Hello World!” examples specifying the natural numbers and the Boolean values. Section 2.2 defines many-sorted equational specifications and explains how Maude computes with equations. Section 2.3 describes four important requirements that an equational specification must satisfy for Maude computations to make sense. Section 2.4 shows the Maude specification of other data types, including lists, multisets, and binary trees, and discusses the expressiveness of many-sorted equational specifications. Data types are often related; for example, the natural numbers are a subset of the integers. Such a subset relationship is captured in equational specifications by subsorts, which are treated in Section 2.5, and by sort memberships (Section 2.6). For convenience and performance, efficient versions of a number of data types (natural numbers, Booleans, integers, rationals, and floating-point numbers, as well as strings) are built-in in Maude as explained in Section 2.7. It is sometimes hard to elegantly specify data types so that the specification satisfies the criteria in Section 2.3. Section 2.8 therefore introduces functional attributes that can be used to define lists and multisets in nice ways in Maude. Section 2.9 exemplifies Maude specifications on lists with two well known sorting algorithms: quicksort and mergesort. Finally, Section 2.10 briefly discusses other Maude features, including parametrized modules.

Maude specifications are declarative programs, which specify what to compute, whereas imperative programs, such as Java programs, give a step-by-step description of how to compute something. Declarative languages have some attractive features, including the following:

- Declarative languages do not have pointers/aliasing/side effects which make imperative programs very hard to understand and reason about.
- Declarative programs are easier to specify and modify. The constructs are more “powerful,” making it easier to specify complicated tasks, and to modify programs, as there are no side effects.
• Specification is programming: Instead of having to worry about all the intricate details of, say, quicksort or insertion sort, declarative programming allows you to specify what quicksort means, and you have the quicksort program for free.

• You can reason mathematically about a declarative Maude program, since such a program specifies a mathematical object as explained in Chapter ???. This implies that a specification has a clear mathematical meaning (semantics). In imperative programs, the meaning of a program is usually given at a low level by how the program changes the values of the memory cells in the CPU. It is much harder to reason at that level, and it often difficult to know what a program really does. Since a Maude specification is a mathematical object, it can be reasoned about mathematically quite easily by following mathematical rules. For example, one can prove properties of programs such as “the power plant will never overheat” and “quicksort returns a sorted list for any input list.” Properties like these can never be claimed by just testing a program, no matter how extensive the testing (we cannot test quicksort for all possible lists). Furthermore, whereas a Maude specification defines a single well-defined mathematical object, the meaning of a C or Java program may depend on the compiler/interpreter used, so that the same program can behave differently on different machines, which is of course undesired in safety-critical systems such as airplanes.

Imperative programs manipulate the store quite directly through assembly-like “low-level” instructions. In declarative programs you don’t have to mess around with such low-level details, but this also means that you lose control over the memory management and of the execution. Declarative programs may therefore use more memory and time for executing a program than a fine-tuned imperative program. Maude has tried to minimize this disadvantage by a very sophisticated implementation which can perform millions of rewrites per second.

2.1 Hello World: Our First Maude Specifications

In this section we define and execute our first Maude specifications: one defining the natural numbers with the addition function, and one defining the Boolean values and the Boolean operations. Such data types are defined as many-sorted equational specifications (see e.g. [?]), which consist of a set of sorts, where each sort roughly corresponds to a data type, a set of function symbols (also called operators)—some of which are used to define the “values” of the data types, and others which are “ordinary” functions on these values—, and equations defining the functions.

In Maude, an equational specification is a functional module which is introduced with the following syntax:

\[ \text{fmod } \text{MODULENAME} \text{ is } \text{BODY} \text{ endfm} \]
2.1 Hello World: Our First Maude Specifications

where **MODULENAME** is the name of the module being introduced, and **BODY** is a set of declarations of sorts, function symbols, variables, and equations. Since **BODY** is a set of declarations, the order of the declarations does not matter. A comment starts with *** or --- and lasts until the end of the line, or it starts with *** ( or --- ( and lasts until the first matching occurrence of ’)’.

### 2.1.1 Natural Numbers with Addition

The following Maude module **NAT-ADD** specifies the natural numbers and a function ‘+’ on the natural numbers:

```maude
fmod NAT-ADD is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .
  vars M N : Nat .
  *** Define the addition function recursively:
  eq 0 + M = M .
  eq s(M) + N = s(M + N) .
endfm
```

This module declares a sort **Nat** and three functions symbols (or operators): 0, which does not take any arguments (such function symbols are called **constants**) and gives an element of sort **Nat**; s, which takes an element of sort **Nat** as argument and gives an element of **Nat**; and +, which takes two elements of sort **Nat** as arguments and “returns” a **Nat**-value. The underscore (‘_’) tells where the arguments should be placed in “mix-fix” notation. If there are no underscores (as is the case for s), then the function symbol must be written using “prefix” notation.

The function symbols define all possible **expressions**, or **ground terms**, in our system; some of the terms of the sort **Nat** are 0, s(0), s(s(0)), ..., 0 + 0, s(0) + s(s(0)), .... The function symbols 0 and s are declared to be **constructors** (ctor). The terms built up by the constructors, 0, s(0), s(s(0)), s(s(s(0))), ..., denote the **values** of **Nat**, and intuitively represent, respectively, the numbers 0, 1, 2, 3, ...

After declaring two variables **M** and **N** of sort **Nat**, the module defines the function + recursively by two equations. The variables **M** and **N** are **mathematical variables** as we know from equations such as \((x+y)^2 = x^2 + 2xy + y^2\); they are not “program variables” in the imperative programming sense that can be assigned values. Just like an equation \((x+y)^2 = x^2 + 2xy + y^2\) is usually applied from **left to right** to simplify an expression, Maude also applies the equations from left to right to simplify an expression until it cannot be further simplified. The variables in the equations say that the equations hold for all possible values for **M** and **N**. The equations define a **recursive function** for computing the sum \(m+n\) of two numbers **m** and **n**: if \(m\) is 0,
apply the first equation and we are done; if \( m \) has the form \( s(m') \), i.e., is greater than 0, the second equation recursively computes \( m' + n \) and adds one to this sum.

Assuming that you have installed Maude according to the instructions given at http://maude.cs.illinois.edu/download/, you can start Maude, and should then get a welcome from Maude that looks something like

--- Welcome to Maude ---

Maude 2.6 built: Dec 10 2010 11:12:39
Copyright 1997-2010 SRI International
Tue Jan 20 00:07:09 2015

Maude>

You now need to enter the module \texttt{NAT-ADD} into Maude. This can be done either by typing the specification directly on Maude’s command line (not recommended) or by writing the module in some file, say \texttt{nat-add.maude}, and then let Maude read this file by using the \texttt{in} command:

Maude> \texttt{in nat-add.maude}

Maude will then reply with:

-- fmod NAT-ADD

Maude>

If you get some error message(s) you should be aware of the following:

- Maude is case-sensitive. The sorts \texttt{Nat} and \texttt{nat} are not the same.
- Each declaration should end with a space followed by a period (’.’). However, there should not be a period after \texttt{endfm}.
- For infix symbols such as + there should be a space before and after +. The equation should be written \texttt{eq 0 + M = M }, not \texttt{eq 0+M = M }.
- There should be no space between \texttt{'} and \texttt{+}’ in the declaration of +.

To exit Maude, give the command \texttt{q} (or \texttt{quit}).

Maude’s \texttt{red} (or \texttt{reduce}) command computes the “value” of a given expression, such as 2 + 3, by using the equations “from left to right” to “replace equals for equals” until no equation can be used:

Maude> \texttt{red s(s(0)) + s(s(s(0)))}.

(Note the trailing period.) Maude answers with

\texttt{reduce in NAT-ADD : s(s(0)) + s(s(s(0)))}.
\texttt{rewrites: 3 in 0ms cpu (0ms real) (3000000 rewrites/second)
result Nat: s(s(s(s(0))))}.

The last line gives the result \( s(s(s(s(0)))) \) (representing the number 5) and states that this result has sort \texttt{Nat}.

\footnote{The command \texttt{load nat-add} does the same thing, but does not print the list of modules.}
2.1 Hello World: Our First Maude Specifications

2.1.2 The Boolean Values and Functions

The following module $\text{BOOLEAN}$ defines a data type for the Boolean values. The "values" in this data type are "true" and "false," which we represent by two constructor constants $\text{true}$ and $\text{false}$. We also declare the usual Boolean function $\text{not}$ (Boolean negation), $\text{and}$ (conjunction), and $\text{or}$ (logical disjunction) as follows:

fmod BOOLEAN is
  sort Boolean .
  ops true false : -> Boolean [ctor] .
  op not_ : Boolean -> Boolean [prec 53] .
  op _or_ : Boolean Boolean -> Boolean [prec 59] .
  var B : Boolean .
  eq not false = true .
  eq not true = false .
  eq true and B = B .
  eq false and B = false .
  eq true or B = true .
  eq false or B = B .
endfm

The actual names of sorts and operators do not matter; we can equally well use the sort name $\text{Bool}$ or $\text{TruthValues}$ instead of $\text{Boolean}$, and the constructors $1$ and $0$ (or $\text{T}$ and $\text{F}$) instead of $\text{true}$ and $\text{false}$.

In first-order logic there is a precedence between the function symbols, where e.g. negation binds tighter than conjunction, so that $\neg x \land y$ is read $(\neg x) \land y$. We can tell the Maude parser to impose a similar precedence on the function symbols by adding an attribute $\text{prec}$ to the function symbol declaration, where $n$ is a natural number. The lower the number of an operator, the higher its precedence! What matters is the relationship between the numbers: instead of 53, 55, and 59 we could have chosen 1, 2, and 3 with the same effect. A term $\text{true and not true or false}$ is now read $(\text{true and (not true)})$ or false, and reduces to false:

Maude> red true and not true or false .
result Boolean: false

2.1.3 Natural Numbers and Booleans: Less Than

A module may $\text{import}$ another module that has already been entered into Maude using the keyword $\text{including}$. For example, the following module imports both our previous modules to define the "less than" function on natural numbers:

fmod NAT< is
  including NAT-ADD .
  including BOOLEAN .
  op _<_ : Nat Nat -> Boolean .
  vars M N : Nat .
  eq 0 < s(M) = true .
  eq M < 0 = false .
endfm
eq s(M) < s(N) = M < N.
endfm

Exercise 1
1. Write the module NAT-ADD in a file, start Maude, and let Maude read the file with the specification.
2. Use Maude’s `red` command to compute $2 + 4$ and $(2 + 3) + 4$.

2.2 Many-Sorted Equational Specifications

In algebraic specifications we use sorts to distinguish different kinds of values, such as integers, strings, the Boolean values, and so on. In Maude sorts are declared using the keywords `sort` and `sorts`:

```
sort Int.
sorts Nat Boolean List.
```

The sorts are just names and do not contain any associated values such as “2” or “5”. We use function symbols (or operator symbols) to define the “elements” of each sort, and to define functions on the domains. In Maude, a declaration of a function symbol has the form

```
op f : s_1 ... s_n -> s.
```

for $n \geq 0$, where $f$ is the introduced function symbol, and $s_1, \ldots, s_n$, and $s$ are sorts. The list $s_1 \ldots s_n$ is the arity of $f$, and $s$ is its value sort (or coarity). Multiple function symbols with the same arity and value sort can be declared in one declaration:

```
ops f g h : s_1 ... s_n -> s.
```

We will use the terms “function symbol”, “function”, “operator symbol,” “operator,” and “operation” interchangeably.

Example 1 In the module NAT-ADD, the function symbol $0$ has the empty list as its arity and Nat as its value sort, the function $s$ has arity Nat and value sort Nat, and the symbol $+$ has arity Nat Nat and value sort Nat.

A function symbol whose arity is the empty list (i.e., $n = 0$) is called a constant.

A many-sorted algebraic signature consists of a set of sorts and a set of function symbol declarations:

**Definition 1 (Signature)** A many-sorted signature $(S, \Sigma)$ consists of a set $S$, whose elements are called sorts, and an $S \times S$-sorted family $\{\Sigma_{w,s} \mid w \in S^*, s \in S\}$ of function symbols. ($\Sigma_{w,s}$ is the set of function symbols with arity $w$ and value sort $s$.) We often write $f : w \rightarrow s \in \Sigma$ for $f \in \Sigma_{w,s}$. \hfill $\blacklozenge$
Example 2 The many-sorted signature \(\{\text{nat}, \Sigma\}\) defined by the module \texttt{NAT-ADD} has \(\Sigma = \{w_{\text{nat}} \mid w \in \{\text{nat}\}^*\}\) where \(\Sigma_{\text{nat.} \text{nat}} = \{0\}, \Sigma_{\text{nat. nat}} = \{s\}, \Sigma_{\text{nat. nat. nat}} = \{+\}\), and \(\Sigma_{\text{nat. nat. nat}} = \emptyset\) for all other \(w\)’s. (The empty list is denoted \(\varepsilon\).) The only constant in this signature is 0.

The **ground terms** define the “expressions” we can talk about. A ground term is built by constants and other function symbols in a “sort-correct” way:

**Definition 2 (Ground terms)** Given a many-sorted signature \((S, \Sigma)\), we can define the \(S\)-sorted set \(\mathcal{T}_S = \{\mathcal{T}_{S, s} \mid s \in S\}\) of ground terms inductively as follows:

1. \(\Sigma_{S, s} \subseteq \mathcal{T}_{S, s}\); that is, every constant of sort \(s\) is a ground term of sort \(s\).
2. If \(f \in \Sigma_{S_{n-1} \ldots s_n}\) and \(t_1 \in \mathcal{T}_{S, t_1}, \ldots, t_n \in \mathcal{T}_{S, t_n}\), and \(n \geq 1\), then \(f(t_1, \ldots, t_n) \in \mathcal{T}_{S, f}\).

That is, a function symbol “applied” to ground terms of the appropriate sorts gives another ground term.

3. In addition, each set \(\mathcal{T}_{S, s}\) is the smallest set satisfying the above conditions.

That is, only “things” which can be built from constants and the application of function symbols to ground terms of the right sorts are ground terms.

**Notation.** I sometimes use type-writer font and write ‘,’ ‘(‘, and ‘)’ instead of ‘,’ ‘(‘, and ‘)’. So that e.g. a term \(f(a, b)\) will also written \(f(a, b)\).

Example 3 The set \(\mathcal{T}_{\text{nat-ADD}}\) of ground terms of sort \text{nat} contains the ground terms 0, \(s(0)\), \(s(s(0))\), \(0 + 0\), \(s(0) + 0\), \(s(0) + \{s(0) + 0\}\), ...

Example 4 Given the signature

\begin{align*}
\text{sorts} & \quad s \quad s' \\
\text{ops} & \quad a \quad b : \rightarrow s \\
& \quad \text{op} \quad f : s \rightarrow s' \\
& \quad \text{op} \quad g : s \rightarrow s' \\
\end{align*}

Then, \(a\) and \(b\) and \(g(a, f(b))\) and \(g(g(a, f(b)), f(a))\) are ground terms of sort \(s\); and \(f(a)\) and \(f(b)\) and \(f(g(a, f(b)))\) are ground terms of sort \(s'\). Neither \(a\) nor \(b\) nor \(f(a, b)\) nor \(q(r, \ldots)\) is a ground term of sort \(s'\).

When a definition mentions “all terms of the form \(f(t_1, \ldots, t_n)\) for \(n \geq 0\),” then this also includes all the constants \((n = 0)\).

As already mentioned, we separate between **constructor** functions, whose job is to define the elements of the data type, and the other, called **defined** functions, that are supposed to be ordinary functions on those elements. More precisely, the elements of a sort are those ground terms consisting only of constructor functions. The other functions are defined recursively by equations.

**Variables** of different sorts are needed to define equations in a convenient way:

**Definition 3 (Variables)** Given a many-sorted signature \((S, \Sigma)\), a variable set \(X\) is an \(S\)-sorted family \(X = \{X_s \mid s \in S\}\) of pairwise disjoint sets (that is, no variable has two different sorts: \(s \neq s' \implies X_s \cap X_{s'} = \emptyset\)), also disjoint from \(\Sigma\) (that is, nothing can be both a variable and a function symbol). We will often write \(x : s\) for \(x \in X_s\).

In Maude, the keywords \texttt{var} and \texttt{vars} are used to declare variables. However, variables of the form \texttt{var:sort} can also be used on-the-fly without explicit declaration, so that the following two specification fragments are equivalent:
vars M N : Nat.
eq 0 + M = M.
eq s(M) + N = s(M + N).

and

\eq 0 + M : \text{Nat} = M : \text{Nat}.
\eq s(M : \text{Nat}) + N : \text{Nat} = s(M : \text{Nat} + N : \text{Nat}).

("Non-ground") terms can contain variables: The set \( T_{S}(X) \) of terms in a signature \((S, \Sigma)\) w.r.t. a set of variables \( X \) are all the "things" that can be built in a sort-consistent way from constants, variables, and the application of functions:

Definition 4 (Terms) Given a many-sorted signature \((S, \Sigma)\) and a variable set \( X = \{X_{i} \mid s \in S\} \), the \( S \)-sorted set of terms \( T_{S}(X) = \{T_{S,i}(X) \mid s \in S\} \) is defined inductively by the following conditions:

1. \( X_{i} \subseteq T_{S,i}(X) \) for \( s \in S \); that is, a variable of sort \( s \) is also a term of sort \( s \).
2. \( \Sigma_{e,s} \subseteq T_{S,i}(X) \) for \( s \in S \); that is, a constant of sort \( s \) is also a term of sort \( s \).
3. \( f(t_{1}, \ldots, t_{n}) \in T_{S,i}(X) \) if \( f \in \Sigma_{i,\ldots,n,s} \) and \( t_{1}, \ldots, t_{n} \in T_{S,i}(X) \) for each \( 1 \leq i \leq n \).
4. \( T_{S}(X) \) is the smallest \( S \)-sorted set satisfying the above conditions.

Non-constructor functions are defined recursively by (unconditional and conditional) equations:

Definition 5 (Equations) Given a many-sorted signature \((S, \Sigma)\), a \((\Sigma-)\) equation is a triple \((X, t, t')\), written \((\forall X) \ t = t'\), where \( X \) is an \( S \)-sorted variable set disjoint from \( \Sigma \), and \( t \) and \( t' \) are terms of the same sort; i.e., \( t, t' \in T_{S,i}(X) \) for some \( s \in S \).

A conditional \((\Sigma-)\) equation is a \( 2(n + 1) + 1\)-tuple \((X, u_{1}, v_{1}, \ldots, u_{n}, v_{n}, t, t')\) for \( n \geq 1 \), written

\[(\forall X) \ u_{1} = v_{1} \land \ldots \land u_{n} = v_{n} \implies t = t',\]

such that there are sorts \( s_{1}, \ldots, s_{n}, s \) in \( S \) with \( t, t' \in T_{S,i}(X) \) and \( u_{1}, v_{1} \in T_{S,i}(X) \) for each \( i \in \{1, \ldots, n\} \).

Definition 6 (Many-sorted equational specifications) A many-sorted equational specification is a tuple \((S, \Sigma, E)\) where \((S, \Sigma)\) is a many-sorted signature and \( E \) is a set of \( \Sigma \)-equations and conditional \( \Sigma \)-equations.

In Maude, equations are written with syntax

\[ \eq t = t' . \]

and conditional equations are written with syntax

\[ \ceq t = t' \text{ if } u_{1} = v_{1} \land \ldots \land u_{n} = v_{n} . \]

Informally, the “mathematical” meaning of an equation \((\forall X) \ t = t'\) is that \( t \) and \( t' \) are equivalent for all “values” of the variables \( X \). For example, the equation \((\forall M : \text{Nat}) \ 0 + M = M\) means that \( 0 + 0 = 0 \) and \( 0 + s(0) = s(0) \), and so on. Similarly, \((\forall X) \ u_{1} = v_{1} \land \ldots \land u_{n} = v_{n} \implies t = t'\) means that if \( u_{1} = v_{1} \) and \( \ldots \) and \( u_{n} = v_{n} \) for some values of the variables in \( X \), then \( t \) equals \( t' \) for those values of the variables.

The operational meaning describes how Maude’s \texttt{eq} command computes with equations. For example, if we ask Maude to compute the “value” of a ground term such as e.g., \( s(s(0 + s(0))) + 0 \), then the following happens:
1. Maude checks whether some equation can be applied somewhere in the term. That is, it checks whether the left-hand side of an equation “matches” the term somewhere. It then applies the equation by “replacing equal by equal.” In the example above, the equation \( 0 + M = M \) could be applied to the term \( s(s(0 + s(0))) + 0 \), reducing it to \( s(s(s(0))) + 0 \). If more than one equation can be applied, and/or if an equation can be applied in more than one place in a term, then the system chooses (pseudo-)arbitrarily what equation to apply and where to apply it. For example, in addition to the previous application, the equation \( s(M) + N = s(M + N) \) could be applied to \( s(s(0 + s(0))) + 0 \), giving \( s(s(0 + s(0))) + 0 \).

2. The above process is repeated on the resulting term as long as there is some equation which can be applied.

3. When no equation can be applied anywhere, Maude outputs the “current” term.

**Example 5** The term \( s(s(0 + s(0))) + 0 \) can reduce to \( s(s(s(0))) + 0 \) in NAT-ADD. In the next step, only the equation \( s(M) + N = s(M + N) \) can be applied, giving \( s(s(s(0))) + 0 \). In the next step, only this same equation can be applied, giving \( s(s(s(0))) \). In the next step, this same equation can be applied, giving \( s(s(s(0))) \). Now, only the equation \( 0 + M = M \) can be applied, giving the term \( s(s(s(0))) \). Now, no more equation can be applied, and therefore the result is \( s(s(s(0))) \). The sequence \( s(s(0 + s(0))) + 0 \leadsto s(s(s(0))) + 0 \leadsto ... \leadsto s(s(s(0))) \) is called a derivation, a computation, or a reduction sequence.

**Exercise 2** Overloading a function symbol means that the same function symbol can have different arities and/or value sorts. This can be quite convenient, since a constant \( 0 \) could be both a bit value, a Boolean value, and a natural number:

```plaintext
sorts Bit Boolean Nat .
ops 0 1 : -> Bit .
ops 0 1 : -> Boolean .
ops 0 : -> Nat .
```

1. Is such overloading allowed according to Definition 1?
2. If it is allowed, how can you modify Definition 1 to disallow such overloading?

**Exercise 3** In the signature given in Example 4,

1. explain why the terms \( a \) and \( b \) and \( g(a, f(b)) \) and \( g(g(a, f(b)), f(a)) \) are ground terms of sort \( s \);
2. explain why neither \( f(a) \) nor \( g(,,..) \) is a ground term of sort \( s \);
3. is \( f(f(a)) \) a ground term of sort \( s \) or sort \( s' \)?

**Exercise 4**

1. Retrace the derivation given in Example 5. That is, show exactly how and where the equations are applied in each step.
2. Show a derivation from \( s(s(0 + s(0))) + 0 \) where the equation \( s(M) + N = s(M + N) \) is applied in the first step.
2.3 Requirements of Equational Specifications

This section introduces four requirements that an equational specification must satisfy to make Maude computations meaningful. Chapters 4 and 5 explain how two of these requirements, termination and confluence, can be analyzed.

2.3.1 One-to-one Constructor Basis

A data type consists of a set of elements (the domain) and a set of functions on those elements. Examples of domains are the set $\mathbb{N}$ of natural numbers, the set $\mathbb{Z}$ of integers, the set of all lists of natural numbers, the set of all binary trees of a certain kind, and so on.

In Maude, the elements in a data type are represented by the ground terms built by the constructor function symbols. For this to make sense, each element in the domain should be represented by one such constructor ground term, and vice versa. (Formally, there should exist a surjective and injective function from the intended domain to the set of constructor ground terms of the corresponding sort.) This implies that: (i) each element in the domain we want to model is represented by a constructor ground term; (ii) there is no “confusion”: each element is only represented by one constructor ground term; and (iii) there are no “junk” constructor ground terms that do not represent any elements.

For the natural numbers and their Maude representation in the module \texttt{NAT-ADD} we have the desired one-to-one correspondence: each number $n \in \mathbb{N}$ is represented by a constructor ground term $s(s(\ldots(s(0))\ldots)$; and a constructor ground term of sort \texttt{Nat} is either $0$ (representing the number 0) or has the form $s(s(\ldots(s(0))\ldots)$, for $m \geq 1$, which represents the number $m$.

Likewise, there is a one-to-one correspondence between the truth values “true” and “false” and constructor ground terms $\texttt{true}$ and $\texttt{false}$ in \texttt{BOOLEAN}.

2.3.2 Termination (No Infinite Looping)

To use Maude to compute the value of an expression, the computation of any expression must terminate; i.e., there should not exist infinite derivations (“infinite loops”) from any ground term. For example, in the module \texttt{NAT-ADD}, no matter how the equations to apply are chosen, each computation would always end up with a term to which no equation applies. However, in a specification

\begin{verbatim}
sort s .
ops a b : -> s .
eq a = b .
eq b = a .
\end{verbatim}
the system would “simplify” $a$ to $b$ using the first equation, and then $b$ would be
simplified to $a$ using the second equation, and then $a$ would again be simplified to $b$
using the first equation, and so on, giving an infinite computation

$$a \leadsto b \leadsto a \leadsto b \leadsto \ldots$$

starting from $a$. Similarly, adding the equation $eq M + N = N + M$. to NAT-ADD
could lead to infinite computations such as $s(s(0)) + s(0) \leadsto s(0) + s(s(0))$
$\leadsto s(s(0)) + s(0) \leadsto \ldots$.

A specification is called terminating if it does not allow any infinite computation.

A simple rule of thumb is that the value in some argument position in the re-
cursive calls must decrease in some way$^2$; other arguments may become larger.
Furthermore, an equation can have multiple recursive calls, as long as the appropri-
ate argument decreases in all recursive calls. For example, the module NAT-ADD
extended with a function $op f : Nat \ Nat \rightarrow Nat$ defined by

$$eq f(0, M) = s(s(M)) .$$
$$eq f(s(M), N) = f(M, M + N) + f(M, N) .$$

is terminating, since the first argument of $f$ decreases in each recursive call. How-
ever, if the second equation is replaced by

$$eq f(s(M), N) = f(M, M + N) + f(N, M) .$$

then the specification would no longer be terminating.

### 2.3.3 Uniqueness of the “Result”

By definition, a function $f : A \rightarrow B$ assigns a single value $b \in B$ to each $a \in A$.
Therefore, since we are computing the value of functional expressions, the “result”
of the computation should be the same, no matter how Maude applies the equations.
For example, any computation of $s(s(0 + s(0))) + 0$ should always end with the
result $s(s(s(0)))$, and not with $s(s(s(0))) + 0$ or $s(0)$ or anything else. (Since
we have no control over the application of equations, it would be unsatisfactory if
the result of computing the value of an expression would depend on how the Maude
systems internally chooses which equations to apply.)

**Example 6** In the following terminating specification

```plaintext
sort s .
ops a b c : -> s .
eq a = b . 
eq a = c .
```

the term $a$ does not have a unique result, since it can be simplified to both $b$ and $c$.

\[\blacktriangle\]

$^2$ A “decrease” typically means that the number of function symbol occurrences in a constructor
ground term must decrease.
A result of a computation of a term $t$ is called a normal form of $t$. If it is in addition unique, then this unique normal form is written $t!$. For example, the normal form of $s(s(0 + s(0))) + 0$ is $s(s(s(0)))$ in the module NAT-ADD.

The property that the computation of an expression (in a terminating specification) gives the same result no matter how the equations are applied is formalized as the confluence property in Chapter 5.

2.3.4 Definedness: The Result Should be a Constructor Term

We want to compute the value (i.e., a constructor ground term) of a functional expression (i.e., a ground term). Therefore, each expression should be reducible to a constructor ground term. For example, if we “forget” the equation $0 + M = M$ in NAT-ADD, then $s(s(0 + s(0))) + 0$ reduces to $s(s((0 + s(0)) + 0))$, which cannot be further reduced, and which is not the result we really wanted.

This is the same as requiring that a non-constructor function is “defined” on all constructor ground terms. For instance, for natural numbers, $n_1 + n_2$ is defined for all values/constructor ground terms $n_1$ and $n_2$, since $n_1$ (and $n_2$ as well for that matter) should have the form $0$ or $s(n)$ for some $n$. In the first case, the equation $0 + M = M$ will apply, and in the second case $s(M) + N = s(M + N)$ can be applied.

Functions are often defined by having one equation for each constructor, although sometimes we need fewer or more equations:

```maude
op double : Nat -> Nat . var N : Nat . eq double(N) = N + N .
```

The above equation covers all arguments of double. A function minusTwo which decreases any number greater than one by two can be defined by three equations:

```maude
op minusTwo : Nat -> Nat . var N : Nat .
eq minusTwo(0) = 0 .
eq minusTwo(s(0)) = 0 .
eq minusTwo(s(s(N))) = N .
```

For any constructor ground term $n$, some equation can be applied on minusTwo($n$).

The function $<_$ in Section 2.1.3 is defined for all pairs $(m,n)$ of constructor ground terms $m$ and $n$; this can be checked by considering all possible values for this pair: $(0,0), (0,s(n)), (s(m),0)$, and $(s(m), s(n))$. In each of these cases, an equation defining $<_$ can be applied. For example, the equation $eq M < 0 = false$ can be applied to any term of the form $s(m) < 0$.

A more precise name for the definedness property is sufficient completeness: the final result of a computation of a ground term should be a constructor ground term.

2.3.5 Maude and the Requirements

Maude does not check whether your specification satisfies these requirements. The first one obviously cannot be checked, since Maude cannot know what domain you
are trying to represent in Maude; the name of a sort does not have a “predefined” meaning. The other three requirements are in general undecidable. That is, there is no algorithm that can look at any user module and always tell whether the module satisfies the requirements or not. However, Maude has (external) termination checkers [?], confluence checkers [?], and sufficient completeness checkers [?] that can often be used to check those respective requirements.

You must make sure that the above requirements are satisfied independently of how Maude is implemented. Since we have no control over the application of equations, it would be unsatisfactory if the result of computing a term would depend on how the Maude system internally chooses which equations to apply. Furthermore, it would also leave the mathematical reasoning about the models a mess.

Exercise 5 Explain why there can be no infinite computations in the modules NAT-ADD and NAT<.

Exercise 6 Explain why there could be an infinite computation in NAT-ADD extended with the above equation eq f(s(M), N) = f(M, M + N) + f(N, M).

2.4 Many-Sorted Specification of Data Types

This section defines some data types as many-sorted equational specifications.

2.4.1 Defining Functions: Getting Started

Although there is no automatic way to define functions, one hint to help get you quickly started is to define a function \( \text{op} \ f : S \rightarrow S' \) by one (or more) equation(s) for each constructor of \( S \). For example, if the constructors for the sort \( S \) are two constants \( a \) and \( b \), one unary operator \( g \) (i.e., a function taking one argument), and one binary operator \( h \) (i.e., a function taking two arguments), then one could first try to define \( f \) by four equations of the form

\[
\begin{align*}
\text{eq } f(a) &= \ldots \\
\text{eq } f(b) &= \ldots \\
\text{eq } f(g(X)) &= \ldots \\
\text{eq } f(h(X, Y)) &= \ldots
\end{align*}
\]

for variables \( X \) and \( Y \) of appropriate sorts. For the sort \( \text{Nat} \) the constructors are \( 0 \) and \( s \). Therefore, we can follow this scheme to define the function \( \text{double} \), which doubles its argument, also without using +:

\[
\begin{align*}
\text{eq double}(0) &= 0 \\
\text{eq double}(s(N)) &= s(s(\text{double}(N)))
\end{align*}
\]

If the function \( f \) takes two arguments, you can define \( f \) by “case” on the constructors for one of the arguments, or for both. \( \text{NAT-ADD} \) defines addition by “case” on the first argument, but it could equally well have used the second argument. We can use this technique to define multiplication by “case” on the first argument:
fmod NAT-MULT is including NAT-ADD.
  op _*_ : Nat Nat -> Nat.
  vars M N : Nat.
  eq 0 * N = 0.
  eq s(M) * N = N + (M * N).
endfm

For such binary functions (or more generally, n-ary) functions, sometimes such case definitions only work for one of the arguments (like list concatenation that we will see later). However, sometimes we may need to do a "case" on both arguments. For example, for less-than on natural numbers, we need to consider both arguments: the first argument may be 0 or have the form \( s(m) \), and the second argument is either 0 or has the form \( s(n) \):

\[
\begin{align*}
\text{eq } 0 < 0 &= \text{false} . \\
\text{eq } s(M) < 0 &= \text{false} . \\
\text{eq } 0 < s(N) &= \text{true} . \\
\text{eq } s(M) < s(N) &= M < N .
\end{align*}
\]

Again, this is just help to get you started; once you have defined your function, you should make its definition more elegant: the upper two equations can be combined into the single equation \( \text{eq } M < 0 = \text{false} \), yielding the definition in Section 2.1.3. While this is a useful starting point, sometimes you need more elaborate definitions, such as for the function \text{minusTwo} above.

An important thing discussed next is that it is often convenient, or even necessary, to introduce auxiliary functions in order to define a given function.

### 2.4.2 Expressiveness of Many-Sorted Equational Specifications

Bergstra and Tucker showed in [?] that it is impossible to define the square function on natural numbers in Maude without using other functions than 0 and \( s \). And try to define a function for exponentiation without using other functions than addition and exponentiation! However, both the square function and exponentiation are easily defined if you introduce (addition and) multiplication as auxiliary functions:

fmod NAT-EXP is including NAT-MULT.
  op square : Nat -> Nat.
  op _**_ : Nat Nat -> Nat.
  vars M N : Nat.
  eq square(N) = N * N.
  eq M ** 0 = s(0).
  eq M ** s(N) = M * (M ** N).
endfm

What does this difficulty of defining simple functions without introducing auxiliary functions say about the expressive power of terminating and confluent finitary many-sorted equational specifications? It turns out that by adding a finite number of auxiliary functions, you can define whatever you want in this way. For example,
Section 4.1 and Exercise XXX show that Turing machines and register machines can be easily expressed/simulated by many-sorted equational specifications. However, the corresponding specifications may not be terminating (since the corresponding Turing or register machine may not terminate).

More formally, any recursive (i.e., computable) function on finite products of natural numbers can be defined by a terminating and confluent finitary many-sorted equational specification (see, e.g., [?, Section 3.2]). Furthermore, Bergstra and Tucker proved the following remarkable result in [?, ?] (see also the discussion in [?]):

**Theorem 2.1.** Any computable algebra\(^4\) can be specified by a finitary terminating and confluent many-sorted equational specification.

This means that anything you can do in your favorite programming language, you can also do in Maude! Just add auxiliary functions (new sorts are not needed).

### 2.4.3 Maude Specifications of Some Data Types

This section shows the Maude specification of some well known data types that satisfy the four requirements in Section 2.3.

#### 2.4.3.1 Lists

How can lists of, say, natural numbers, be represented in a many-sorted equational specification? A constructor for the empty list is obviously needed:

```plaintext
sort List.
op nil : -> List [ctor].
```

A natural way of constructing lists is by appending an element to an existing (possibly empty) list:

```plaintext
op app : List Nat -> List [ctor].
```

In this case, a list “1 2 3” is represented by the constructor term

```plaintext
app(app(app(nil, s(0)), s(s(0))), s(s(s(0))))
```

A more appealing way of representing lists is to let the append function instead be a mix-fix function symbol:

```plaintext
op _++_: List Nat -> List [ctor].
```

---

\(^3\) That is, using only a finite number of sorts, functions, and equations.

\(^4\) A computable algebra is one whose carriers are recursive sets (i.e., we can decide whether an element is a member of the set) and whose functions are recursive (i.e., computable) functions.
Then, the list “1 2 3” can be written nil ++ s(0) ++ s(s(0)) ++ s(s(s(0))). We can further shorten the representation of lists by removing the "++" part from the above append function; i.e., by using mix-fix empty syntax:

\[
\text{op } __ : \text{List Nat } \rightarrow \text{List [ctor]} .
\]

The list “1 2 3” is now represented by the term nil s(0) s(s(0)) s(s(s(0))).

The following module then defines lists of natural numbers and some functions on such lists.\(^5\)

\[
\text{fmod LIST-NAT1 is protecting NAT1 . protecting BOOLEAN1 .}
\]

\[
\text{sort List .}
\]

\[
\text{op nil : } \rightarrow \text{List [ctor] .}
\]

\[
\text{op __ : List Nat } \rightarrow \text{List [ctor] .}
\]

\[
\text{op length : List } \rightarrow \text{Nat .} \quad \text{*** } \# \text{ of elements in a list}
\]

\[
\text{op concat : List List } \rightarrow \text{List .} \quad \text{*** Concatenate two lists}
\]

\[
\text{op insertFront : Nat List } \rightarrow \text{List .} \quad \text{*** Insert element first}
\]

\[
\text{ops first last : List } \rightarrow \text{Nat .} \quad \text{*** First/last element}
\]

\[
\text{op empty? : List } \rightarrow \text{Boolean .} \quad \text{*** Is the list empty?}
\]

\[
\text{op rest : List } \rightarrow \text{List .} \quad \text{*** Remove first element.}
\]

\[
\text{op reverse : List } \rightarrow \text{List .} \quad \text{*** Reverse list}
\]

\[
\text{op _occursIn_ : Nat List } \rightarrow \text{Boolean .} \quad \text{*** Remove element(s)}
\]

\[
\mathsf{eq} \quad \text{max : List } \rightarrow \text{Nat .} \quad \text{*** Largest element in list}
\]

\[
\text{op isSorted : List } \rightarrow \text{Boolean .} \quad \text{*** Is the list sorted?}
\]

\[
\text{vars N N’ : Nat .} \quad \text{vars L L’ : List .}
\]

The length function, giving the number of numbers in the list, can be defined using the techniques suggested above; i.e., by recursion on the argument w.r.t. the constructors nil and __:

\[
\text{eq length(nil) = 0 .}
\]

\[
\text{eq length(L N) = s(length(L)) .}
\]

To define the list concatenation function \text{concat}, it turns out that doing the recursion on the second argument works:

\[
\text{eq concat(L, nil) = L .}
\]

\[
\text{eq concat(L, L’ N) = concat(L, L’) N .}
\]

The function first gives the value of the first element in the list. But what is the first element in an empty list? The function first is a partial function that is not defined on all lists, but only on non-empty lists. Partial functions are treated in Sections 2.5 and 2.6; in the meantime we just define that the first element in an empty list is 0:

\[
\text{eq first(nil) = 0 .} \quad \text{*** Default/error value}
\]

\[
\text{eq first(nil N) = N .}
\]

\[
\text{eq first(L N N’) = first(L N) .}
\]

\(^5\) The modules \text{NAT1} and \text{BOOLEAN1} are defined in Exercise 9.
2.4.3.2 Binary Trees

A binary tree whose nodes are (labeled with) natural numbers can be represented by the following constructors:

- `sort BinTree`.
- `op empty : -> BinTree [ctor]`.

where `bintree(t, n, t')` represents the tree with root labeled `n` which has `t` as its left subtree and `t'` as its right subtree. For example, the tree in Fig. 2.1 can be represented by the term

```
4
\downarrow
7
```

Fig. 2.1: A (small) binary tree.

```
bintree(empty, s(s(s(0)))),
bintree(empty, s(s(s(s(s(s(s(0))))))), empty)
```

It is easy to see that each binary tree can be represented by a unique constructor ground term of sort `BinTree` and that each such term represents a binary tree.

The following module defines a data type for binary trees:

```
fmod BINTREE-NAT1 is protecting LIST-NAT1.
sort BinTree.
  op empty : -> BinTree [ctor].
  op bintree : BinTree Nat BinTree -> BinTree [ctor].
  ops preorder inorder postorder : BinTree -> List.
  ops size weight : BinTree -> Nat.
  op isSearchTree : BinTree -> Boolean.
  op reverse : BinTree -> BinTree.
vars BT BT' : BinTree. vars N N' : Nat.
  eq preorder(empty) = nil.
  eq preorder(bintree(BT, N, BT')) =
    insertFront(N, *** Root first, then left and right subtree: concat(preorder(BT), preorder(BT'))).
  eq size(empty) = 0.
  eq size(bintree(BT, N, BT')) = s(size(BT) + size(BT')).
  ...
endfm
```

The functions `preorder`, `inorder`, and `postorder` list the elements of a tree in the order they are encountered in, respectively, a preorder, an inorder, and a postorder.
traversal of the tree. \texttt{weight} gives the sum of the elements in the tree, \texttt{size} gives the number of elements, and \texttt{isSearchTree} returns \texttt{true} if and only if the tree is a binary search tree; that is, an inorder traversal (“from left to right”) encounters the elements in increasing (or at least non-decreasing) order). The function \texttt{reverse} reverses the tree; i.e. “flips it” around its vertical axis, and then does the same recursively for each subtree.

### 2.4.3.3 What About Sets?

Sets and multisets (which are essentially “sets,” but where the number of occurrences of each element matters) are important data types. However, since the sets \{\texttt{a,b}\} and \{\texttt{b,a}\} are the same sets, it is hard to define a one-to-one constructor basis. For example, using constructors

\begin{verbatim}
op emptySet : -> Set [ctor] .
op _;_ : Set Nat -> Set [ctor] .
\end{verbatim}

the same set \{0,1\} = \{1,0\} could be represented by the two different constructor ground terms \texttt{0 ; s(0)} and \texttt{s(0) ; 0}. Section 2.8.3 defines multisets and sets with one-to-one constructors.

---

**Exercise 7** Define a function \texttt{square : Nat -> Nat} that computes the square of a number, without using any other function except \texttt{s}, \texttt{0}, \texttt{+}, and \texttt{square} itself.

**Exercise 8** Explain why parentheses are not needed when using the constructors \texttt{nil} and \texttt{__} for lists. That is, show that expressions such as \texttt{nil s(0) s(s(0))} \texttt{s(s(s(0)))} only can be parsed in one way.

**Exercise 9** 1. Define a module \texttt{NAT1} that extends \texttt{NAT<} with the following functions:

\begin{verbatim}
op half : Nat -> Nat .
ops _monus_ diff max min : Nat Nat -> Nat .
ops _<=_ _>_ _>=_ _==_ : Nat Nat -> Boolean .
\end{verbatim}

where \texttt{half} is “integer division by 2,” \texttt{monus n} is “minus down to 0,” i.e., \texttt{max(m n,0), diff is the difference between two numbers, max and min is the greatest, resp. smallest, of two numbers, and odd and even is true if its argument is an odd, resp., even, number. The other functions are the usual comparison functions.}

2. Define a module \texttt{BOOLEAN1} that extends \texttt{BOOLEAN} with the following functions:

\begin{verbatim}
op if_then_else_fi : Boolean Boolean Boolean -> Boolean .
\end{verbatim}

where \texttt{x implies y is false only when x is true and y is false.}

Test your specifications in Maude. Two things to remember is that: (i) since there are also built-in Boolean values in Maude, you must give the Maude command \texttt{set protect BOOL off .} before entering the specifications into Maude; and (ii) you
must have loaded the files containing the modules that you import. This can be achieved by starting your file as follows (for the appropriate file names):

```maude
set include BOOL off .
load nat-mult.maude
load boolean.maude
load less-than.maude
```

**Exercise 10** Define the other functions in the module LIST-NAT1.

**Exercise 11** Lists of natural numbers can be compared lexicographically, similar to the order in a phone book. A list \( l \) is greater than a list \( l' \) if there is a number \( i \) such that

- the \( i \)th element of \( l \) exists, and it is greater than the \( i \)th element in \( l' \) or the \( i \)th element in \( l' \) does not exist; and
- for all \( j < i \), the \( j \)th element of \( l \) is the same as the \( j \)th element of \( l' \).

In short, \( l \) is greater than \( l' \) if both lists are the same until either \( l' \) stops or until an element in \( l \) is greater than the corresponding element in \( l' \). For example, the list “4 5 6” is greater than both “3 4 5 6 7”, “4 5”, and “4 5 2 10”.

1. Show (by an example) that there is an infinite sequence \( l_0 > l_1 > l_2 > l_3 > \ldots \) of lists such that \( l_i \) is greater than \( l_{i+1} \) for each \( i \).
2. Explain informally why there is no infinite sequence \( l_0 > l_1 > l_2 > l_3 > \ldots \) of lists of the same length such that \( l_i \) is greater than \( l_{i+1} \) for each \( i \).
3. Define a function

```maude
op _greaterThan_ : List List -> Boolean .
```

which compares two lists lexicographically, and test your definition in Maude.

**Exercise 12** Represent the binary tree in Fig. 2.2 as a term of sort BinTree.

![Binary Tree](image)

Fig. 2.2: A binary tree.

**Exercise 13** Define in Maude the remaining functions in the module BINTREE-NAT1.
2.5 Order-Sorted Equational Specifications

Different sorts are not related in any way in the many-sorted world. This hardly seems practical. For example, it is natural to have a sort \( \text{Nat} \) for the natural numbers and a sort \( \text{Int} \) for the integers. In the many-sorted world these sorts are unrelated. Using only the sort \( \text{Int} \) and forgetting about \( \text{Nat} \) is not very elegant, since some functions, such as the factorial function, are partial functions on the integers that do not take negative numbers as arguments. We have seen other partial functions, such as \( \text{first}, \text{last}, \text{and rest} \) on lists, which should only be defined on non-empty lists. To have two unrelated sorts \( \text{Int} \) and \( \text{Nat} \) is unsatisfactory as well, since it requires functions used both for natural numbers and integers to be declared and defined twice, and does not allow the use of a natural number in place of an integer.

Maude supports order-sorted specifications (see e.g. \cite{?}), in which a sort may have subsorts. Intuitively, a subsort declaration

\[
\text{subsort } s' < s .
\]

means that the sort \( s' \) is “included” in the sort \( s \) in the sense that each element of \( s' \) is also an element of \( s \). For example, since the natural numbers are a subset of the integers, it is natural to have a subsort \( \text{subsort } \text{Nat} < \text{Int} . \) Multiple subsort declarations can be combined into a single one: \( \text{subsorts } \text{Nat} \text{ Neg} < \text{Int} \), which states that both \( \text{Nat} \) and \( \text{Neg} \) are subsorts of \( \text{Int} \). (A subsort declaration does not declare the sorts, so the above sorts must also have been declared as usual).

Formally, in an order-sorted signature, the set of sorts is equipped with a partial order \( \leq \). The subsort relation \( \leq \) induces a subsort relation \( \leq \) on lists of sorts of the same length, where \( s_1 \ldots s_n \leq s'_1 \ldots s'_n \) holds if and only if \( s_i \leq s'_i \) for each \( 1 \leq i \leq n \).

If \( \text{Nat} \) is a subsort of \( \text{Int} \), a function which takes \( \text{Int} \) arguments will also accept \( \text{Nat} \) arguments, since any \( \text{Nat} \) value is also a \( \text{Int} \) value. For example, a function

\[
\text{op } _+ : \text{Int} \text{ Int} \rightarrow \text{Int} .
\]

also applies to natural numbers. One could add a declaration

\[
\text{op } _+ : \text{Nat} \text{ Nat} \rightarrow \text{Nat} .
\]

to tell Maude that the value of \( m + n \) has sort \( \text{Nat} \) if both \( m \) and \( n \) have sort \( \text{Nat} \). As explained in Section 2.6, such declarations of subsort overloaded functions are only needed for constructors, to ensure that each (sub)sort has the desired domain.

An order-sorted signature is just a many-sorted signature with an additional partial order \( \leq \) on the sorts:

Definition 7 (Order-sorted signature) An order-sorted signature \( (S, \leq, \Sigma) \) consists of a set \( S \) (of sorts), a partial order \( \leq \) on \( S \), and an \( S^* \times S\)-sorted family \( \Sigma = \{ \Sigma_{w,s} \mid w \in S^*, s \in S \} \) of “function symbol declarations.”

Terms are defined as expected: if \( s' \leq s \), then any term of sort \( s' \) is also a term of sort \( s \).

Definition 8 (Terms in order-sorted signatures) Given an order-sorted signature \( (S, \leq, \Sigma) \) and a variable set \( X = \{ X_s \mid s \in S \} \), the \( S \)-sorted set of terms \( \mathcal{F}_S(X) = \{ \mathcal{F}_S(X) \mid s \in S \} \) is defined by “adding” the following condition

Be sure to define partial orders!
0. \( T_{S',X}(X) \subseteq T_{S,X}(X) \) if \( S' \leq S \); that is, a term of a subsort \( S' \) is also a term of the supersort \( S \).

to Definition 4, which defines the terms in a many-sorted signature.

The set of ground terms is defined as expected: \( T_S = \{ T_{S,s} \mid T_{S,s} = T_{S,s}(\emptyset), s \in S \} \).

The following example shows that the sort of a term could be ambiguous in the sense of the term having completely unrelated sorts, which is of course undesired:

\[
\begin{align*}
\text{sorts } & s_1 \ s_2 \ s_{12} \ u_1 \ u_2 . \\
\text{subsorts } & s_{12} < s_1 \ s_2 . \\
\text{op } & a : \rightarrow s_1 . \quad \text{op } b : \rightarrow s_2 . \quad \text{op } c : \rightarrow s_{12} . \\
\text{op } & f : s_1 \rightarrow u_1 . \quad \text{op } f : s_2 \rightarrow u_2 . \quad \text{op } h : u_1 \rightarrow u_1 .
\end{align*}
\]

What is the sort of the term \( f(c) \)? Since \( c \) is an element of sort \( s_1 \), the term \( f(c) \) should have sort \( u_1 \), but since \( c \) is also an element of sort \( s_2 \), the term \( f(c) \) should have sort \( u_2 \). Such ambiguity is not desired since \( u_1 \) and \( u_2 \) are totally unrelated (is, e.g., \( h(f(c)) \) a term?). Maude therefore requires that each non-constant term has a unique least sort as explained below. The specification would be OK if we added

\[
\begin{align*}
\text{sort } & u_{12} . \quad \text{subsorts } u_{12} < u_1 \ u_2 . \quad \text{op } f : s_{12} \rightarrow u_{12} .
\end{align*}
\]

since the smallest sort of \( f(c) \) would be \( u_{12} \).

**Definition 9 (Least sort)** A term \( t \in T_{S,X}(X) \) has a unique least sort if the set \( \{ s \mid t \in T_{S,s}(X) \} \) of sorts of \( t \) has a unique smallest element w.r.t. \( \leq \), in which case this unique least sort of \( t \) is denoted \( LS(t) \).

Order-sorted signatures should be prerregular, which ensures that each non-constant term has a unique least sort.

**Definition 10 (Prerregular signatures)** An order-sorted signature \( (S, \leq, \Sigma) \) is prerregular if for any function symbol declaration \( f : s_1 \ldots s_n \rightarrow s \in \Sigma \) with \( n \geq 1 \), and any sequence \( s'_1 \ldots s'_n \) with \( s'_i \leq s_i \) for all \( i \), the term \( f(x_1, \ldots, x_n) \), where \( x_i \) is a variable of sort \( s'_i \) for each \( i \), has a unique least sort.

**Example 7** We may use 0 for both false in a sort Boolean and for the value 0 in a sort Nat, and may use + for disjunction (“or”) on the Booleans and for addition on the natural numbers. The signature

\[
\begin{align*}
\text{fmod OVERLOADED is} \\
\text{sorts } & \text{Boolean Nat} . \\
\text{ops } & 0 \ 1 : \rightarrow \text{Boolean [ctor]} . \quad \text{ops } \# \quad \text{and } \# \text{'true'}. \\
\text{op } & 0 : \rightarrow \text{Nat [ctor]} . \quad \text{op } s : \text{Nat} \rightarrow \text{Nat [ctor]} . \\
\text{op } & _+ : \text{Boolean Boolean} \rightarrow \text{Boolean} . \quad \text{op } _+ : \text{Nat Nat} \rightarrow \text{Nat} .
\end{align*}
\]

is prerregular. If we know the sorts of the constants in a term, each term has a unique least sort in sensible signatures. For example, \( 0 + 0 \) has sorts Nat and Boolean, but the term \( (0).\text{Nat} + (0).\text{Nat} \) (we have told the system that we consider the Nat-constant 0) has unique least sort Nat. ✗
An order-sorted equational specification consists of an order-sorted signature and a set of unconditional and conditional equations, where the sorts of the terms \( t \) and \( t' \) in an equation \( t = t' \) must be in the same connected component\(^6\) of the partially ordered set \((S, \leq)\) of sorts, and analogously for conditional equations \([?]\). (Intuitively, two sorts \( s \) and \( s' \) are in the same connected component of \((S, \leq)\) if there is a "path" from \( s \) to \( s' \) when you draw the partially ordered set \((S, \leq)\) as an undirected graph.)

### 2.5.1 Examples of Order-Sorted Equational Specifications

This section shows some typical uses of order-sorted specifications.

#### 2.5.1.1 Partiality

We have defined the data type of natural numbers with addition, subtraction, and multiplication. However, we omitted division. The reason is that division is a partial function on the natural numbers, since \( n/0 \) is undefined for any \( n \). The point is that we can define an appropriate (sub)sort \( \text{NzNat} \), for the nonzero natural numbers, of \( \text{Nat} \), so that division is well-defined on (the domain defined by) the subsort.

```plaintext
sorts NzNat Nat . subsort NzNat < Nat .
```

The constructors must be declared so that the constructor ground terms of sort \( \text{NzNat} \) are exactly all the nonzero positive numbers:

```plaintext
```

The division operator can then be declared to have only nonzero denominators:

```plaintext
op _/_ : Nat NzNat -> Nat .
```

A subsort \( \text{NeList} \) of \( \text{List} \) for non-empty lists can be defined in the same way, so that \( \text{first} \), \( \text{last} \), \( \text{rest} \), and \( \text{max} \) become total functions on that subdomain:

```plaintext
sorts List NeList . subsort NeList < List .
```

The first three of the above functions are then defined as follows:

```plaintext
ops first last : NeList -> Nat .
op rest : NeList -> List .
var N : Nat . var L : List . var NEL : NeList .
eq first(nil N) = N . eq first(NEL N) = first(NEL) .
eq last(L N) = N . eq rest(nil N) = nil . eq rest(NEL N) = rest(NEL) N .
```

\(^6\) A connected component of \((S, \leq)\) is an equivalence class in the transitive and symmetric closure of \((S, \leq)\).
Likewise, as mentioned, in the context of integers, a number of functions, such as the factorial and the fibonacci functions, are partial functions that are only defined on the natural numbers, which leads us to the next topic.

2.5.1.2 Constructors for the Integers

Without subsorts it is fairly tricky to represent the integers so that each integer corresponds to exactly one constructor ground term, and vice versa. However, it is easy to have this desired one-to-one correspondence by using subsorts as follows:

```plaintext
sorts Zero NzNat NzNeg Nat Neg Int .
subsorts Zero < Nat Neg < Int .
subsort NzNat < Nat .
subsort NzNeg < Neg .
```

The sort Zero is the intended sort for 0, NzNat and NzNeg denote the nonzero natural and negative numbers, respectively, Nat and Neg all natural, respectively negative numbers, including 0, and Int denotes all integers. The sort NzInt denoting nonzero integers is added to deal with division:

```plaintext
sort NzInt . subsorts NzNat NzNeg < NzInt < Int .
```

We use the following well-known constructors for the natural numbers:

```plaintext
```

There are two intuitive ways of constructing the negative numbers. One is to negate a natural number to get a negative number (like \(- s(s(0))\) to represent \(-2\)):

```plaintext
```

The other option is symmetric to the construction of the natural numbers, namely, to use a “predecessor” function \(p\), where \(p(x)\) is the predecessor of \(x\) (that is, \(x - 1\)), just as \(s(n)\) is the successor of \(n\). Such a constructor should be declared

```plaintext
op p : Neg -> NzNeg [ctor] .
```

In this case, \(-2\) is represented by \(p(p(0))\). In either case, it should be possible to see that each constructor term represents exactly one integer, and vice versa.

Addition and subtraction on the integers (using the constructor \(-_\) for the negative numbers) can then be defined as follows.

```plaintext
ops _+_ _-_ : Int Int -> Int [prec 33] .
```

First, addition on the natural numbers is defined in the usual way:

```plaintext
eq 0 + I = I .
\mathrm{eq} s(M) + N = s(M + N) .
```

Subtraction on the naturals is defined as follows:
Addition on all integers can then be defined:\footnote{The extra parentheses in the following equations are not needed, due to the precedence on the operators. They are just added for readability.}

\begin{align*}
eq & - \text{NZN} + (- \text{NZN'}) = -(\text{NZN} + \text{NZN'}) . \\
eq & M + (- \text{NZN}) = M - \text{NZN} . \\
eq & (- \text{NZN}) + N = N - \text{NZN} .
\end{align*}

Finally, we define subtraction on all integers:

\begin{align*}
eq & 0 - (- \text{NZN}) = \text{NZN} . \\
eq & (- \text{NZN}) - (- \text{NZN'}) = \text{NZN'} - \text{NZN} . \\
eq & M - (- \text{NZN}) = M + \text{NZN} . \\
eq & (- \text{NZN}) - N = -(\text{NZN} + N) .
\end{align*}

\subsection*{2.5.1.3 Elements in a List}

Our lists have the form \texttt{nil \_ \ldots \_ \_}. It is possible to “get rid of” \texttt{nil} from this list by saying that a natural number is also a list (of length one):

\begin{align*}
\text{subsort Nat < List .} \\
\text{op \_ \_ : List List -> List [ctor] .}
\end{align*}

Hence, \texttt{nil} is a list, \texttt{s(s(0))} is a list, \texttt{(0 s(s(0)))} \texttt{(s(0) s(s(s(0))))} is a list, and \texttt{nil s(s(0))} is a list. The advantage is that lists are more elegantly presented as a sequence of numbers. The disadvantage is that the one-to-one correspondence between constructor ground terms and “lists” is gone, since \texttt{s(s(0))} and \texttt{nil s(s(0))} represent the same list, as do \texttt{(s(s(0)) 0) s(0) and s(s(0)) (0 s(0))}. Section 2.8 shows some Maude features which “remove” these problems.

\subsection*{2.5.1.4 “Undefined” Values}

An additional “default” (or “error” or “uninitialized”) value must sometimes be added to a sort. The following supersort \texttt{DefNat} adds a such constant \texttt{noNat} to the natural numbers:

\begin{align*}
\text{sort DefNat .} \\
\text{subsort Nat < DefNat .} \\
\text{op noNat : -> DefNat [ctor] .}
\end{align*}

\section*{Exercise 14}

Consider the following signature:

\begin{align*}
\text{sorts s1 s2 s3 s4 .} \\
\text{subsorts s2 s3 < s4 .} \\
\text{subsort s2 < s1 .} \\
\text{op a : -> s3 .} \\
\text{op b : -> s2 .} \\
\text{op g : s3 s2 -> s1 .} \\
\text{op g : s2 s1 -> s2 .} \\
\text{op g : s1 s1 -> s4 .}
\end{align*}

\begin{enumerate}
\item Is the signature preregular?
\end{enumerate}
2. Can you list at least 4 ground terms of sort $s4$? Of sort $s1$?
3. What is the least sort of the terms $a$ and $g(b, g(b, g(a, b)))$?
4. Explain why we cannot add a declaration $\text{op } g : s4 \rightarrow s4$ . and still have a preregular signature.

**Exercise 15** Define a Maude module $\text{NAT2}$ with sorts $\text{NzNat}$ and $\text{Nat}$ and with all the usual functions for natural numbers, including the integer division function $\div$.

**Exercise 16** Define the other functions in $\text{NAT1}$ (see Exercise 9) on integers.

**Exercise 17** Define the integers and the above functions when the predecessor function is used as the constructor for the nonzero negative numbers.

**Exercise 18** An attempt to define the comparison function $\leq$ could be

$$
eq \text{NEG} \leftarrow N \Rightarrow \text{true} .$$
$$
eq N \leftarrow \text{NZNEG} \Rightarrow \text{false} .$$
$$
eq (- \text{NZN}) \leftarrow (- \text{NZN'}) \Rightarrow \text{NZN'} \leftarrow \text{NZN} .$$
$$
eq s(M) \leftarrow s(N) \Rightarrow M \leftarrow N .$$

Explain why these equations do not define $\leq$ for all pairs of integers. Then add the “missing” equation(s).

### 2.6 Membership Equational Logic Specifications

Order-sorted specifications have some limitations:

1. An important subset of binary trees are binary search trees, with partial functions, such as $\text{insertSorted}$, which inserts an element in the right place in a search tree, that only makes sense for search trees. A subsort $\text{SortedList}$ of sorted lists, with functions $\text{insertSorted}$ and $\text{merge}$, could also be useful. However, such “semantic” subsorts cannot be defined as order-sorted specifications.

2. The subsort $\text{NzInt}$ for non-zero integers was defined to avoid problems with division by 0, so that $s(s(0)) \div 0$ is not a (well-formed) term. A side effect is that an expression like $s(s(0)) \div (s(0) - 0)$ (i.e., $2/(1-0)$), which denotes a well-defined mathematical expression, is not a term, since the least sort of $s(0)$ - 0 is $\text{Int}$. Likewise, we use a subsort $\text{NeList}$ for non-empty lists to avoid problems with $\text{first}$ and $\text{last}$ of an empty list. However, this means that a sensible expression like $\text{first}(\text{rest}(\text{nil } s(0) \ s(s(0))))$ is not a well-formed term, since $\text{rest}(\text{nil } s(0) \ s(s(0)))$ is not a term of $\text{NeList}$.

**Membership equational logic** [?, ?] is an elegant extension of order-sorted specifications that solves these problems by allowing us to define membership axioms

$$\text{mb } t : s . \quad \text{and} \quad \text{cmb } t : s \text{ if cond} .$$

---

8 These are binary trees where, for each subtree, the root element of the (sub)tree is greater than or equal to all elements in its left subtree and is less than or equal to all elements in its right subtree.
stating that the term \( t \) (of some supersort of the sort \( s \)) is also a term of sort \( s \) (provided that the condition \( \text{cond} \), consisting of a conjunction of memberships \( t' : s' \) and equalities \( u = u' \), holds in the case of conditional membership axioms). The subsort \( \text{SortedList} \) of \( \text{List} \) can then be defined as follows:

```maude
fmod SORTED-LIST-NAT1 is protecting LIST-NAT1 .
  sort SortedList .  subsort SortedList < List .
  var L : List .
  cmb L : SortedList if isSorted(L) = true .
endfm
```

A term \( \text{nil} \ 0 \ s(0) \) is also a term of sort \( \text{SortedList} \), whereas \( \text{nil} \ s(0) \ 0 \) is not.

Considering our second problem, membership equational logic allows expressions like \( s(s(0)) / (s(0) - 0) \) and gives them “the benefit of doubt.” Such an expression does not have a sort like \( \text{Int} \) but an “error sort” \([\text{Int}]\). The term \( s(s(0)) / (s(0) - 0) \) is evaluated by computing wherever possible, and is eventually evaluated to \( s(s(0)) / s(0) \) using the equations for \(-\). This term is a well-formed term of sort \( \text{Int} \) and the computation can proceed to give the expected result:

```
Maude> red s(s(0)) / (s(0)) - 0) .
result NzNat: s(s(0))
```

The term \( s(0) / (s(0) - s(0)) \) is also given the benefit of doubt and is reduced to \( s(0) / 0 \), which does not have a sort and cannot be further reduced, and is therefore a term of “error sort” \([\text{Int}]\):

```
Maude> red s(0) / (s(0) - s(0)) .
result [Int]: s(0) / 0
```

The theoretical reason for this possibility of giving a term “the benefit of doubt” is that, in membership equational logic, each connected component of the partially ordered set \((S, \leq)\) of sorts has a kind. We write \([s]\) for the kind of the connected component of sort \( s \). Terms which do not have sorts and only have a kind can be seen as “error terms.” Maude automatically adds a declaration

```maude
op f : [\[s_1\]] ... [\[s_n\]] -> [s] .
```

for any declaration

```maude
op f : s_1 ... s_n -> s .
```
in the specification. In particular, this means that our Maude specification of the integers (implicitly) also contains a declaration

```maude
op _/_ : [\text{Int}] [\text{NzInt}] -> [\text{Int}] .
```

Since \( s(0) - s(0) \) is a term of sort \( \text{Int} \), and therefore also of kind \([\text{Int}]\), the term \( s(0) / (s(0) - s(0)) \) has kind \( \text{Int} \) due to the implicit declaration above and the fact that \([\text{NzInt}] = [\text{Int}]\). Since \( s(0) / (s(0) - s(0)) \) is a “well-kinded” term, it can be further reduced to the term \( s(0) / 0 \) of kind \([\text{Int}]\). This term cannot be reduced any further, and, although well-kinded, has no sort.
Exercise 19 Define a subsort $\text{BinSearchTree}$ of $\text{BinTree}$ for binary search trees and define a function $\text{insertSorted} : \text{BinSearchTree} \ Nat \rightarrow \text{BinSearchTree}$ which inserts an element in the right place in the tree.

2.7 Built-in Data Types

The representation of the natural numbers and integers we have seen so far is not very convenient for computing with large numbers. Maude therefore provides built-in versions of the natural numbers, the integers, the rational numbers, and the IEEE-754 double precision floating-point numbers, in addition to strings and Boolean values. These built-in modules provide the standard notation for numbers and strings, such as 2015, -273, 22/7, and "Maude", and the expected operations on numbers and strings efficiently implemented in C++. In contrast to many programming languages, Maude provides an efficient implementation of the unbounded natural numbers, integers, and rational numbers, instead of only 32-bits or 64-bits numbers.

These built-in data types are defined in the file `prelude.maude` which is read when you start Maude. You can change this file if you feel like redefining the built-in modules or giving commands which should be executed when Maude starts. Only the built-in Booleans are included automatically into any user module; to import Maude’s natural numbers, you need to explicitly import the module `NAT` into your module in the usual way. To automatically include `NAT` into all your modules, just add the Maude command `set include NAT on` to the file `prelude.maude`. This section briefly introduces some of Maude’s built-in modules; see the file `prelude.maude` for more details about these and other built-in modules.

2.7.1 The Booleans

The following module `BOOL` defines the Boolean values and some useful functions:

```maude
fmod TRUTH-VALUE is
  sort Bool .
  op true : -> Bool [ctor special {id-hook SystemTrue}] .
  op false : -> Bool [ctor special {id-hook SystemFalse}] .
endfm

fmod BOOL-OPS is
  protecting TRUTH-VALUE .
  op _or_ : Bool Bool -> Bool [assoc comm prec 59] .
  op _xor_ : Bool Bool -> Bool [assoc comm prec 57] .
  op _implies_ : Bool Bool -> Bool [gather (e E) prec 61] .
endfm
```

9 Parts of the specifications are omitted and replaced by ‘...’.
vars A B C : Bool .
eq true and A = A .
eq false and A = false .
...
endfm

fmod TRUTH is
  protecting TRUTH-VALUE .
  op if_then_else_fi : Bool Universal Universal -> Universal
      [poly (2 3 0) special {...}] .
  op _==_ : Universal Universal -> Bool
      [prec 51 poly (1 2) special {...}] .
  op _=/=_ : Universal Universal -> Bool
      [prec 51 poly (1 2) special {...}] .
endfm

fmod BOOL is
  protecting BOOL-OPS .
  protecting TRUTH .
endfm

The special attribute associated to some operators says that these are built-in operators/functions implemented in C++. The attributes assoc and comm mean that the function is, respectively, associative and commutative; these attributes are explained in Section 2.8. We ignore the gather attribute (see [?] for an explanation of this parsing issue). The poly attribute states that the corresponding arguments (of sort Universal) may have any sort. The operator if_then_else_fi behaves as expected, x == y equals true if and only x and y are equal, and conversely for the inequality operator.

A condition b = true in an equation can be written just b:

c\_eq M monus N = 0 if M <= N .

Finally, t :: s is a term of sort Bool which is true if and only if the term t has sort s.

### 2.7.2 The Natural Numbers

Maude provides the following module for arbitrarily large natural numbers, whose implementation uses the GNU GMP library [?] \(^{10}\).

fmod NAT is
  protecting BOOL .
  sorts Zero NzNat Nat .
  subsort Zero NzNat < Nat .
  op 0 : -> Zero [ctor] .
  op s_ : Nat -> NzNat [ctor iter special {...}] .
  op _+_ : NzNat Nat -> NzNat [assoc comm 33 special {...}] .
  op sd : Nat Nat -> Nat [comm special {...}] .

\(^{10}\) Multiple declarations of the same non-constructor function are usually not needed, since equations will reduce a term to a constructor term of the right sort. However, in built-in modules, operators such as + have multiple declarations, since it is a built-in function not defined by equations.
The constructors for \texttt{Nat} are \texttt{0} and \texttt{s\_}, so the natural numbers are represented by the terms \texttt{0}, \texttt{s\_0}, \texttt{ss\_0}, \ldots. For convenience, we can also write \texttt{0}, \texttt{1}, \texttt{2}, \ldots:

\begin{verbatim}
Maude> red s s 0 + s s s 0 .
result NzNat: 5
Maude> red 1234567 * 89 .
result NzNat: 109876463
\end{verbatim}

There is no subtraction function on the natural numbers (why?). Instead, the function \texttt{sd} denotes the (symmetric) difference between two numbers.

\textbf{Example 8} The factorial function can be defined by induction on the constructors:

\begin{verbatim}
mod FACTORIAL is protecting NAT .
  op _! : Nat \rightarrow Nat .
  var N : Nat .
  eq 0 ! = 1 .
  eq (s N) ! = s N * (N !) .
endfm
\end{verbatim}

or using the "standard" natural numbers and replacing the above equations with

\begin{verbatim}
eq N ! = if N == 0 then 1 else N * (sd(N, 1) !) fi .
\end{verbatim}

\textbf{\textbullet} 

The function \texttt{quo} defines division, \texttt{rem} the \textit{remainder} function, \texttt{\_\_\_\_\_} exponentiation \texttt{(m \_\_\_\_\_ n = m\_\_\_\_n)}, \texttt{gcd} denotes the greatest common divisor, \texttt{lcm} the least common multiple, and \texttt{<}, \texttt{<=}, \texttt{>, and >} are the usual comparison operators. The module \texttt{NAT} also has bit manipulating functions such as bitwise and (\texttt{&}), bitwise or (\texttt{|}), bitwise xor (\texttt{xor}), right shift (\texttt{>>}), and left shift (\texttt{<<}).

Finally, subsort overloaded operators must have the same attributes, except for \texttt{ctor}. \texttt{Maude} provides the attribute \texttt{ditto} to avoid having to repeat the attributes. \texttt{ditto} stands for \textit{all attributes except ctor} in previous declarations of the same (subsort overloaded) function symbol.
2.7.3 The Integers

The integers are constructed from the natural numbers using the constructor \(-\)\-, so that negative numbers can be written as \(-\ 0\), \(-\ 2009\), \ldots, and also as \(-1\), \(-2009\), \ldots. The built-in efficient implementation of (unbounded) integers are given in the following module, where functions in \text{NAT} with a straight-forward extension into the integers (such as +, *, <, \ldots) are omitted:

\begin{verbatim}
|fmod INT is protecting NAT .|
|sorts NzInt Int . subsorts NzNat < NzInt Nat < Int .|
|op \_\_ : NzNat -> NzInt [ctor special {\ldots}] .|
|op \_\_ : NzInt -> NzInt [ditto] .|
|op \_\_ : Int Int -> Int [assoc comm prec 33 special {\ldots}] .|
|op abs : Int -> Nat [\ldots] .|
...
endfm
\end{verbatim}

(abs gives the absolute value of a number.) The function \_\_\_ is a constructor only on NzNat, and is a non-constructor on NzInt and Int. The term \(-\ 2009\) is therefore a constructor term of sort Int, while \(-\ 0\) and \(-\ 1234\) are not.

2.7.4 The Rational Numbers

The rational numbers are defined in the module \text{RAT} which defines the sorts NzRat (nonzero rational numbers), PosRat (non-zero positive rational numbers), and Rat (all rational numbers) as follows:

\begin{verbatim}
|fmod RAT is protecting INT .|
|sorts PosRat NzRat Rat . subsorts NzInt < NzRat Int < Rat .|
|subsorts NzNat < PosRat < NzRat .|
|op \_\_/ : NzInt NzNat -> NzRat [ctor prec 31 ... special {\ldots}] .|
|op \_\_/ : NzNat NzNat -> PosRat [ctor ditto] .|
|op \_\_/ : PosRat PosRat -> PosRat [ditto] .|
|op \_\_/ : NzRat NzRat -> NzRat [ditto] .|
|op \_\_/ : Rat NzRat -> Rat [ditto] .|
...
|ops trunc floor : PosRat -> Nat .|
|ops trunc floor ceiling : Rat -> Int .|
|op ceiling : PosRat -> NzNat .|
|op frac : Rat -> Rat .|
|var I : NzInt . var N M : NzNat . var K : Int .|
|eq trunc(K) = K . eq trunc(I / N) = I quo N .|
|eq floor(K) = K . eq floor(N / M) = N quo M .|
|eq floor(- N / M) = - ceiling(N / M) .|
|eq ceiling(K) = K . eq ceiling(N / M) = (N + M - 1) quo M .|
\end{verbatim}
2.7 Built-in Data Types

\[
\begin{align*}
eq \text{ceiling}\left(- \frac{N}{M}\right) &= - \text{floor}\left(\frac{N}{M}\right) . \\
eq \text{frac}(k) &= 0 . \\
eq \text{frac}\left(\frac{I}{N}\right) &= \left(\frac{I \ \text{rem} \ N}{N}\right) .
\end{align*}
\]
endfm

The module RAT also defines well-known functions such as \(-\) (unary minus), +, -, *, /, <, <=, >, and >= on the rationals.

2.7.5 The Floating-Point Numbers

The built-in module FLOAT implements 64-bits IEEE-754 double precision floating-point numbers all the expected functions such as \(\sqrt{}\) (for square root), the trigonometric functions, the logarithm function, and so on.

fmod FLOAT is protecting BOOL .
sorts FiniteFloat Float . subsort FiniteFloat < Float .
op <Floats> : -> FiniteFloat [special {id-hook FloatSymbol}] .
op <Floats> : -> Float [ditto] .
...
op sqrt : Float \to Float [...] .
op log : Float \to Float [...] .
op sin : Float \to Float [...] . op cos : Float \to Float [...] .
op asin : Float \to Float [...] . op acos : Float \to Float [...] .
...
endfm

The syntax \(<\text{Floats}>\) means that the constructors are built-in as a set of constants such as 1.0, -9.87654321, and -1.23e+14 (for \(1.23 \cdot 10^{14}\)). The sort Float also contains two constants Infinity and -Infinity that denote out of range values:

Maude> red 3.45e+223 \times 2.99e+210 .
result Float: Infinity

2.7.6 Strings

The built-in Maude module STRING defines the sort String of strings of the form "this is a string". Strings of length 1 are constants of a subsort Char.

fmod STRING is protecting NAT .
sorts String Char FindResult .
subsort Char < String . subsort Nat < FindResult .
op <Strings> : -> Char [special {id-hook StringSymbol}] .
op <Strings> : -> String [ditto] .
op notFound : -> FindResult [ctor] .
op ascii : Char \to Nat [...] . op char : Nat \to Char [...] .
op _+_ : String String \to String [...] .
op length : String \to Nat [...] .
42 2 Equational Specification in Maude

op substr : String Nat Nat -> String [...] .
op find : String String Nat -> FindResult [...] .
op rfind : String String Nat -> FindResult [...] .
op _<=_ : String String -> Bool [...] .

endfm

The function \( \text{ascii} \) gives the ASCII value of a character, \( \text{char} \) does the opposite, \( + \) denotes string concatenation, and \( \text{length} \) returns the length of a string. \( \text{substr}(s,p,l) \) returns the substring of \( s \) which starts at character \( p+1 \) and is \( l \) characters long. \( \text{find}(s_1,s_2,p) \) finds the starting position (minus 1) of the substring \( s_2 \) in \( s_1 \), starting at character number \( p+1 \) in \( s_1 \) (and returns \( \text{notFound} \) if \( s_2 \) is not such a substring of \( s_1 \). \( \text{rfind} \) does the same, but starts looking “from the right.”

The comparison operators \( <, <=, >, \text{and} \geq \) compare strings lexicographically.

The module \( \text{CONVERSION} \) defines functions for converting between numbers and strings, and between rational numbers and floating-point numbers. For example, \( \text{string}(r,n) \) takes a rational number \( r \) and a base \( n \) (between 2 and 36), and displays the number as a String in the given base. That is, \( \text{string}(123,10) \) equals \( "123" \) and \( \text{string}(5,2) \) equals \( "101" \). The function \( \text{rat} \) does the opposite.

---

**Exercise 20** Use the built-in module \( \text{NAT} \) and define a function

\[\text{op isPrime : NzNat -> Bool .}\]

which returns \( \text{true} \) if and only if its argument is a prime number (that is, a number which is not divisible by any number except 1 and itself). Test your specification on 14091 (not a prime), 2 (prime), 31 (prime), and 135727 (?).

**Exercise 21** Explain why the constructor for rational numbers is not declared

\[\text{op \_/_ : Int NzNat -> Rat [ctor ...] .}\]

or

\[\text{op \_/_ : NzInt NzInt -> NzRat [ctor ...] .}\]

**Exercise 22** Explain what the functions \( \text{trunc}, \text{floor}, \text{ceiling}, \text{and} \text{frac} \) in the module \( \text{RAT} \) are supposed to compute.

**Exercise 23** American sports scores have the form “49ers 39 Giants 38” while Europeans prefer the notation “49ers - Giants 39-38”. Define a function \( \text{europify} : \text{String} \to \text{String} \) which transforms a score from American format to European format. You may assume that there are no blanks in the name of a team.

**Exercise 24** Define a function \( \text{op binary : Nat -> Nat} \) which gives the “binary” value of a natural number, so that e.g. \( \text{binary}(7) \) equals the number 111.
2.8  Associativity and Commutativity: Lists and Multisets

This section defines some equational attributes, such as associativity and commutativity, that enable us to define lists and sets in a nice way, and that can make the computation of certain functions more efficient.

2.8.1 Commutativity, Associativity, and Identity

The (multi)sets \( \{a, b\} \) and \( \{b, a\} \) are the same, and therefore their representations should be equivalent. More generally, it is sometimes needed or useful to define a function \( f \) (such as, e.g., set union) to be commutative:

\[
 f(x, y) = f(y, x).
\]

However, such an equation leads to infinite loops \( f(x, y) \rightarrow f(y, x) \rightarrow f(x, y) \rightarrow \ldots \). The Maude solution to having both commutativity and termination is to declare that \( f \) is commutative, so that Maude always "keeps in mind" that \( f \) is commutative. We can declare that a function \( f \) is commutative by giving it an attribute \( \text{comm} \):

```latex
fmod COMM1 is
  sort s .
  op f : s s -> s [comm] .
  ops a b c : -> s .
  eq f(a,b) = b .
endfm
```

When a function is declared to be commutative, computations are no longer performed on terms but on \( C \)-equivalence classes of terms, \( \mathcal{C} \), where \( \mathcal{C} \) is the commutativity axiom \( f(x, y) = f(y, x) \). In \( \text{COMM1} \), the function \( f \) is declared to be commutative, and one therefore works on the set \( \mathcal{P}_C = \mathcal{P}_C / \mathcal{C} \) of equivalence classes of terms modulo commutativity with \( [t]_C = \{u | t \sim_C u\} \), where \( C \) is the specification \( \{ f(x, y) = f(y, x) \} \), and \( t \sim_C u \) holds if and only if \( t \) and \( u \) are equal up to commutativity of \( f \) that is, there are zero or more simplification steps \( t \sim_C \cdots \sim_C u \) from \( t \) to \( u \) using the above commutativity equation. For example,

\[
\begin{align*}
[f(a, f(b, c))],
[f(a, f(b, c))_C] &= f(a, f(c, b)),
[f(f(b, c), a)],
[f(f(c, b), a)]
\end{align*}
\]

**Important:** To avoid too many symbols, I will most often write \( t \) for \( [t]_C \).

The term \( f(b, a) \) can be reduced to \( b \) in \( \text{COMM1} \) since by \( f(b, a) \) we mean \( [f(b, a)]_C \) and

\[
[f(b, a)]_C = [f(a, b)]_C = [b]_C.
\]
The point is that one-to-one constructor bases, termination, confluence, definedness, etc., are now defined on $C$-equivalence classes of terms instead of on terms.

**Example 9** A function `minimum` which returns the smallest of two integers can be elegantly defined by a single equation:

```maude
fmmod MIN1 is protecting INT .
  op minimum : Int Int -> Int [comm] .
  vars I J : Int .
  ceq minimum(I, J) = I if I <= J .
endfm

A binary function $f$ is associative if

$$f(f(x,y),z) = f(x,f(y,z))$$

holds for all $x,y,z$. Addition on the integers is associative since $(x+y)+z = x+(y+z)$. We can declare a function to be associative using the `assoc` attribute:

```maude
op f : s s -> s [assoc] .
```

A term $t$ is considered to be equivalent to a term $u$ if they are equivalent under the associativity axiom; that is, if you can perform zero or simplification steps to go from $t$ to $u$ using the associativity axiom in both directions. For example, the term $t(z(a,b),z(c,d))$ is considered the same as $t(a,t(b,z(c,d)))$ and $t(z(t(a,b),c),d)$ and $t(a,z(t(b,c),d))$ if $t$ is declared associative. Since the parentheses can be rearranged arbitrarily for associative operators, they are no longer needed for $t$ and we can write $t(a,b,c,d)$ instead of the above terms. Likewise, if an infix symbol $+$ is declared associative, we can write $1+2+3$.

Although the associativity axiom $f(f(x,y),z) = f(x,f(y,z))$ does not cause non-termination, there are some good reasons to treat associativity in this way:

- Specifications of important data types such as lists and sets/multisets are much more elegant as we may omit parentheses and define functions on such types much more naturally.
- Although associativity by itself does not lead to nontermination, it leads to non-termination if the function already is declared commutative:

```maude
op f : s s -> s [comm] .
  vars X Y Z : s .
  eq f(f(X,Y),Z) = f(X,f(Y,Z)) .

⇒ Associativity
```

The specification is nonterminating modulo commutativity since there is an infinite derivation

$$[f(f(a,b),c)]c \leadsto [f(a,f(b,c))]c = [f(f(b,c),a)]c \leadsto [f(b,f(c,a))]c$$
$$= [f(f(c,a),b)]c \leadsto [f(c,f(a,b))]c = [f(f(a,b),c)]c \leadsto \cdots$$

Therefore, if $t$ is declared commutative, associativity of $t$ must be taken care of by adding `assoc` as an attribute:
2.8 Associativity and Commutativity: Lists and Multisets

A binary function can also be defined to have an identity element \( t \):

\[
op f : s s \rightarrow s \ [\text{ctor id: } t]\.
\]

which means that computations are performed modulo the equations

\[
f(t,x) = x \text{ and } f(x,t) = x.
\]

That is, any term \( u \) of sort \( s \) will be considered to be identical to \( f(u,t) \) and \( f(t,u) \).

For example, in

\[
\text{sort } s . \quad \text{ops } a \ b \ e : \rightarrow s \ [\text{ctor}] . \quad \text{op } f : s s \rightarrow s \ [\text{ctor id: } e] . \quad \text{vars } X \ Y : s . \quad \text{eq } f(X,Y) = a.
\]

the term \( b \) reduces to \( a \), since \( b \) is the same as \( f(b,e) \). However, be careful with termination; even the seemingly terminating equation above is nonterminating, since we have an infinite computation \( [a] = [f(a,e)] = [f(a,e)] = \cdots \).

2.8.2 Associativity and Identity: Lists

Section 2.4.3.1 defines lists using a constructor \( _\_ \) : List Nat \( \rightarrow \) List and a constant \( \text{nil} \). All lists have the form \( \ldots(((\text{nil} n_1) n_2)\ldots)n_k \) (even though the parentheses may be omitted since there is only way to parse a term). However, it is more natural to view lists as “flat” structures; this suggests the following representation of lists, where an integer is also a list (of one element):

\[
\text{sort List . \quad subsort Int < List . \quad op } \text{nil} : \rightarrow \text{List [ctor]} . \quad \text{op } _\_ : \text{List List } \rightarrow \text{List [ctor assoc]} .
\]

Both \( 4 \) and \( 7 \) are terms of sort \( \text{List} \), since \( \text{Int} \) is a subsort of \( \text{List} \). These two lists can be concatenated using the concatenation operator \( _\_ \) so that \( 4 \ 7 \) is also a term of sort \( \text{List} \). This list can be concatenated with the list \( 11 \), which gives a term \( 4 \ 7 \ 11 \), which can be concatenated with the list \( 99 \) to get the list \( 4 \ 7 \ 11 \ 99 \). Or, the two lists \( 4 \ 7 \) and \( 11 \ 99 \) can be concatenated into \( 4 \ 7 \ 11 \ 99 \). Since \( _\_ \) is declared to be associative, these two lists are the same list, and we can ignore parentheses: \( 4 \ 7 \ 11 \ 99 \).

Unfortunately, since \( \text{nil} \) is a term of sort \( \text{List} \), also \( \text{nil} \ 4 \) and \( 7 \ \text{nil} \) are Lists, and so is their concatenation \( \text{nil} \ 4 \ 7 \ \text{nil} \). The point is that we can easily “eliminate” these \( \text{nils} \) by declaring \( _\_ \) to have \( \text{nil} \) as its identity element:

\[
\text{op } _\_ : \text{List List } \rightarrow \text{List [ctor assoc id: nil]} .
\]
nil 4 and 4 are now exactly the same list (i.e., \([\text{nil 4}_A] = [4]_A\)), and so are therefore nil 4 7 nil and 4 7. This gives the desired one-to-one correspondence between (equivalence classes of) constructor ground terms modulo associativity and identity of the list concatenation constructor and the set of all lists of integers.

A list is now either the empty list nil or has the form \(i \, l\), for \(i\) an integer and \(l\) a list (since the one-element list \(i\) is identical to \(i \, \text{nil}\)) or, equivalently, the form \(l \, i\). This is reflected in the definitions below, which are much simpler than the corresponding ones in Section 2.4.3.1:

\[
\text{fmod LIST-INT is protecting INT.}
\]
\[
\text{sorts List NeList. subsorts Int < NeList < List.}
\]
\[
\text{op nil : -> List [ctor].}
\]
\[
\text{op __ : List List -> List [assoc id: nil ctor].}
\]
\[
\text{op ___ : NeList NeList -> NeList [assoc id: nil ctor].}
\]
\[
\text{op length : List -> Nat.}
\]
\[
\text{ops first last : NeList -> Int.}
\]
\[
\text{op empty? : List -> Bool.}
\]
\[
\text{op rest : NeList -> List.}
\]
\[
\text{op reverse : List -> List.}
\]
\[
\text{op _occursIn_ : Int List -> Bool.}
\]
\[
\text{op max : NeList -> Int.}
\]
\[
\text{op isSorted : List -> Bool.}
\]
\[
\vars I J : Int. \quad \text{var L : List.}
\]
\[
\text{eq length(nil) = 0.}
\]
\[
\text{eq length(I L) = 1 + length(L).}
\]
\[
\text{eq first(I L) = I.}
\]
\[
\text{eq last(L I) = I.}
\]
\[
\text{eq I occursIn nil = false.}
\]
\[
\text{eq I occursIn J L = (I == J) or (I occursIn L).}
\]
\[
\[...\]
\]
\[
\text{endfm}
\]

### 2.8.3 Associativity, Commutativity, and Identity: Multisets and Sets

A multiset over a set \(S\) is a “set” of \(S\)-elements where the number of occurrences of each element matters: while the sets \(\{a, b\}\) and \(\{a, a, b\}\) are the same, the multisets \(\{a, b\}\) and \(\{a, a, b\}\) are different. (Formally, a multiset \(m\) over \(S\) is a function \(m : S \to \mathbb{N}\) where \(m(s)\) denotes the multiplicity (the number of occurrences) of \(s\) in \(m\). A finite multiset is a multiset \(m\) whose support \(\{s \mid s \in S \land m(s) > 0\}\) is a finite set.)

To compare two multisets over totally ordered sets like the integers, just remove (equally many of) the common elements in the multisets; the one with the largest remaining element is the largest multiset (a non-empty multiset is greater than the empty multiset). For example, \(\{2, 2, 1\}\) is greater than \(\{1, 1, 0, 1, 2\}\), and \(\{28099, 3, 8\}\) is greater than \(\{28099, 7, 6, 5, 7, 5, 5, 6, 0, 1\}\).

A multiset can be seen as a list where the order of the elements does not matter. Finite multisets can therefore be modeled by “lists” where the multiset union operator \(\_\_\_\) is also commutative:

\[
\text{fmod MSET-INT is protecting INT.}
\]
\[
\text{sorts Mset NeMset.}
\]
\[
\text{ops first last : NeMset -> Int.}
\]
\[
\text{ops rest : NeMset -> Mset.
}\]
\[
\text{vars I J : Int. \quad \text{var L : List.}
\]
\[
\text{eq length(nil) = 0.}
\]
\[
\text{eq length(I L) = 1 + length(L).}
\]
\[
\text{eq first(I L) = I.}
\]
\[
\text{eq last(L I) = I.}
\]
\[
\text{eq I occursIn nil = false.}
\]
\[
\text{eq I occursIn J L = (I == J) or (I occursIn L).}
\]
\[
\[...\]
\]
\[
\text{endfm}
\]
2.8 Associativity and Commutativity: Lists and Multisets

op none : -> Mset [ctor] . *** Empty multiset

op size : Mset -> Nat . *** No of elements in a multiset
op mult : Int Mset -> Nat . *** Multiplicity of an element
op delete : Int Mset -> Mset . *** Remove ONE occurrence
op _in_ : Int Mset -> Bool . *** Is element in multiset?
op max : NeMset -> Int . *** Largest element
op _>mul_ : Mset Mset -> Bool . *** Multiset comparison

var I : Int . var MS : Mset .
eq delete(I, I MS) = MS .
ceq delete(I, MS) = MS if not I in MS .
eq I in MS = mult(I, MS) > 0 .
...
endfm

A set is essentially a multiset where the multiplicity of elements does not matter. Sets of integers can therefore be defined as multisets of integers with the extra axiom eq I I = I . (for I a variable of sort Int) which removes duplicates. 11

Exercise 25 For each of the (equivalence classes of the) terms f(f(b, a), a) and f(b, b) and f(f(a, b), f(b, a)), compute its normal form in COMM1 “by hand” and using Maude’s red command.

Exercise 26 Complete the module LIST-INT by defining the functions empty?, rest, reverse, max, and isSorted.

Exercise 27 Define the functions

op comesBeforeIn : Int Int List -> Bool .
op _>lex_ : List List -> Bool .

such that comesBeforeIn(i, j, l) is true if and only if there are elements i and j in the list l, and where the first occurrence of i comes before the first occurrence of j in l; and where l_1 >lex l_2 is true if l_1 is lexicographically greater than l_2 (see Exercise 11 for the definition of lexicographic comparison).

Exercise 28 Explain why delete(2015, 1 2 2015 3) returns the multiset 1 2 3 when the function is defined as in the module MSET-INT.

Exercise 29 Define the functions size, mult, max, empty?, and the multiset comparison operator >mul in the module MSET-INT.

Exercise 30 Show that for any multiset m_0 over the natural numbers, there is no infinite sequence

m_0 > m_1 > m_2 > m_3 > ...
Exercise 31 Assume that we have already defined two sorts \texttt{Obj} and \texttt{Msg}. Define a sort \texttt{Mset-ObjMsg} whose elements are multisets of \texttt{Obj} and \texttt{Msg} elements (that is, a multiset may contain both \texttt{Obj} and \texttt{Msg} elements).

Exercise 32 Define a data type of sets of integers with functions \texttt{in} (does the given number belong to the set?), \texttt{delete} (remove an element from a set), \texttt{card} (the cardinality (number of (distinct) elements) of a set), \texttt{setMinus} (set difference), and \texttt{intersect} (the intersection of two sets). Make sure that your specification is confluent. \texttt{delete(1, 0 1 2 1)} should give \texttt{0 2} no matter how the equations are applied. Similarly, the cardinality of the set \texttt{0 1 2 1} is always \texttt{3}.

Exercise 33 1. Define a data type \texttt{StringList} of lists of strings. In this exercise you will use both lists and sets, and you are therefore advised to use a symbol other than \texttt{__} (such as \texttt{:_}) for list concatenation. Only define the functions you will need in this exercise.

2. Define a data type \texttt{Set-StringList} of sets of lists of strings

3. Define a function \texttt{op perm : StringList -> Set-StringList} which takes a list of strings and returns the set of all permutations of this list. (A permutation of a list is a list where the elements are the same but are "rearranged.") For instance, the set of all permutations of the list \texttt{"a" : "b" : "c"} is

\[
\begin{align*}
("a" : "b" : "c") & \quad ("a" : "c" : "b") \\
("b" : "a" : "c") & \quad ("b" : "c" : "a") \\
("c" : "a" : "b") & \quad ("c" : "b" : "a")
\end{align*}
\]

Hint: It might be helpful to use an auxiliary function \texttt{p}, where \texttt{p(L1, L2, L3)} generates all permutations of \texttt{L1 : L2 : L3} which start with \texttt{L1}, and where the next string is taken from the list \texttt{L2}. The \texttt{L1}-elements have already been used.

2.9 Examples: Quicksort and Mergesort

This section shows how the sorting algorithms quicksort and mergesort can be formally specified in Maude; such a formal specification has a number of benefits:

- In contrast to prose and pseudo-code (and even an imperative program), a Maude specification gives a precise, un-ambiguous specification of the algorithm.
- The specification is also at the same time a program, defined in a much simpler and less error-prone way than, e.g., a Java implementation.\footnote{Is it \texttt{i=0} or \texttt{i=1}? \texttt{j=1} or \texttt{j=i+1}? \texttt{i++} or \texttt{++i}? Until \texttt{j<k} or \texttt{j>=k}? A \texttt{-1} or \texttt{+1} missing somewhere?}
- The intuitive Maude formalism may well provide a more understandable description of the protocol than prose and pseudo-code.
- It is possible to reason mathematically about the Maude specification; it is also much easier to reason informally about the correctness of the Maude program than about the Java program, since we can focus on checking the correctness of single equations, instead of having to reason about the entire program.
2.9 Examples: Quicksort and Mergesort

2.9.1 Quicksort

The following description of quicksort is taken from the textbook [?]:

“The quick-sort algorithm sorts a list $L$ using a simple recursive approach. The main idea is to apply the divide-and-conquer technique, whereby we divide $L$ into sublists whose range of elements are disjoint, recurse to sort each sublist, and then combine the sorted sublists by a simple concatenation. In particular, the quick-sort algorithm consists of the following three steps:

1. **Divide**: [... ] select a specific element $N$ from $L$ [where $L$ is $L \setminus N \cup N'$ in the equation below], which is called the **pivot**. For instance, let the pivot $N$ be the last element. Remove all elements from $L$ and put them into three lists:
   - $S$, storing the elements in $L$ less than $N$
   - $E$, storing the elements in $L$ equal to $N$
   - $G$, storing the elements in $L$ greater than $N$

2. **Recurse**: Recursively sort the lists $S$ and $G$.

3. **Conquer**: Put back the elements into $L$ by first inserting the elements of $S$, then those of $E$, and finally those of $G$.

The following Maude definition is a more general specification in that it chooses the pivot $N$ “nondeterministically” instead of being forced to choose “for instance” the last element. Which description is easier to read, the Maude specification or the textbook description?

fmod QUICK-SORT is protecting LIST-INT .
op quicksort : List -> List .
vars L L' : List . vars M N : Int .
eq quicksort(nil) = nil .
eq quicksort(L N L') = quicksort(smallerElements(L L', N))
  equalElements(L N L', N)
  quicksort(greaterElements(L L', N)) .
where smallerElements($l, n$) contains the elements in $l$ that are smaller than $n$:

ops smallerElements greaterElements
  equalElements : List Int -> List .
eq smallerElements(nil, N) = nil .
eq smallerElements(N L, M) = if N < M then
  (N smallerElements(L, M))
else smallerElements(L, M) fi .
eq equalElements(nil, N) = nil .
eq equalElements(N L, M) = if N == M then (N equalElements(L, M))
else equalElements(L, M) fi .
eq greaterElements(nil, N) = nil .
eq greaterElements(N L, M) = if N > M then
  (N greaterElements(L, M))
else greaterElements(L, M) fi .
endfm
2.9.2 Mergesort

The mergesort algorithm for sorting a list $S$ with $n$ elements is given as follows [?]:

1. **Divide**: Is $S$ has at least two elements (nothing needs to be done if $S$ has zero or one element), remove all the elements from $S$ and put them into two sequences $S_1$ and $S_2$, each containing about half the elements of $S$, that is, $S_1$ contains the first $\lceil n/2 \rceil$ elements of $S$, and $S_2$ contains the remaining $\lfloor n/2 \rfloor$ elements.
2. **Recurse**: Recursively sort sequences $S_1$ and $S_2$.
3. **Conquer**: Put back the elements into $S$ by merging the sorted sequences $S_1$ and $S_2$ into a unique sorted sequence.

A Maude specification of mergesort can be given as follows:

```maude
fmod MERGE-SORT is protecting LIST-INT .
  op mergeSort : List -> List .
  eq mergeSort(nil) = nil .
  eq mergeSort(I) = I .
  ceq mergeSort(NEL NEL') = merge(mergeSort(NEL), mergeSort(NEL')) if length(NEL) == length(NEL') or length(NEL) == s length(NEL') .
  eq merge(nil, L) = L .
  ceq merge(I L, J L') = I merge(L, J L') if I <= J .
endfm
```

The raison d’être for mergesort is that its execution time is $O(n \log n)$. However, the above specification may be less efficient, since the splitting of a list into two halves is done by matching. The usefulness of this specification is that it provides a precise description of mergesort that is also a prototype that can be used to quickly test the mergesort algorithm before a more detailed and efficient algorithm is implemented. (Of course, we already know that the mergesort algorithm is supposed to work. But imagine that you have a complex algorithm that you hope solves a difficult problem. It is then very useful to quickly being able to develop and test a prototype/model of your algorithm before implementing it in all its glorious detail.)

**Exercise 34** Define a version of quicksort which, for lists of at least two elements, will look at the first and the last element in the list, and choose as pivot element the number $\frac{\text{first} + \text{last}}{2}$. (It is possible that such a number is not an element in the list, but that doesn’t matter.) Explain also why your specification is terminating.

**Exercise 35** Specify the insertion sort algorithm in Maude. Insertion sort works as when you get some cards and have to sort them: you take the (unsorted) cards one by one, and put them into the right place in your hand, which always remains sorted.
2.10 * Some Other Maude Features

Maude has a number of useful features that will not be mentioned elsewhere in this book; the reader is referred to the Maude book [?] or the Maude manual for details.

2.10.1 Parameterized Modules

Instead of defining a data type, such as lists, from scratch for each kind of list (lists of integers, lists of strings, lists of lists of . . . , and so on), we can define parameterized modules. Assume that we want to define a generic mergesort function that can sort all kinds of lists, as long as we can compare the elements in the lists. That is, any parameter must have a sort for the elements and a total order on those elements; such a “formal parameter” is defined as a theory:

```maude
fth ORDERED-SET is protecting BOOL .
  sort Element .
  op _le_ : Element Element -> Bool .
  vars E E1 E2 E3 : Element .
  --- reflexivity, anti-symmetry, transitivity, and totality:
  eq E le E = true [nonexec] .
  ceq E1 = E2 if (E1 le E2) and (E2 le E1) [nonexec] .
  ceq E1 le E3 = true if (E1 le E2) and (E2 le E3) [nonexec] .
  eq (E1 le E2) or (E2 le E1) = true [nonexec] .
endfth
```

This theory defines an “interface” or “formal parameter” ORDERED-SET that any actual parameter must “satisfy.” That is, the actual parameter must interpret the sort Element and the function symbol _le_ (the comparison operator), so that the four equations for a total order are satisfied.

The parametric mergesort module is then given as follows:

```maude
fmod PARAM-SORT(X :: ORDERED-SET) is protecting INT .
  sorts List NeList .
  subsort X$Element < NeList < List .
  op nil : -> List [ctor] .
  op #_ : List -> Nat .
  eq # nil = 0 .
  eq # (E1 L) = 1 + # L .
  op mergeSort : List -> List .
  vars L L' : List .
  vars NEL NEL' : NeList .
  vars E1 E2 : X$Element .
  eq mergeSort(nil) = nil .
  eq mergeSort(E1) = E1 .
  ceq mergeSort(NEL NEL') = merge(mergeSort(NEL), mergeSort(NEL'))
    if (# NEL == # NEL') or (# NEL == s ( # NEL')) .
  eq merge(nil, L) = L .
  ceq merge(E1 L, E2 L') = E1 merge(L, E2 L') if E1 le E2 .
endfm
```
The module defines lists of the sort `Element` of the parameter `X`. The rest is our
mergesort function, with the comparison operator `_le_` is used for the sorting.

Assume that we want to sort integers. We cannot just write something like
`PARAM-SORT{INT}`, since it would be unclear whether we want lists of integers,
of natural numbers, or of any other sort in `INT`. And what should the comparison
operator `_le_` be? `_<=_`, `_>=_`, or something else?

Views define how the actual parameter module interprets the formal parameter
module. A view maps the sorts (resp. operators) of the formal parameter to sorts
(resp. operators or even expressions) in the actual parameter. For example, the fol-
lowing view `Int<=` says that we want to use `INT` as the actual parameter, with the
sort `Element` mapped to the sort `Int`, and with the function `_le_` mapped to `_<=_`:

```
view Int<= from ORDERED-SET to INT is
  sort Element to Int .
  op _le_ to _<=_ .
endv
```

The following module `SORT-INT<=` then defines lists of integers and the mergesort
function w.r.t. the usual comparison operator `_<=_`:

```
Maude> fmod SORT-INT<= is protecting PARAM-SORT{Int<=} . endfm
Maude> red mergeSort(5 2 11 23 -4 8) .
result NeList: -4 2 5 8 11 23
```

We can use the view

```
view Int>= from ORDERED-SET to INT is
  sort Element to Int .
  op _le_ to _>=_ .
endv
```

to sort lists of integers in decreasing order instead:

```
Maude> fmod SORT-INT=> is protecting PARAM-SORT{Int=>} . endfm
Maude> red mergeSort(5 2 11 23 -4 8) .
result NeList: 23 11 8 5 2 -4
```

Finally, we can also define lists of strings using the view

```
view String<= from ORDERED-SET to STRING is
  sort Element to String .
  op _le_ to _<=_ .
endv
```

which allows us to sort lists of strings according to `_<=_` on strings:

```
Maude> fmod SORT-STRINGS is protecting PARAM-SORT{String<=} . endfm
Maude> red mergeSort("Hi" "how" "are" "you" "today") .
result NeList: "Hi" "are" "how" "today" "you"
```
2.10 Some Other Maude Features

2.10.2 Telling Maude how to Evaluate an Expression

Consider a standard definition of the factorial function:

\[ \text{eq N \texttt{!} = if N > 0 then N * sd(N, 1) \texttt{!} else 1 fi} . \]

Although \texttt{if\_then\_else\_fi} is a built-in function, we could assume that it is explicitly defined by the following equations:

\[ \text{eq if true then X else Y fi = X} . \]
\[ \text{eq if false then X else Y fi = Y} . \]

The specification of \texttt{!} is nonterminating since we have the following derivation:

\[ 0 \texttt{!} \leadsto \text{if 0 > 0 then 0 * sd(0,1) \texttt{!} else 1 fi} \]
\[ \leadsto \text{if 0 > 0 then 0 \texttt{!} else 1 fi} \]
\[ \leadsto \text{if 0 > 0 then} \]
\[ \quad 0 * (\text{if 1 > 0 then 1 * sd(1,1) \texttt{!} else 1 fi) else 1 fi} \]
\[ \leadsto \text{if 0 > 0 then} \]
\[ \quad 0 * (\text{if 1 > 0 then 0 \texttt{!} else 1 fi) else 1 fi} \]
\[ \leadsto \ldots \]

Since the derivation started with \texttt{0 \texttt{!}} and has reached a term containing \texttt{0 \texttt{!}}, the specification is nonterminating. The point is that we assume that \texttt{if\_then\_else\_fi} first computes the value of its first argument, and then evaluates “itself” using the \texttt{if\_then\_else\_fi-}equations above. However, a term \texttt{if \_\_ then \_ \_ else \_ \_ fi} could equally well be evaluated by \texttt{first} evaluating \_\_, as happened above.

To avoid such undesired computations, and to increase the efficiency of Maude computations, we can tell Maude how to evaluate a term by defining an \texttt{evaluation strategy} of a function using the attribute \texttt{strat}. For example, a declaration

\[ \text{op f : s1 s2 s3 -> s [strat (2 0 1 3 0)]} . \]

tells Maude to first evaluate the second argument (2), then the whole term (0), then the first argument (1), and so on. That is, an expression \texttt{f(t1, t2, t3)} will be evaluated by first reducing \texttt{t2} as much as possible to \texttt{t2'}, and then simplify the term \texttt{f(t1, t2', t3)} “at the top” using \texttt{f-}equations. If the resulting term still has the form \texttt{f(u1, u2, u3)}, then \texttt{u3} is again evaluated, and so on. For example, \texttt{if\_then\_else\_fi} should have the attribute \texttt{strat (1 0 2 3 0)} (or even \texttt{strat (1 0)}), which states that the test is computed first, followed by the application of an \texttt{if\_then\_else\_fi-}equation.

The default evaluation strategy of a function with \( n \) arguments in Maude is, by the way, \( (1 2 \ldots n 0) \); such an execution strategy, in which all subterms are evaluated \textit{before} the entire term is evaluated, is called \textit{eager} evaluation. Likewise, a strategy that starts with \( 0 \) denotes \textit{laz}y evaluation, since subterms are not computed before the entire term is evaluated.

The evaluation strategy can have a significant impact on the efficiency of the computation. For example, efficient evaluation strategies for a function \( f \) defined by \( f(x,y,z) = y \) are \( (2 0) \) or \( (0) \), whereas the strategy \( (1 3 2 0) \) is unnecessary inefficient (why?).
2.10.3 Other Features

**owise Equations.** An equation of the form \( f(...) = t \) with the `owise` (for “otherwise”) attribute can only be applied if no other equation for \( f \) can be applied. This greatly simplifies the definition of some functions, as shown below:

vars \( I, J : \text{Int} \). vars \( L, L_1, L_2, L_3 : \text{List} \).

var \( N : \text{Nat} \). vars \( MS, MS' : \text{Mset} \).

eq \( I \) occursIn \( L_1 \) \( I \) \( L_2 \) = true .

eq I occursIn \( L \) = false [owise] .

c\( \text{eq isSorted}(L_1 \ I \ L_2 \ J \ L_3) = false \) if \( I > J \).

eq \text{isSorted}(L) = true [owise] .

eq I \text{ in } I \text{ MS} = true .

eq I \text{ in } MS = false [owise] .

It is worth remarking how easily the NP-complete *Subset Sum* problem [?]—which asks whether there is a subset \( s' \) of a given set \( s \) of natural numbers where the sum of the elements in \( s' \) equals a given number \( n \)—can be solved using the `owise` attribute and assoc and comm symbols (see also Section 3.3.2):

c\( \text{eq subsetSum}(MS \ MS', N) = true \) if \( \text{sum}(MS) = N \).

As explained in [?], the `owise` construct is not an extra-logical feature of Maude: any specification can be reduced to an equivalent one *without* `owise` equations.

**Formatting of Terms.** Large terms can sometimes be hard to read. Maude therefore provides an operator attribute `format` that can be used to control how terms are printed, e.g., with different colors and indentations.

**Tracing and Debugging.** Maude provides features for tracing the computations and gathering statistics about the number of executions of each statement, as well as an advanced debugger. They are all described in [?, Chapter 22].

**Exercise 36** What is the most efficient evaluation strategy for the functions \( f, g, \) and \( h \) in the specification

\[
\{ f(x) = x + x + x + x, \quad g(x, y, z) = a, \quad h(x, y, z) = k(y, y) \}\]

**Exercise 37** The Boolean tests `&&` and `||` evaluate their second argument only if necessary in languages like C and Java, so that \( b_2 \) is not evaluated in \( b_1 \&& b_2 \) if \( b_1 \) evaluates to “false.” The built-in functions `and` and `or` evaluate both their arguments in Maude:

Maude> `red 0 > 0 and (5 / 0 > 4)`.

result `[Bool]`: `false` and `5 / 0 > 4`

Define two Boolean functions `and-then` and `or-else` which work more like the C conjunctions and disjunctions.