Models of Software Systems

David Garlan, Jeannette Wing, and Orieta Celiku

August 17, 2010

Table of Contents

Preface

In 1992 a group of faculty undertook a project to rethink the core curriculum of the Carnegie Mellon Masters in Software Engineering degree program. The degree was designed for professional software engineers who would return to industry after the 16-month masters program. Our challenge was to come up with the outlines for a body of material that would provide life-long principles for effective software engineering, balancing both theory and practice in a set of five core courses.

One of our basic guiding principles for the curriculum design was the idea that each course should focus on skills that would retain their value well into the future, even as the face of software engineering was rapidly changing. Central among those skills was the ability to manage complexity of complex software systems. We also felt strongly that good engineering was based on the ability to reason about systems at a high level of abstraction, and that while software engineering was only then in the process of understanding how to do that for software-based systems, the skills of modeling and formal reasoning were central to the discipline. Out of those convictions arose the course Models of Software Systems, initially taught by two of the authors of this book.

But what to teach in such a course? At the time existing courses on formal modeling and formal methods either (a) focused on a single modeling notation, or (b) provided a survey of a variety of notations. Neither of these seemed appropriate. What was missing was an explanation of the enduring principles underlying modeling – abstraction, precision, refinement, formal reasoning. – and the ways in which those principles manifested themselves in the variety of formal approaches that are available.

To get at that essence, we realized that we needed to take a somewhat different approach: attacking the central ideas of formal modeling in a notation- and method-independent way, initially, and then seeing how those concepts could be made practical through specific modeling notations. What emerged over the next

several years was the three-part approach that we present in this book. We start with the subset of basic mathematics that is most centrally relevant to software modeling. Next we introduce the concepts needed to relate mathematics to computation: state machines, traces, invariants, and so on. Finally, we provide a set of examples of specific notations to show how the abstract modeling concepts can be realized in a concrete, scalable way, and how tools can be used to make tractable the job of specifying and analyzing models.

Initially when we taught this material we relied on standard mathematical texts for the first part and a variety of method-specific texts for the third part, bridging the gap in Part 2 with handouts. At some point, however, we realized that there was some value in putting it all together in a smallish book that could be used as a springboard for anyone interested in gaining an appreciation of formal modeling for practical software development. Moreover, we hoped that the foundations and examples covered by the book would be useful to others in the formal modeling community who might be interested in contributing additional modules to Part 3 – essentially making this text an extensible and continuously evolving asset to the formal modeling community.

This book and the course that motivated its creation owe much to the members of past and present members of the software engineering community at Carnegie Mellon University. In particular we would like to thank Daniel Jackson and Jim Tomayko, who served on the MSE curriculum committee that positioned Models of Software Systems in a central place in the software engineering curriculum; Masters students who took the course and made many good suggestions for improvement; and numerous teaching assistants who contributed to the body of exercises. In addition we would like to thank Daniel Kroening for his early contributions to this book and helping us formulate the overall plan of the book and its approach. We also thank Sungwon Kang and Paul Strooper, who provided detailed comments on early drafts, Microsoft Corporation, for supporting the production of this book through an educational development grant, and various funding agencies (The National Science Foundation, DARPA, NASA, and others), for supporting our research in formal methods.

Chapter 1

Introduction

The idea that mathematics could be used to understand computing systems is as old as electronic computation itself. From the earliest days of Turing and von Neumann, it was recognized that computation could be mapped into models that would allow one to use the formal power of mathematics to both characterize and also reason about a computational system. As computing matured many kinds of mathematical models were invented to represent different aspects of computing – from explaining the nature of computation itself, to understanding complexity of algorithms, to expressing the behavior of a computation in terms of its effects on inputs, to representing common patterns of behavior, to modeling phases of a compiler, to providing formal semantics for programming languages.

One important subgroup of this community was particularly interested in using mathematics to develop software systems. They recognized that mathematics could help humans master the complexity of a software-intensive system, and, through added precision, could provide methods for formally establishing the suitability of a computer program. Out of this group there grew an active body of researchers and practitioners who proposed "formal methods" as a way to develop software. The underlying concept was that by using systematic mathematical reasoning throughout the development of a software system, one could move from formal specifications of a system to a provably correct implementation of it. Alternatively, given a specification of a system and its implementation, one could prove that the implementation satisfied the specification.

Here again, a variety of notations, tools, methods, and logical theories were proposed. Many of these were taught in courses in universities, and there were several notable success stories from industrial software development in which formal models and methods were used to improve the quality of the resulting

system. Moreover, in situations where correctness was paramount, such as the software that runs heart pacemakers or nuclear power plants, the cost of gaining confidence in the systems through formal methods could be justified. Some even argued that if one considered the overall costs of maintenance and rework due to error, producing software using formal methods more than made up for higher costs at the front end of the software life cycle.

But despite several well-publicized successes, the overall impact of formal methods on software development was relatively small. Few software developers used any kind of formal techniques on real projects. Few managers were willing to commit resources to train their employees in the use of formalisms. And few commercially supported tools were available to assist the software developer who might want to apply those techniques.

However, in the past decade the situation has changed dramatically. Today we find numerous software systems companies that have adopted various kinds of formal modeling methods and model-based analysis tools. Standardization bodies, like the Object Management Group, promote many widely-used modeling notations, some based on solid mathematical theory. There are conferences on industrial uses of formal methods, and journals devoted to software modeling and analysis for large-scale systems. Undergraduates routinely learn concepts from formal methods, such as the use of pre- and post-conditions, the idea of a correctness proof, and various uses of state machines.

Indeed, it is increasingly the case that any well-trained software engineering professional should have some familiarity with formal models and formal methods. More importantly, formal models are quickly becoming an essential tool of a practicing software engineer to create complex systems with acceptable and predictable levels of reliability, performance, and security.

Why the change? What is so different now? Why is it suddenly so important to understand how to create and use formal models?

Broadly speaking, two major factors have driven this new state of software engineering practice: the nature of the software systems today, and the technology that supports its production.

1.1 The Nature of Software Systems Today

As software has become increasingly pervasive, and as everyday systems depend more and more on software, requirements for software reliability, safety, availability and utility have risen dramatically. Without effective and reliable software

our current systems of banking, medical care, commerce, national security, entertainment, communication, transportation, and energy would be unthinkable. Such software-based systems must continue to run, even in the presence of flaws. They must serve millions of people on a daily basis. They must provide high degrees of security and privacy for their users.

To handle the increasingly demanding requirements of modern software-based systems the complexity of the underlying software has also risen dramatically. Many types of systems that once were differentiated primarily by their hardware, are now commercially competitive by virtue of value-added features provided by software. As a result, the amount of software resident in appliances, cars, telephones, televisions, and so on, is increasing dramatically.

But in addition to increased size and functionality of software, complexity due to the context in which systems must operate has also risen dramatically. Currently most systems must function in a distributed setting, communicating with other systems over networks or other communication channels. They must interoperate with other systems, often unknown at the time of their creation. They must be engineered in ways that permit incremental introduction of new capabilities. They must be built by development teams spread across the globe.

In short, the day of the simple stand-alone application, created, distributed, and maintained by a single co-located organization, is over. Software engineers must now create systems that exhibit high degrees of availability, reliability, security, and interoperability. Maintaining intellectual control over such systems becomes a daunting task.

In that context techniques for managing complexity and for ensuring critical system properties during design become not just a luxury, but a necessity. Formal models by their very nature can play a significant role in that regard. Through abstraction they allow software engineers to focus on the critical issues facing them. Through precision they provide a way to document intended functions and properties of a system to be built. Through refinement they provide ways to help guarantee that implementations respect design principles and properties. Through logical foundations they support the ability to perform analyses to determine consequences of design and implementation choices.

1.2 Enabling Technology

The second important change that is leading to increased use of formal methods is the dramatic improvement in technology that supports automated analysis. In

the early days of formal methods reasoning about a formal model was largely a manual exercise. Establishing properties of a model, or relating two models, usually boiled down to proving a set of theorems. And while several powerful automated theorem provers and semi-automatic proof assistants were developed in research communities, their use required considerable mathematical expertise and patience to develop all of the underlying theories and lemmas required to demonstrate even the most rudimentary properties of a formal model.

Over the past decade, however, numerous automated tools have made the job of analyzing formal models considerably more tractable and accessible to practicing software engineers. One important class of tools is model checkers. A model checker takes a formal model and a property to check about it. The checker then explores all computational paths of the model, and either certifies that the property holds over all of those paths, or provides a counterexample that shows a computational sequence that leads to the violation of the property.

Model checkers found initial success in the hardware design domain, where their use is now *de rigueur*. But over the past decade considerable progress has also been made in applying these tools to software-based systems. While there remain obstacles to their use on very complex software systems, for many restricted domains, or for sufficiently simple models of a system, they can be remarkably effective in increasing the engineering payoff of formal modeling by providing ways to explore properties of those models.

Other tools for static analysis that take advantage of formal specifications have drastically improved our ability to eliminate certain classes of errors from our systems. For example by embedding certain annotations in code, one can use tools to check for absence of race conditions in concurrent code, or the absence of buffer overruns in software that would otherwise permit it.

Moreover, the state of automated theorem provers and proof assistants has also improved considerably since their early days. A number of fully automated theorem provers for specialized theories have been developed. They take advantage of advances in optimization and machine learning techniques for proof search. Powerful interactive proof assistants incorporate automated theorem provers and link to other external tools (such as SAT solvers and model checkers) to improve performance. Theorem provers have already been used to prove impressive mathematical theorems, some never proved before. Some large industrial verifications have also been carried out.

1.3 Formal Modeling as an Engineering Enterprise

Given the increasing importance of formal models and formal methods, as well as the tools that can be used to create and analyze them, the question arises: what should the practicing software engineer know about formal modeling?

In this book we attempt to answer this question by providing an introduction to the foundations and uses of formal modeling. The key driver of our approach is the view the *way in which formal modeling is used* on a particular system or family of systems should be based on sound engineering choices. That is to say, the costs of formalization and analysis must be commensurate with the expected benefits. In other words, formalism is not good for its own sake: it is good because of the improvements in the systems that we can realize, and its use comes at a cost that must be weighed against other techniques that a software engineer can use to improve systems.

This engineering perspective has led us to write a book that adopts several philosophical principles that frame our exposition of formal modeling.

The first principle is the use of lightweight models. In the early days of formal methods, advocates promoted the notion that a system should be completely specified and verified for correctness against that specification, or, alternatively, derived formally from it. In contrast, today it is recognized that it is possible to gain significant benefit from partial models. Such models may be partial because they characterize a system at a high level of abstraction. Or they may be partial because they model only one part or one aspect of a system. In either case, the use of lightweight formal models makes it possible to apply the formal techniques to the most pressing part or aspect of a system. In this way engineering benefits are maximized, while the effort is minimized.

As we elaborate in the remainder of this book, techniques for creating appropriate abstractions or for determining what aspects of a system to model are at the heart of effective formal modeling. Thankfully, many of these techniques are now well understood, and while their application differs from one modeling approach to another, it is possible to learn general principles of adopting the right level of abstraction or partial specification to suit the problem at hand.

The second principle is the selection of modeling approach to match the problem at hand. In the early days of formal methods, researchers debated the relative merits of alternative modeling approaches, hoping to demonstrate superiority of one over another. Today, we recognize that each formal modeling approach has its strengths and weaknesses, its benefits and costs - and that it is important first to ask what goals we want to achieve before selecting an approach. For example,

one kind of model may be ideal for analyzing protocols of communication, while another might be better suited to understanding the relationships between data. Other models may be better for understanding the performance or the reliability of a system.

In the succeeding chapters we will be considering a variety of modeling languages and their associated methods. As we describe each, we will be looking for a clear understanding of the contexts in which they are appropriate and the kind of problems that they can best solve. The goal is to empower the engineer with the ability to select a formal modeling approach that will best solve a problem at hand.

The third principle is the use of tool-assisted reasoning. As we noted earlier, one of the enablers of formal methods has been the remarkable improvement in tools for reasoning about formal models. We believe that such tools should be leveraged as appropriate. Of course, as with the choice of modeling approach, the strengths and weaknesses of a given tool must be evaluated with respect to its engineering benefits to the project.

In this book, as we introduce various modeling approaches we will also attempt to describe the way that tools can be used to assist in analysis of the models. While we will not be able to talk about individual tools in depth, we will try to show what kinds of benefits various tools provide, and give examples of how they are used.

The fourth principle is that formal models are engineering artifacts themselves. That is to say, when we create a formal model we should be concerned not only with what it tells us about a system, but also the properties of that formal model that make it a usable engineered artifact. In particular, we need to consider how easily the models described using a given approach can be incrementally extended, enhanced, composed with other models, read by software developers, reused across different development projects, and so on.

As we will see in later chapters, different models take very different approaches to these engineering concerns. Specifically, the way in which a modeling approach allows us to compose a model from smaller models becomes a crucial discriminator when understanding the costs and benefits of that approach.

1.4 A Guide to Using This Book

The remaining chapters of this book are structured into three conceptual layers, each building on the other. The first layer sets out the mathematical foundations on

which the rest of the book is based. It presents the mathematical concepts, such as logics, proofs, theories, sets, functions, relations, etc., on which all modern modeling approaches build. For some readers this material will be familiar from a course in discrete mathematics. In that case, a quick skim of that part of the book may be sufficient to remind the reader of those concepts and to become familiar with the specific mathematical notation that we use in this book. For other readers, particularly those who have not been exposed to a course in mathematics for computer science, this may be a particularly challenging section of the book. Indeed, the reader may want to refer to one of many textbooks in this area for additional examples and practice beyond what we can offer in this book.

The second layer presents the concepts that allow us to relate raw mathematics to computation and to software. We consider topics such as state machines, invariants, pre- and post-conditions, proving properties about programs, and so on. The goal here is to introduce these concepts in the simplest possible way, independently of any specific notation or method. To the extent possible we will rely on standard mathematical notations, sugared only enough to make the job of specifying computations more natural.

The third layer consists of a set of modules, each focusing on a particular notation and method of formal modeling. The goal in this layer is to show how the general concepts of the second layer can be turned into effective engineering tools through the specialization of the concepts, and the introduction of special-purpose notations and tools. For the purposes of the book we will provide a small number of such modules to illustrate some of the more important parts of the space of formal modeling approaches. Our hope is that over time the set of such modules will continue to grow, and that others in the community of formal modeling will contribute their own modules.

An important cross-cutting theme for the book is the use of exercises to illustrate the main points of the text. We strongly encourage the reader to try out these exercises, as there is no substitute for engaging directly in the process of formal modeling. Additionally, we use some of the exercises to explore certain themes in formal modeling that we do not have the space to detail in the main body of the text.

Finally, each chapter of the book contains a list of additional readings. The field of formal methods is large, and in this book we can only scratch the surface. The readings point the way to more in-depth treatment of many of the topics that we cover, and many that we can only hint at.

Further Reading

[TBD]

Part I

Foundations

Introduction

The starting point for any treatment of formal modeling is mathematics. Essentially all approaches to formal methods are founded on mathematical principles, and look to mathematics for the underlying mechanisms of reasoning about precise models.

But what parts of mathematics are needed? The field of mathematics is huge in its own right, and arguably almost any branch of it might have *some* applicability to software systems. Moreover, learning sophisticated mathematical techniques and ideas could occupy many courses by themselves.

Thankfully, over the past two decades there has emerged an understanding that, in fact, most of the models needed by a practicing software engineer rely on a relatively small set of mathematical notions. In many cases these concepts are taught in specific courses, labeled by names like "Discrete Mathematics" or "Mathematics for Computer Scientists."

It is this body of material that we will examine in Part 1. We start by introducing the idea of a formal model, illustrating how mathematical abstraction can help us reason about interesting systems. As we will see, every formal system is constructed from a certain set of basic building blocks that determine what kinds of models we can express, and what kinds of judgments we can make about those models. Next we consider the logical apparatus that we will need to reason formally about mathematical models. This covers much of the standard material on propositional and predicate logic, with particular emphasis on the ability to translate between formal and informal models. Finally, we consider the building blocks for creating models of complex software systems and their behavior: sets, relations, functions, sequences, and so on.

In presenting the material of Part 1 we will attempt to find a middle ground between thoroughness and brevity. While we do not expect Part 1 to substitute for a full course in discrete mathematics, we hope that it will provide a solid overview of the concepts, and we include pointers for readers who may want to go beyond

what we cover here. Additionally, we include a number of exercises that explore areas outside the central focus of this book.

Chapter 2

Formal Models

2.1 Models in Engineering

One of the hallmarks of any engineering discipline is effective use of system models. For example, civil engineers use stress models to determine whether the supporting structure for a bridge will support anticipated traffic loads. Aeronautical engineers use airflow models to design wing surfaces of jets. Electrical engineers use heat-transfer models to reason about power consumption of a computing device.

Models are central to the engineering enterprise because they permit engineers to reason about a system design or implementation at a level of abstraction at which the system's essential properties can be better understood. This capability in turn supports early exploration of design tradeoffs, often making it possible to try out various approaches to system design before committing to a particular solution. These models are often useful in detecting design flaws early in the system development cycle, when errors are relatively less expensive to fix. In some cases engineering models can also be used as blueprints for more detailed design and implementation.

When an engineer decides to use models there are a number of important considerations in choosing what to model and how to model it. These considerations arise from the fact that engineering resources are limited, and building, analyzing, and maintaining models takes time and effort. So a central question is how can an engineer get the maximum value out of the modeling effort?

2.2 Choosing the Right Models

To answer this question it is important to realize that there are three essential considerations that an engineer is typically faced with: What types of models to use? What to represent within a given model? How to relate different models? Let us briefly consider each.

First, there are many kinds of models that could potentially be useful to an engineer. Normally different models have different strengths and weaknesses. For example, one model might be suitable for discovering stress points of a bridge, while another might be suitable for creating a work plan for its construction.

Thus a key issue is how to pick the most appropriate model(s) for a given purpose. Normally that choice requires the engineer to have a deep appreciation of the benefits and costs of using specific kinds of models. And this will often vary depending on the kind of system, the skills of the design team, the need for having answers to certain kinds of design questions, the availability of tools, and many other factors.

Second, having selected a particular type of model, there are issues of fidelity of representation. In order to be useful, an engineering model defines an abstraction of the system under consideration. That is, models represent certain aspects of a system, hiding other aspects that may not be important for a specific kind of analysis or evaluation. For example, in one model a complex subsystem might be treated as a primitive "black box," while a more detailed model might expose the details of that subsystem for inspection and analysis.

The need to use abstraction leads to the question of what level of detail should be modeled. Once again, pragmatic considerations usually dictate the answer. The more faithful a model is to the system, the more accurate predictions one can make with it. But also the more effort it takes to create and analyze that model. Additionally, if there is too much detail the model may become so complex that it is difficult to understand or analyze. So how does an engineer decide on the right level of abstraction?

Third, the possible use of multiple models raises the issue of relating them to each other. This is an important issue because models are often interdependent. That is, properties of one model should have some relationship to properties of the other models. For example, it would be unfortunate if an architect's electrical models inadvertently prescribed allocated space to be used for electrical conduit, while a plumbing model simultaneously allocated it to water cooling pipes.

One particularly common situation in which multiple models are used occurs when an engineer develops a series of models of the same kind, each representing

increasingly lower-level designs. Ideally one would like to make sure that properties of the more abstract models are preserved by the lower-level ones. (This is sometimes referred to as a refinement relationship.) So the question arises, how can we guarantee cross-model properties, and at what cost?

2.3 Models for Software Engineers

An engineering discipline for *software* also needs models. Software systems are increasingly complex systems and can clearly benefit from better ways of understanding their properties, reasoning about consequences of design and implementation decisions, and identifying potential flaws.

Indeed, software engineers already use many kinds of models. For example, class diagrams for an object-oriented design are a kind of model that can help software engineers express relationships between the types of objects in the system. Among other things, such models allow software developers to reason about the potential for reuse, coding dependencies, and ontological relationships between the elements of a system. Similarly, real-time systems models can be used to specify the timing behavior of a set of executable tasks, and schedulability analysis can permit an engineer to reason about the ability of a system to meet its timing deadlines.

As with other engineering models, the questions noted above arise: What models should a software engineer use? What level of abstraction should a given model adopt? How can one relate multiple models?

In this book we hope to provide guidance for answering these questions. While we cannot describe *all* possible software engineering models, we show how to approach these questions systematically so that you will have the intellectual tools to answer the questions for yourself.

2.4 Formal Models

In practice, models can be represented in many ways. In some cases they may be defined as systems of equations. In other cases a physical artifact might be constructed. In others, rules of thumb might be used to reason informally about some aspect of a system.

In this book we are particularly interested in *formal models*. Formal models represent a system in such a way that their meaning can be defined in terms of

mathematics. The advantage of formality is that such models are (a) precise – their meaning is unambiguous; (b) formally analyzable – we can use mathematical reasoning to determine properties of the model; and (c) mechanizable – they can be processed and analyzed by computer programs.

These three properties are critical in applying formal modeling to real systems. Without precision we would be unable to satisfy our need to communicate our ideas unambiguously to others, or to have confidence that we have expressed what we intend. Without analyzability, models have limited usefulness: in general, the effort required to produce them would not be commensurate with the benefit derived. Without tools to process our models, it is difficult to scale them to the needs of practical systems.

We are also tangentially interested in formal *methods*. Formal methods prescribe the way in which you can create and reason about formal models. In particular, they provide guidance on

- how to pick a model;
- how to use a given modeling language effectively;
- how to relate different kinds of models;
- how to take advantage of existing bodies of knowledge for simplifying the task of formal modeling; and
- how to use tools to assist with the modeling process.

Usually such methods are tied to specific kinds of models or domains. For example, methods for refining models from abstract to more concrete usually depend on a particular notation or logic.

2.5 An Example

Let us now consider a simple example to illustrate the points we have been making. Suppose we are given a description of a game that is to be played as follows: We start with a large container that contains a number of identical white and black balls. We are also given a large supply of black balls on the side. To play the game we repeatedly draw two balls randomly from the container, and then put a single ball back in the container. The rules for deciding what color ball to put back are as follows:

1. *"If the two selected balls are both black or both white, put a black ball back into the container."*

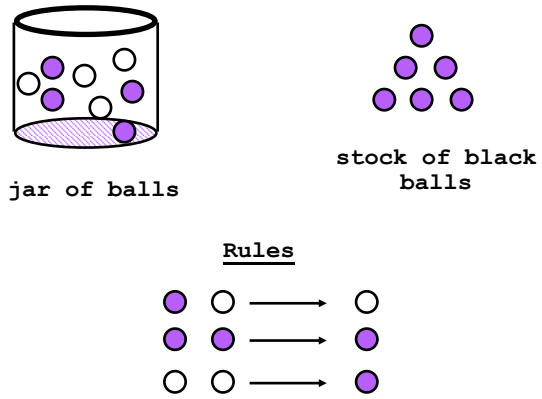


Figure 2.1: Game set-up and rules

2. *"If one ball is white and the other black, put a white ball back into the container."*
3. *"If there is only one ball left in the container, stop."*

Figure 2.1 illustrates the game set-up and rules.

Let us now see if we can say anything about how this game behaves. In particular, can we predict anything interesting about it? One question that naturally arises is: Is the game guaranteed to stop, regardless of the number and color of the balls in the initial container?

In this case a little informal reasoning can help out. We might argue that, yes, it does stop because the rules tell us that at each turn we take out two balls and put one back. Thus at each turn we decrease the overall number of balls in the container by one. Since there are only a finite number of balls initially in the container, eventually only one will be left – at which point the rules tell us to stop.

Of course, a skeptic might like to have a stronger form of argument than this, and you might consider what it would mean to prove the result more formally. After all, we are reasoning about an arbitrary number of initial balls, and so we need to make sure that our argument would apply no matter how many balls are in the container initially, and how we withdraw them.

Rather than taking that diversion, however, let us move on to a somewhat harder question. Can we say anything about the final configuration of the container once the game does stop? That is to say, can we predict the color of the final ball in the container? This question has a far less obvious answer, and it is not clear what we might do to make progress in answering it.

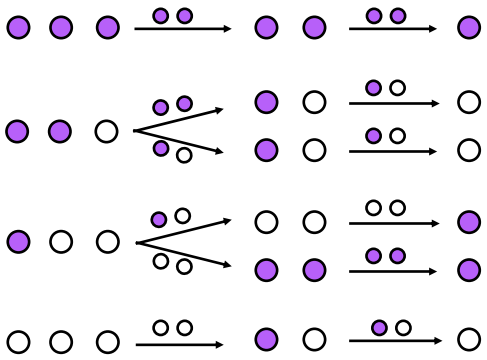


Figure 2.2: Experimenting with the game

One possibility is to use a form of experimentation, or, as we say in the software business, "testing." In other words, we could try out the game for various starting configurations and ball selections, and see what happens as a result. That is we "execute" it and see if we can detect any significant properties. If we were to approach this systematically, moving from simple configurations to more complex ones, we might start with a container with just two balls, then move on to three, four, and so on.

In the first round of testing, with only two balls, the game rules directly prescribe what the possible results will be. So far we have not learned anything new. For the next round of testing we consider a container with three balls as illustrated in Figure 2.2. We explore all possible scenarios as illustrated.

Is the situation any clearer? Probably not, although we might note that the color of the final ball does not seem to depend on the order in which we withdraw balls. For example, in the case with two black balls and one white ball, whether we start by picking two black balls, or a white and a black, does not change the final outcome.

Let us move on to testing the situation with four initial balls. Once again we explore all of the possibilities, and note that our observations made earlier still hold. But we may not yet have a good idea or any deep insight into the game. We could go on exploring possible outcomes in this fashion, hoping that some pattern might emerge. If we are lucky we might spot one, but it is becoming increasingly difficult to manage all of the cases, whose number is growing exponentially as we increase the number of balls in the container.

Now let us consider a different approach. This time we will produce a formal model of the system. To do this we will need to figure out what we want to

b = black, w = white, T = transition relation

$$
\begin{array}{rcll}
 & & (b-2+1, w) & \text{(Rule 1.)} \\
T(b,w) & = & (b+1, w-2) & \text{(Rule 1.)} \\
 & & (b-1, w-1+1) & \text{(Rule 2.)}
\end{array}
$$

$$T(0,1) = (0,1) \quad T(1,0) = (1,0) \quad \text{(Rule 3.)}$$

Figure 2.3: Modeling the game

$$
\begin{array}{rcll}
 & & (b-1, w) & \text{(Rule 1.)} \\
T(b,w) & = & (b+1, w-2) & \text{(Rule 1.)} \\
 & & (b-1, w) & \text{(Rule 2.)}
\end{array}
$$

Figure 2.4: A transformed model

represent in that model. Figure 2.3 has a summary of the model we choose. The basic idea is to represent the game as the value of two numeric variables, b and w, representing (respectively) the number of black and white balls remaining in the container.

We then prescribe the rules of the game in terms of those variables. We will do this by defining a transition relation T that says how we change the values of b and w on each turn. In Figure 2.3 you can see that the three cases are defined to fit the rules listed above. For example, the first rule models the removal of two black balls: we decrease b by two when we remove the balls, and then increase it by 1 when we put a black ball back into the container. The final line describes what happens when there is only one ball left: we leave the configuration the same.[1]

Thus far we have not done anything more than use a mathematical notation to express what we already knew. But let us now manipulate the model a little bit. In this case we will perform simple arithmetic on T to get the simplified rules of Figure 2.4.

Does the model now suggest something interesting? As a hint, note that the number of white balls either remains the same or is decreased by two. What does that imply?

The answer is that the color of the final ball can be predicted if we know something about the original number of white balls in the container. If w is even the result will be black; if odd, then white. To see why this is so, we note that

[1] The observant reader might notice that our model does not actually "stop," but we won't worry about that detail here.

the parity of w (i.e., its "even"-ness or "odd"-ness) is never changed. Hence, if we start with an even number of white balls there will always be an even number. That means it is impossible to have a single white ball in the container. So we know that we cannot end the game with a white. Similarly for the case where w is odd.

Impressed? Maybe not, but let us just review what we did. First we created a mathematically-based representation. This allowed us to say precisely what happens in the game. In doing this we had to be clear, however, about what each of the mathematical parts meant in terms of the phenomena of the real world (w representing the number of white balls, T representing a "turn" in the game, etc.).

Second we used abstraction to suppress irrelevant details of the game. For example, we did not represent the extra black balls. We ignored the fact that in real life the container can only hold a finite number of balls. We did not say anything about the shape of the balls or color of the container. And so on. The use of abstraction gave us a simpler way to view the game, and (hopefully) to spot interesting properties.

Third, we used simple transformation rules from mathematics (in this case arithmetic) to simplify the initial description. Less obviously, we also used a result from number theory to reason about the "even"-ness of white balls. This was necessary in arguing that the "even"-ness of w does not change when we increase or decrease it by two.

Of course, reasoning about software systems will in general involve much more complex models than this, but the basic principles are the same: we pick some aspects of a system to model; we express the model in a formal notation rooted in mathematics; and we reason about the model to infer interesting properties of the original system. In the remainder of the book we will see how we can apply these same techniques to complex software-based systems.

Chapter Notes

The example of the black and white balls was adapted from the book *The Specification of Complex Systems* by B. Cohen, W.T. Harwood, and M.I. Jackson [2].

Further Reading

[TBD]

2.6 Exercises

1. Consider the game described in this chapter. Suppose that the container starts out with N balls.

   (a) List five aspects of the real world that were *not* represented in our formal model.

   (b) How many "turns" will it take for the game to stop?

   (c) What is the largest number of extra black balls needed, and what configuration of the container causes this number to be required? Assume that when two black balls are taken out of the container one is put back into the container and the other into the stock of extra balls.

   (d) Argue formally that the game stops.

2. Consider a simple version of the game of Nim (as presented in [1]) in which two players alternate removing one or two toothpicks from a pile of N toothpicks. The game stops when there are no toothpicks left in the pile; the player who removes the last toothpick loses the game.

   (a) Describe a formal model of the game by (a) specifying the state model, and (b) giving a transition relation that describes the valid moves of the game in terms of changes to the state.

   (b) What aspects of the game are represented in the model? What aspects are left out? Would the model be different for more than two players?

   (c) Argue that the game eventually stops.

   (d) When $N > 1$ and $(N \bmod 3) \neq 1$ the first player has a winning strategy. What is that strategy?

   (e) Extend your formal model to encode the winning strategy for the first player. That is to say, the rules should automatically lead to a win for the first player. What new aspects do you need to represent?

Chapter 3

Formal Systems

The goal of this book is to help you understand how to model complex software systems. There are many ways in which we might describe these systems. Here we will be particularly interested in approaches that have a mathematical basis, and hence allow us to be precise about what we want to model, as well as to reason about properties of the models. That is, we are interested in formal models.

But there are many kinds of formal models. For example, some are based on systems of equations. Others on mathematical logic. Others on computational rules. Each of these may have different notational structure and specific ways of reasoning about it. How are we to make sense of this complex space of possibilities?

Thankfully, all types of formal models have a similar underlying form. First, they define a *language* for describing a certain class of models. And second, they provide a set of *inference rules* for manipulating models of that type and for proving results about them. As we will see in this chapter, the combination of these two things — a formal notation and a set of inference rules — defines a *formal system*. In addition, we will also be interested in assigning a meaning to the models in a formal system by relating them to some domain of interest, or defining their *semantics*. Finally, there may exist a body of useful *results* associated with a type of model. These will make our life a lot easier by giving us a rich starting point for working with that type of model.

3.1 Formal Languages

The first step in creating a formal system is to define a language with which to describe models of that system.

As an example, suppose we would like to define a language, called *Decimals*, for describing decimal numbers. The kind of expressions that we have in mind are those like 42 and 3.14. Informally we might go about defining the language by saying that a decimal number is either (a) a simple number consisting of one or more digits, 0 through 9, or (b) a compound number consisting of a simple number, followed by a decimal point, followed by another simple number.

This kind of informal definition is fine as far as it goes: using it we can probably infer that the two examples above are acceptable expressions, while 3.5.6 and 2..7 are not. But what about expressions like 45. or .22? To resolve issues such as these we will need a more precise and complete way of characterizing what expressions are legal in the language.

In this book we use a common form for describing a formal language. A *formal language* is defined by a set of symbols, called its *alphabet*, together with a collection of *syntax rules*, called a *grammar*, that prescribes how those symbols can be combined to form expressions in the language. When the alphabet can be inferred directly from the grammar, or is otherwise obvious, we can choose not to specify it explicitly.

We will use a standard notation (or meta-language) to define the syntax rules of the grammar. Each rule will describe a grammatical element of the language, and will be given a name that can be referenced by other rules. Following the rule name is an equal sign, and then a sequence of grammatical elements. Grammatical elements may indicate a choice between alternatives, indicated by a "|", or a sequence of elements, each specified by its rule name and separated by a ",". Symbols from the alphabet are enclosed in quotation marks.

Example 3.1. The language *Decimals* has alphabet $\{0,1,2,3,4,5,6,7,8,9,.\}$. Its syntax rules are defined as follows:

$$
\begin{array}{ll}
\textit{decimal number} & = \textit{number} \mid \textit{number}, \text{“.”}, \textit{number} \\
\textit{number} & = \textit{digit} \mid \textit{number}, \textit{digit} \\
\textit{digit} & = \text{“0”} \mid \text{“1”} \mid \text{“2”} \mid \text{“3”} \mid \text{“4”} \mid \text{“5”} \mid \text{“6”} \mid \text{“7”} \mid \text{“8”} \mid \text{“9”}
\end{array}
$$

□

In Example 3.1 the first rule of the grammar, *decimal number*, expresses the idea that a decimal number can be either a simple number or else two simple numbers separated by a decimal point.

The second rule, *number*, expresses the idea that a simple number is a sequence of digits. To express this idea the rule uses the choice operator: a number is either a single digit, or a number followed by a single digit. Note that the rule is recursive. That is, it uses the rule's name (*number*) as part of its own definition. Hence, numbers of arbitrary length can be built up by applying the rule multiple times.

The third rule, describes a *digit* as any of the ten decimal digits of the alphabet.

Such rules determine whether a given sequence of symbols in the alphabet of the language is a *well-formed formula*, or *wff* (pronounced "woof"). A sequence of symbols is a wff of a grammar rule in the language if and only if its structure conforms to that grammar rule (and any other rules on which that rule depends). Normally we will designate one of the rules as the "topmost" rule for the grammar – typically the first rule – and say that a *w* is a wff of the language if it conforms to that rule. Determining whether a given sequence of symbols is a wff of the language is often called *parsing*.

For example we can see that for Example 3.1, as expected, 42 and 3.14 are wffs. The first is a wff by virtue of the first branch of the *decimal number* rule. The second is a wff by virtue of the second branch of it. But 45. and .22 are not wffs, since according to the first rule any decimal number with a decimal point must have a simple number on either side of it.

Example 3.2. Consider a language *StarDiamond* with alphabet $\{\diamond, *\}$ and the following grammar:

$$\begin{aligned} \textit{expression} &= \textit{stars} \mid \textit{diamonds} \mid \textit{stars}, \textit{diamonds} \\ \textit{stars} &= \text{“}*\text{”} \mid \textit{stars}, \text{“}*\text{”} \\ \textit{diamonds} &= \text{“}\diamond\text{”} \mid \textit{diamonds}, \text{“}\diamond\text{”} \end{aligned}$$

The following are wffs in this language

- $***$
- $\diamond$
- $**\diamond\diamond\diamond$

while the following are not wffs

- $*\diamond*$
- $\diamond\diamond\diamond**$

□

Example 3.3. As another example, consider the language *Smileys* with alphabet { : , ; , - ,) , (}, and the following grammar:

$$\begin{aligned} \mathit{smiley} &= \mathit{eyes}, \mathit{mouth} \mid \mathit{eyes}, \mathit{nose}, \mathit{mouth} \\ \mathit{eyes} &= \text{“ :”} \mid \text{“; ”} \\ \mathit{nose} &= \text{“}-\text{”} \\ \mathit{mouth} &= \text{“}(\text{”} \mid \text{“})\text{”} \end{aligned}$$

The language includes wffs such as “ : -) ” and “ ; (”, but not “ : : -)) ”. □

3.2 Semantics

It is important to note that wffs in a formal language are merely strings of symbols: there is no intrinsic meaning associated with them. This is particularly clear with a language like *StarDiamond*. For example, we might choose to interpret “$*$” as a 2, “$\diamond$” as a 3, and juxtaposition (following one symbol by another) as addition. In that case, the wff $**\diamond\diamond\diamond$ would represent the number 13. On the other hand, we might just as easily interpret “$*$” as a 10, “$\diamond$” as a 5, and juxtaposition as multiplication. In that case the same wff would denote the number 12500.

Hence in order for a language to be useful we will need to explicitly assign meanings to the wffs in that language. To do this we will need to pick a *domain of interest* and rules that tell us how each wff in the language is mapped to some value in that domain. Such an assignment of meanings is called an *interpretation* of the language. Providing an interpretation is often called giving the language a *semantics*.

In many cases there will be a *natural* interpretation for a language. For example, it would be surprising if the domain of interest for the language *Decimals* were not the decimal numbers, in which, the symbol 1 denotes the number one, 2 the number two, etc. Similarly in the language of set theory (Chapter 6), the wff $\{1,2\}$ would naturally be interpreted as the set containing the numbers one and two.

In other cases we will need to be explicit. For example, in the language of propositional logic (Chapter 4), to interpret a wff like $p \wedge q \Rightarrow r$, we will need to be clear about the interpretation of its constituent symbols p, q, and r.

There are many ways that one might go about defining the semantics of a language. Indeed, a study of ways in which one can do this formally is itself an important subfield of computer science. However, a detailed examination of this topic is beyond the scope of this book. For now we will use relatively informal

ways of assigning a semantics to a language, relying whenever possible on natural interpretations and context to make the interpretation clear.

3.3 Inference Systems

Having explained how we can formally define a language and what it means to give its wffs an interpretation, we now look at ways in which we can manipulate wffs as purely syntactic entities – that is, focusing only on their symbolic structure, and not on any interpretation of the symbols. We will call such a symbol manipulation system an *inference system.* The combination of a formal language and an inference system is termed a *formal system.*

An inference system is composed of two parts: a collection of axioms and a collection of inference rules. *Axioms* are wffs that can be written down without reference to any other wff. *Inference rules* allow us to produce new wffs as consequences of other wffs.

Inference rules typically are of the form: "If you see a set of wffs exhibiting certain syntactic structure, then you can write down a new wff whose structure is determined from those other wffs in certain prescribed ways." The new wff is said to be an *immediate consequence* of the preceding wffs and the inference rule that was used to generate it.

Example 3.4. Let's illustrate with an example. Consider the formal system, called *Stars*, with alphabet $\{\diamond, *, \circ\}$ and grammar

$$\begin{aligned} \textit{sentence} &= \textit{stars}, \text{“}\diamond\text{”}, \textit{stars}, \text{“}\circ\text{”}, \textit{stars} \\ \textit{stars} &= \text{“}*\text{”} \mid \textit{stars}, \text{“}*\text{”} \end{aligned}$$

Wffs in this language consist of three strings of stars separated by a diamond and a circle. Examples of wffs in this language include

- $*\diamond*\circ**$
- $**\diamond*\circ***$
- $*\diamond**\circ******$.

The inference system for *Stars* consists of the following:

Axiom A $*\diamond*\circ**$

Rule R If $m\diamond n\circ r$ is a wff, where m, n, r are strings of stars, then an immediate consequence is $m \diamond n* \circ r*$.

Using this inference system we can apply rule R to the wff $*\diamond***\circ***$ to get $*\diamond*****\circ*****$ as an immediate consequence, where the m, n, and r in R are filled by $*$, $**$ and $***$, respectively. □

In the example above, we characterized the inference rule informally. To eliminate ambiguity in rule definitions, we will need a way to specify rules more precisely. To do this we will use the following general form:

$$\frac{\text{existing patterns}}{\text{consequence pattern}}\ \text{rule name}$$

Above the line is a list of the wff structures required to apply the rule. Below the line is the resulting structure derived from the existing ones. The rule name appears on the right. (In some cases we will also include an extra condition of applicability, called a *side condition* – we will see examples of this in later chapters.)

To describe wff patterns we use *schema variables*. These are variables that can be instantiated with any wff of the appropriate grammar rule.

Example 3.5. The inference rule from Example 3.4 would be written formally as

$$\frac{m \diamond n \circ r}{m \diamond n * \circ r *}\ \text{R}$$

In this rule m, n, and r are schema variables representing arbitrary sequences of stars. □

Although *Stars* stands on its own as a purely syntactic system, we can also associate a semantics to it.

Example 3.6. Consider the following interpretation of *Stars*

$$\begin{array}{lcl} * & \rightarrow & 1 \\ ** & \rightarrow & 2 \\ *** & \rightarrow & 3 \\ \text{etc.} & & \\ \diamond & \rightarrow & + \\ \circ & \rightarrow & = \end{array}$$

That is, a string of N stars denotes the number N. For example, a wff of the form $\overbrace{**\ldots*}^{m}\diamond\overbrace{**\ldots*}^{n}\circ\overbrace{**\ldots*}^{r}$, containing strings of stars of length m, n, and r represents a statement of the form $m+n=r$.

When interpreted in this way, we can evaluate whether a given wff in *Stars* is true or false. For instance, the wff $*\diamond***\circ****$, denoting $1+2=3$ would be true, while the wff $**\diamond***\circ****$, denoting $2+2=3$ would be false.

Notice that the inference system of *Stars* makes sense according to this interpretation. The interpretation of the axiom $*\diamond*\circ**$ is true since $1+1=2$. And the inference rule says that if we know $m+n=r$, then we can conclude $m+n+1=r+1$. □

3.4 Proofs and Theorems

We can now define what we mean by a theorem and a proof. A *proof* in a formal system F is simply a sequence of wffs in which each wff is either an axiom of F or is an immediate consequence of one or more preceding wffs via an inference rule of F. A *theorem* is any wff that appears as the last wff in a proof. If W is a theorem in F, we say that W *can be proved in* F, and write $\vdash W$ to denote this fact.[1]

Example 3.7. The following are theorems of *Stars*:

1. $*\diamond*\circ**$ is a theorem because this wff is an axiom of *Stars*.
2. $*\diamond***\circ****$ is a theorem by the application of the inference rule R of *Stars* to its axiom.
3. $*\diamond****\circ*****$ is a theorem by application of R to the previous wff.

□

When we want to be precise about the line of reasoning that we are using, we will format proofs as a numbered sequence of wffs, together with an indication of the justification for writing each wff down. The justification consists of the name of the axiom, or the name of the rule and the lines on which it depends.

Example 3.8. The inference rule for *Stars* would be represented as

$$\frac{\text{a.}\quad m\diamond n\circ r}{m\diamond n*\circ r*}\ \text{R, a}$$

and a proof of $*\diamond****\circ*****$ would be written:

[1] Formally, we should indicate that the proof was carried out in the system F, for example, by writing $\vdash_F W$. However, usually the context makes it clear which formal system we are reasoning with.

1.	$*\diamond*\circ**$	axiom A
2.	$*\diamond***\circ***$	R, 1
3.	$*\diamond*****\circ*****$	R, 2

□

To make it easier to know what lines a rule should reference, we will often write our inference rules using labels for the wffs above the line, and an indication of how those labels follow the use of the rule.

Example 3.9. Using labeling, the inference rule for *Stars* would be represented as

$$\frac{\text{a.}\quad m\diamond n\circ r}{m\diamond n*\circ r*}\ \text{R, a}$$

□

The collection of all theorems for a formal system F is called the *theory* of F. For example, for the formal system of sets (Chapter 6) the set of theorems is called "set theory." Sometimes a formal system is given the name *calculus*. For example, Chapter 4 discusses the Propositional Calculus and Chapter 5 the Predicate Calculus.

The example system, *Stars*, is extremely simple, and it is relatively easy to determine whether a given wff is in its theory (i.e., can be proved), and if so, how to prove it. More typically, however, the formal systems that we will be using in this book will have several axioms and many inference rules. As we will see later, deciding whether a given wff is a theorem or not in such a system becomes a non-trivial task, often involving ingenuity and creativity.

It is important to emphasize again that when we prove things in a formal system we we are manipulating wffs in purely syntactic ways, appealing only to their linguistic structure and not any interpretations of them.

However, it is interesting to see what happens when we give our formal system an interpretation in a domain in which each wff denotes a statement that is either true or false. In that case it is reasonable to ask whether all theorems that we can prove in the formal system are true. Conversely, we might wonder whether every true statement in the domain of interpretation has some proof. If the first condition holds we say that the formal system is *sound* with respect to that interpretation. If the second condition holds we say that the formal system is *complete* with respect to the interpretation.

Note that in order for a system to be sound the axioms must be true when interpreted in the semantic domain, since axioms are themselves trivially theorems

in the formal system. Furthermore, if a set of wffs is true, then any immediate consequence of the those wffs should also be true.

For example, under the interpretation given earlier for *Stars* the formal system that we defined is sound. As we noted earlier, the axiom represents a true statement, and the consequences of the inference rule will be true if the wff that it is applied to is also true.

How about statements like $2 + 1 = 3$? These are true in the semantic domain, and we might hope that there would be proofs of them in the formal system. Can we prove them? A little thought indicates that we can't. Since the axiom permits only a single star in the first place, and the inference rule never increases the number of stars in that place, there is no way that we can produce as a theorem a wff containing more than one star in the first place. Hence *Stars* is not complete for the interpretation that we gave it.

As we will see in later chapters, this situation is often the case: most formal systems of interest are sound, but not complete. It turns out that incompleteness is a consequence of the fundamental nature of formal systems. Moreover, from a practical perspective, lack of completeness is entirely reasonable. Naturally, we want it to be the case that all theorems in our formal system are true. Otherwise there would be little point in proving them. But often in order to simplify our reasoning and reduce the cost of developing models, our formal systems will be partial. That is, they will attempt to express only some aspects of a semantic domain of interest. Hence, by choice we will be leaving out many of the details of a formal system that would be needed to prove a broader class of theorems.

For some formal systems it is possible to formally prove that the system is sound and/or complete. Such theorems are instances of what logicians refer to as *meta-theorem*. This is because they are theorems about the theorems of a formal system – they tell us something about the kinds of things that we can prove, and the ways we can prove them in that system.

3.5 Derivations

When we prove a theorem, we are arguing from "first principles" – namely the axioms of the formal system. In many cases, what we would like to do, however, is to reason about some consequence under a set of assumptions.

As a simple example, consider the problem of modeling a certain kind of medical patient monitoring device. It might be possible to characterize such devices building up from a set of axioms about the fundamental nature of these devices –

the physics of the sensors, the electrical properties of the circuits and processor, etc. Such a set of axioms would likely be quite complex and require considerable effort to develop.

Another way, however, is to make a set of assumptions about these devices, such as that certain primitive actions will have certain effects, that sensors have certain behavioral characteristics, etc. We would then like to reason about the properties of the device under the condition that those assumptions hold. (Of course, if we are wrong about our assumptions, our conclusions will have little value.)

To enable such an approach, we augment our notion of proof to allow the introduction of a set of assumptions. These assumptions can be treated as if they were additional axioms of the formal system, introduced temporarily for the purpose of a particular proof.

A *derivation* of a wff W in formal system F from a set P of wffs, called *premises*, is a sequence of wffs in the language of F, in which W is the last wff, and where each wff in the sequence is either

- an axiom of F; or
- a premise in P; or
- an immediate consequence of the previous wffs using one of the inference rules of F.

We say that W is *derived from* P, and write $P \vdash W$. When formatting the proof, we indicate the use of a premise by noting that fact in the justification column. We use shorthand $P \dashv\vdash W$ to indicate that both $P \vdash W$ and $W \vdash P$.

Example 3.10. For *Stars* prove the following: $**\diamond*\circ***\vdash **\diamond*****\circ*****$
Proof

1.	$**\diamond*\circ***$	premise
2.	$**\diamond*****\circ*****$	Rule R, 1
3.	$**\diamond**********\circ**********$	Rule R, 2

Relative to the earlier interpretation of *Stars*, we have shown that if assume that $2+1=3$, then we can prove that $2+3=5$. □

Example 3.11. For *Stars* prove the following: $**\diamond*\circ**\vdash **\diamond*****\circ*****$
Proof

1.	$**\diamond*\circ**$	premise
2.	$**\diamond*****\circ*****$	Rule R, 1
3.	$**\diamond**********\circ**********$	Rule R, 2

□

The example above illustrates an interesting point: we can prove nonsense if our premises are not valid. In this case we showed that by assuming that $2 + 1 = 2$ we can prove that $2 + 3 = 4$. This is to be expected: since our formal derivation machinery is independent of any interpretation, it can't be expected to discriminate between premises that are true and those that are false. In fact, there may well exist some other interpretation for which those premises are true.

One final comment: there is, of course, a close relationship between proofs and derivations. In particular, every proof is a derivation in which the set of premises is empty. And, conversely, every derivation can be thought of as a proof in a "richer" formal system in which the premises have been added to the set of axioms.

Chapter Notes

The *Decimals*, *StarDiamond*, and *Stars* formal systems were adapted from the book *Software Engineering Mathematics* by J. Woodcock and M. Loomes [6].

We have shown one possible way of structuring proofs and derivations. Other ways of strucuring derivations exist. For example, in the Gentzen style of proofs [3] proofs are given in a tree format with the root of the tree being the wff to be proved. The application of an inference rule generates branching of the tree in the following way: the root is matched to the conclusion of the rule and as many branches as there are antecedents in the rule are created. Each of these branches in turn is "expanded" (by applying suitable inference rules) causing the proof tree to grow. The details of the rule applications, especially rules that make use of assumptions, are not important at this point. The proof is considered complete when all the leaves are premises, theorems of the system, or assumptions introduced by the inference rules, and all the subproofs are complete.

Further Reading

[TBD]

3.6 Exercises

1. Two formal languages can have the same alphabet but different syntactic rules. Consider the language of section numbers, whose alphabet is the

same as Example 3.1, namely $\{0,1,2,3,4,5,6,7,8,9,.\}$. Wffs in this grammar include 3.4.5, 2, and 0.17. Write the syntactic rules for this language.

2. For the grammar of Example 3.1 it turns out that expressions like 000 and 0.0 are wffs. Write a version of the grammar that does not permit these kinds of expressions to be wffs.

3. Consider the following simple language of expressions with alphabet $\{0,1,2,3,4,5,6,7,8,9,+,=\}$. A wff in this language obeys the following rules:

   - It contains exactly one equality symbol.
   - It begins with a number – a sequence of one or more digits – and ends with a number.
   - A plus symbol may appear between two numbers, but not next to another plus symbol or the equality symbol.

   Write the syntactic rules of the language.

4. The inference system of *Stars* in Section 3.3 is powerful enough to prove theorems like $\vdash **\diamond**********\circ**********$. However, it is impossible to prove that wffs such as $*****\diamond*\circ******$ are theorems. There are no rules that allow adding stars before $\diamond$.

   (a) Extend the inference system for *Stars* so that wffs like $****\diamond*\circ****$ $**$ can be proved to be theorems.
   (b) Illustrate the power of the extended system by providing a proof for the following theorems:
      i. $\vdash *****\diamond*\circ******$
      ii. $\vdash ****\diamond**\circ******$

5. **Smiiileys**

   We extend the grammar of Example 3.3 so that a smiley's nose can be arbitrarily long:

$$
\begin{array}{ll}
\mathit{smiiiley} = & \mathit{eyes}, \mathit{mouth} \mid \mathit{eyes}, \mathit{nose}, \mathit{mouth} \\
\mathit{eyes} & = \text{“:”} \mid \text{“;”} \\
\mathit{nose} & = \text{“}-\text{”} \mid \text{“}-\text{”}, \mathit{nose} \\
\mathit{mouth} & = \text{“(”} \mid \text{“)”}
\end{array}
$$

We interpret *smiiileys* using {happy, sad, flirty, weeping} as follows:

$$
\begin{array}{ll}
:) & \rightarrow \text{ happy} \\
: n) & \rightarrow \text{ happy} \\
:(& \rightarrow \text{ sad} \\
: n(& \rightarrow \text{ sad} \\
;) & \rightarrow \text{ flirty} \\
; n) & \rightarrow \text{ flirty} \\
;(& \rightarrow \text{ weeping} \\
; n(& \rightarrow \text{ weeping}
\end{array}
$$

where n stands for a *nose* of any length.

(a) Formalize the following axioms and inference rules of *Smiiileys*.

A. A sad smiiiley without a nose is a theorem.

B. A smiiiley can be transformed to produce a smiiiley of the same emotion but with its nose extended by one unit; when the smiiiley has no nose to start with, it can transformed to one with a nose of one unit.

C. A sad smiiiley with a nose can be transformed to a happy smiiiley with a nose of equal length.

D. A happy smiiiley with a nose can be transformed to a flirty smiiiley with a nose of equal length.

(b) Provide derivations for the following

i. $\vdash ; ----)$

ii. $\vdash : -)$

iii. $; (\vdash ; -----($

(c) Weeping smiiileys are not theorems of *Smiiileys*. Provide an extension of the inference system of *Smiiileys* that would allow you to prove that weeping smiiileys.

(d) Prove that $; ----($ is a theorem in the new system.

(e) Are there are alternative ways to extend the original system to achieve the same effect as the system you proposed above?

6. **The formal language of Horn formulas**

The formal language of Horn formulas has alphabet $\{\bot, \top, p, q, r, s, \ldots\}$ and

syntax:

$$
\begin{array}{ll}
\mathit{hornFormula} & = \mathit{hornClause} \mid \mathit{hornClause}, \text{“}\wedge\text{”}, \mathit{hornFormula} \\
\mathit{hornClause} & = \text{“(”}, \mathit{assumption}, \text{“}\Rightarrow\text{”}, \mathit{atom}, \text{“)”} \\
\mathit{assumption} & = \mathit{atom} \mid \mathit{atom}, \text{“}\wedge\text{”}, \mathit{assumption} \\
\mathit{atom} & = \text{“}\bot\text{”} \mid \text{“}\top\text{”} \mid \text{“}p\text{”} \mid \text{“}q\text{”} \mid \text{“}r\text{”} \mid \text{“}s\text{”} \mid \ldots
\end{array}
$$

Which of the following are wffs in this formal language?

(a) $(p \wedge q \wedge s \Rightarrow p) \wedge (q \wedge r \Rightarrow p) \wedge (p \wedge s \Rightarrow s)$
(b) $(p \wedge q \wedge s \Rightarrow \neg\, p) \wedge (q \wedge r \Rightarrow p) \wedge (p \wedge s \Rightarrow s)$
(c) $(p \wedge q \wedge s \Rightarrow \bot) \wedge (q \wedge r \Rightarrow p) \wedge (\top \Rightarrow s)$
(d) $(p \wedge q \wedge s \Rightarrow \bot) \wedge (\neg\, q \wedge r \Rightarrow p) \wedge (\top \Rightarrow s)$

7. **A formal language for sequences of events**

A formal language to express sequences of events is defined as follows:

Alphabet: $\{a, \ldots, z, A, \ldots, Z, \text{Ⓢ}, =, \rightarrow\}$ where lower case letters represent events, and upper case letters represent named processes. A named process represents a sequence of events that can be referred to in other sequences of events. The Ⓢ symbol is a special process that represents the fact that no events happen (empty sequence of events).

Syntax: Wffs of this language are called *models*. A model consists of a top level process definition, optionally followed by auxiliary process definitions, as specified by the following grammar:

$$
\begin{array}{ll}
\mathit{model} & = \mathit{processDefinition} \mid \mathit{processDefinition}, \text{“;”}, \mathit{model} \\
\mathit{processDefinition} & = \mathit{processName}, \text{“}=\text{”}, \mathit{sequentialProcess} \\
\mathit{sequentialProcess} & = \mathit{processName} \mid \mathit{eventName}, \text{“}\rightarrow\text{”}, \mathit{sequentialProcess} \\
\mathit{eventName} & = \text{“}a\text{”} \mid \ldots \mid \text{“}z\text{”} \\
\mathit{processName} & = \text{“}A\text{”} \mid \ldots \mid \text{“}Z\text{”} \mid \text{“Ⓢ”}
\end{array}
$$

(a) Which of the following are wffs in this formal language:
   i. $A = a \rightarrow b \rightarrow c$
   ii. $B = \text{Ⓢ}$
   iii. $C = a \rightarrow b$; $D = \text{Ⓢ}$
   iv. $D = a \rightarrow z \rightarrow b \rightarrow w \rightarrow D$
   v. $E = a \rightarrow z \rightarrow b \rightarrow w \rightarrow \text{Ⓢ}$
   vi. $a \rightarrow z \rightarrow b \rightarrow w \rightarrow \text{Ⓢ}$

(b) The grammar above allows $\textcircled{S} = C$ to be a wff. Modify the grammar to forbid the case that $\textcircled{S}$ can appear on the left side of a process definition.

(c) The grammar above allows a process to be defined more than twice in a model. For example, $F = a \rightarrow b$; $F = c \rightarrow d$ is a wff. Can you devise a grammar that does not allow for a process names to appear more than once on the left-hand side of $=$?

8. **Internal combustion engine**

The operation of an internal combustion engine that uses the four-stroke combustion cycle can be modeled by a formal language whose wffs correspond to all the possible sequences of events in the operation of the engine. The four-stroke cycle is a sequential process that consists of four stages: *Intake stroke*, *Compression stroke*, *Combustion stroke*, and *Exhaust stroke*.

(a) Design a formal language (alphabet + grammar) to model the cyclic operation of a four-stroke internal combustion engine. (Hint: assume that the transitions between stages in the four-stroke cycle are the events of interest and represent them as symbols in the alphabet of your language.)

(b) Modify your formal language so that it can model the behavior of an engine that runs out of gas. (Hint: assume that when the engine runs out of gas it stops immediately before the combustion stroke since there is no mix for the combustion to take place.)

9. **A formal system for natural numbers**

A formal system to reason about natural numbers can be defined as follows:

Formal Language:

Alphabet: $\{zero, add, s, (,)\}$

Syntax:

$$\begin{aligned} NatNum = {} & \text{“}zero\text{”} \\ & \mid \text{“}s\text{”}, \text{“(”}, NatNum, \text{“)”} \\ & \mid \text{“}add\text{”}, \text{“(”}, NatNum, \text{“,”}, NatNum, \text{“)”} \end{aligned}$$

Inference System:

Axioms:

$$zero$$

Inference Rules:

$$\frac{add(zero, x)}{x}\ \text{rule1}$$

$$\frac{add(s(x), y)}{add(x, s(y))}\ \text{rule2}$$

(a) Is the following wff a theorem in this formal system?

$$add(s(s(s(zero))), s(s(zero)))$$

(b) Construct a derivation to prove

$$add(s(s(s(zero))), s(s(zero))) \vdash s(s(s(s(s(zero)))))$$

Chapter 4

Propositional Logic

We now introduce a formal system developed for expressing a particular kind of statement called a *proposition*, and for determining the truth and falsity of propositions. We start by discussing what propositions are, introduce a formal language called *propositional logic*, provide an interpretation of the well-formed formulae of propositional logic as truth values of propositions, and present an inference system for propositional logic called a *propositional calculus*.

4.1 Propositions

The starting point for any formal modeling enterprise is the ability to make statements about some domain of interest, and to reason about the truth of those statements.

Propositions are statements that are either true or false (but not both).

Example 4.1. The following are propositions:

- *"Boston is the home of the Red Sox baseball team."*
- *"Pittsburgh is the capital of Pennsylvania."*
- *"Some mammals lay eggs."*
- "$2+3=6$"

□

We will avoid statements whose truth or falsity depends on the context. For example, statements such as *"February has 28 days,"* or *"my brother is younger than I,"* express different propositions depending on whether the February in question

is from a leap year or not, and who makes the second statement. Other sentences that are not propositions include interrogative, imperative, and exclamatory sentences. Such sentences cannot be said to be true or false.

Example 4.2. The following are not propositions:

- *"What a day!"*
- *"Are there any questions so far?"*
- *"Today is Sunday."*
- *"Pass the salt, please."*

□

The same proposition can often be expressed in several ways.

Example 4.3. The following express the same proposition:

- *"Five is bigger than four."*
- *"Four is smaller than five."*
- "$4 < 5$"
- "$5 > 4$"

□

Some propositions have structure. For example, the proposition

"David went to the store and Bill went to the movies."

can be decomposed into two simpler propositions:

"'David went to the store' and *'Bill went to the movies'."*

Similarly, any two propositions can be combined using words such as "and", "or", "if ... then ...", etc. to form new propositions.

4.2 Syntax

To create a formal system for statements such as those in the previous section, we first need to define the grammar of the language that we will use to represent propositions.

The alphabet of propositional logic consists of the following symbols:

$$p, q, r, ..., p_1, q_1, r_1, \ldots, \neg, \vee, \wedge, \Rightarrow, \Leftrightarrow, (,)$$

We use lower case letters to denote *primitive* or *atomic* propositions, and assume that we never run out of such symbols. These letters will act as shorthand for statements like *"David went to the store."*

The symbols $\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow$ are used to combine other propositions to form *compound* propositions and are known as *propositional connectives*.

The well-formed formulae of propositional logic are determined by the following grammar:

$$\begin{aligned} \mathit{sentence} = {} & \text{“}p\text{”} \mid \text{“}q\text{”} \mid \text{“}r\text{”} \mid \ldots \mid \text{“}p_1\text{”} \mid \text{“}q_1\text{”} \mid \text{“}r_1\text{”} \mid \ldots \\ & \mid \text{“}\neg\text{”}, \mathit{sentence} \\ & \mid \text{“(”}, \mathit{sentence}, \text{“}\vee\text{”}, \mathit{sentence}, \text{“)”} \\ & \mid \text{“(”}, \mathit{sentence}, \text{“}\wedge\text{”}, \mathit{sentence}, \text{“)”} \\ & \mid \text{“(”}, \mathit{sentence}, \text{“}\Rightarrow\text{”}, \mathit{sentence}, \text{“)”} \\ & \mid \text{“(”}, \mathit{sentence}, \text{“}\Leftrightarrow\text{”}, \mathit{sentence}, \text{“)”} \end{aligned}$$

Example 4.4. The following are sentences in propositional logic:

- $((p \vee q) \wedge \neg(r \Rightarrow \neg q))$
- $\neg\neg\neg p$
- $((p \vee q) \vee r)$

□

We also introduce a few special terms to refer to specific forms of propositions. Let p and q be arbitrary sentences in propositional logic. A sentence of the form

$\neg p$ is called a *negation*,
$p \vee q$ is called a *disjunction*; p and q are *disjuncts*,
$p \wedge q$ is called a *conjunction*; p and q are *conjuncts*,
$p \Rightarrow q$ is called an *implication*; p is the *antecedent*, and q is the *consequent* or *conclusion*,
$p \Leftrightarrow q$ is called a *bi-implication*.

Precedence To increase sentence readability we may drop the parentheses when the resulting sentences are unambiguous; *precedence rules* are used to determine how such sentences are parsed. The precedence order (from highest to lowest) for propositional connectives are:

$$\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$$

Moreover, implication $\Rightarrow$ is *right-associative*: A sentence of the form $p \Rightarrow q \Rightarrow r$ is read as $p \Rightarrow (q \Rightarrow r)$.

Example 4.5. We may drop the parentheses in the following sentences:

1. $(((q \wedge \neg r) \vee p) \Rightarrow \neg q)$ is the same as $q \wedge \neg r \vee p \Rightarrow \neg q$.
2. $((p \wedge q) \Rightarrow r)$ is the same as $p \wedge q \Rightarrow r$.

But we may not drop any parentheses in the following sentences:

3. $p \vee \neg(q \wedge r)$ is *not* the same as $p \vee \neg q \wedge r$.
4. $((\neg r \Leftrightarrow p) \vee \neg q) \wedge r$ is *not* the same as $\neg r \Leftrightarrow p \vee \neg q \wedge r$.
5. $(p \Rightarrow q) \Rightarrow r \Rightarrow s$ is *not* the same as $p \Rightarrow q \Rightarrow r \Rightarrow s$.

□

In practice it is often a good idea to use parentheses to clarify the intended meaning even when they are not strictly necessary. For example, sentence 1 above is more readable if we write it as $(q \wedge \neg r \vee p) \Rightarrow \neg q$. Similarly, it is often a good idea to include parentheses when $\wedge$ and $\vee$ are used next to each other. For example, we would write $(q \wedge \neg r) \vee p$ even though the parentheses are not strictly required.

4.3 Semantics

To define the meaning of a propositional logic, we need to pick a domain of interest, and explain how every propositional sentence is mapped into that domain.

When interpreting propositions the domain of interest is simply that of truth values: *true* (T) and *false* (F). The meaning of propositional sentences is then determined as follows:

1. Each symbol denoting an atomic proposition is interpreted as the truth value associated with that proposition.
2. The meaning of a compound proposition is determined by the meaning of its subparts as specified by a set of rules defined using standard *truth tables* shown in Figure 4.1.

The truth tables of Figure 4.1 encode the following informal rules:

- $\neg p$ is true if and only if p is false,
- $p \vee q$ is true when either or both of p or q are true,
- $p \wedge q$ is true only if both p and q are true,
- $p \Rightarrow q$ is false only when p is true and q is false,

p	$\neg p$
T	F
F	T

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

p	q	$p \Rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

p	q	$p \Leftrightarrow q$
T	T	T
T	F	F
F	T	F
F	F	T

p and q denote any arbitrary propositional sentences.

Figure 4.1: Truth tables for propositional sentences

- $p \Leftrightarrow q$ is true if and only if p and q have the same truth value.

There are a number of special cases for which we introduce specific terminology:

– Sentences that are true under all interpretations of atomic propositions are said to be *valid*; we call such sentences *tautologies*.
– Sentences that are true under *at least one* interpretation of atomic propositions are said to be *consistent* or *satisfiable*.
– Sentences that are false under all interpretations of atomic propositions are said to be *inconsistent*, or *unsatisfiable*; we call such sentences *contradictions*.
– Sentences that are neither tautologies nor contradictions are said to be *contingent*.

Example 4.6. Let p and q be arbitrary propositional sentences.

- $p \Rightarrow (p \vee q)$ is valid.
- $p \wedge \neg p$ is inconsistent.
- $p \Rightarrow (p \wedge q)$ is contingent.

We can convince ourselves about these facts by constructing the truth tables for the sentences. For example, validity of $p \Rightarrow (p \vee q)$ can be demonstrated by the fact that its truth value is always T, regardless of how p and q are interpreted:

p	q	$p \vee q$	$p \Rightarrow (p \vee q)$
T	T	T	T
T	F	T	T
F	T	T	T
F	F	F	T

p	q	$p \wedge q$	$p \Rightarrow (p \wedge q)$
T	T	T	T
T	F	F	F
F	T	F	T
F	F	F	T

Similarly, the truth table for $p \Rightarrow (p \wedge q)$ reveals that for p true and q false, the sentence is false (hence the sentence is not a tautology); however, the sentence is true for all other interpretations of p and q (hence the sentence is not a contradiction). □

Truth tables work fine for understanding properties of simple propositional formulae. However, as the size of a formula increases truth tables rapidly become impractical. Indeed, the number of rows grows exponentially with the number of propositional symbols in a sentence. To make it possible to reason about complex propositional sentences, we will introduce a way to carry out reasoning about propositions at the syntactic level without having to appeal explicitly to the interpretation of propositions. As we discussed in Chapter 3, an inference system allows us to do just that.

4.4 Propositional Calculus

We now complete our formal system for propositional logic by describing an inference system, which is typically called a *propositional calculus*. As with all inference systems, a propositional calculus consists of a set of rules that allow us to derive propositional sentences from a set of given propositional sentences.

The inference rules come in two forms called *elimination rules* and *introduction rules*. For a propositional connective *op*, its elimination rules describe what may be deduced from a sentence of the form *p op q*. The introduction rules for *op* describe under what conditions we can conclude *p op q*. Note that whenever p, q, r appear in inference rules they stand for arbitrary propositional sentences.

4.4.1 Conjunction

- Conjunction introduction:

  a. p
  b. q
  ——————
  $p \wedge q$ $\wedge$-intro, a,b

  a. p
  b. q
  ——————
  $q \wedge p$ $\wedge$-intro, a,b

- Conjunction elimination:

  a. $p \wedge q$
  ——————
  p $\wedge$-elim, a

  a. $p \wedge q$
  ——————
  q $\wedge$-elim, a

Conjunction introduction says that in order to derive $p \wedge q$ it is sufficient to

derive p and q separately. Conjunction elimination says that having derived $p \wedge q$ we can derive p, and q separately.

Example 4.7. **$\wedge$-Commutativity** [1]
We show that $p \wedge q \vdash q \wedge p$.

1.	$p \wedge q$	premise
2.	q	$\wedge$-elim, 1
3.	p	$\wedge$-elim, 1
4.	$q \wedge p$	$\wedge$-intro, 2,3

Commutativity of $\wedge$ means that changing the order of the conjuncts in a conjunction does not change the meaning of the sentence. Disjunction $\vee$, and bi-implication $\Leftrightarrow$ are also commutative. □

4.4.2 Implication

- Implication introduction:

a.	p	assumption
	⋮	
c.	q	
	$p \Rightarrow q$	$\Rightarrow$-intro, a–c

- Implication elimination:

a.	$p \Rightarrow q$	
b.	p	
	q	$\Rightarrow$-elim, a,b

Implication introduction encodes the intuition that if by assuming p we can show q, then we must have $p \Rightarrow q$. This follows from our understanding of logical implication: that if q is true whenever p is true, then $p \Rightarrow q$ is true. To represent this idea the rule for implication introduction makes use of an *assumption* in the derivation. An assumption allows us to introduce an arbitrary sentence that we can treat temporarily just like any other derived sentence. At some point in the proof, that assumption is *discharged* by using it in some inference rule. The part of the proof within which that assumption can be used is called its *scope* (marked by a vertical line). The scope starts at the line in which the assumption is introduced and ends *before* the line of the rule used to discharge the assumption. An

[1] We label some useful results for easy reference. In Section 4.5 we show how such derived results can be used as inference rules.

assumption, and any statements derived within the scope of that assumption *must not* be used outside that assumption's scope.

For historical reasons, implication elimination sometimes goes by the name *modus ponens*.

Example 4.8. We show that $p \Rightarrow (q \Rightarrow r) \vdash p \wedge q \Rightarrow r$.

1.	$p \Rightarrow (q \Rightarrow r)$	premise
2.	$p \wedge q$	assumption
3.	p	$\wedge$-elim, 2
4.	$q \Rightarrow r$	$\Rightarrow$-elim, 1,3
5.	q	$\wedge$-elim, 2
6.	r	$\Rightarrow$-elim, 4,5
7.	$p \wedge q \Rightarrow r$	$\Rightarrow$-intro, 2–6

While in general discovering a legal derivation requires creativity and insight into the problem, in the proof above (and in many proofs) the structure of the sentence that we want to derive helps determine the structure of the derivation. In this case, since $\Rightarrow$ is the outermost connective of $p \wedge q \Rightarrow r$ (remember that $\wedge$ binds tighter than $\Rightarrow$), we try to match the sentence to rules whose conclusion has $\Rightarrow$. This suggests $\Rightarrow$-intro as a potential rule. This in turn causes us to introduce line 2 (and the scope of the assumption). Now we need to derive r under the assumption $p \wedge q$. At this point we may do one of two things: either try to derive as many sentences from the assumption as we can, or look at the premise for clues about what may be useful next. Since the premise has $\Rightarrow$ as its outermost connective, we try to match that to the *premises* part of the inference rules, and notice that we could use $\Rightarrow$-elim if we could derive p. Moreover, we discover that p is easily derivable from $p \wedge q$ using $\wedge$-elim, so we write down line 3. Now line 4 follows from $\Rightarrow$-elim using lines 1 and 3. The rest of the derivation is relatively straightforward. □

4.4.3 Bi-implication

- Bi-implication introduction:

$$\frac{\begin{array}{ll}\text{a.} & p \Rightarrow q \\ \text{b.} & q \Rightarrow p\end{array}}{p \Leftrightarrow q}\ \Leftrightarrow\text{-intro, a,b} \qquad \frac{\begin{array}{ll}\text{a.} & p \Rightarrow q \\ \text{b.} & q \Rightarrow p\end{array}}{q \Leftrightarrow p}\ \Leftrightarrow\text{-intro, a,b}$$

- Bi-implication elimination:

$$\frac{\begin{array}{ll}\text{a.} & p \Leftrightarrow q\end{array}}{p \Rightarrow q}\ \Leftrightarrow\text{-elim, a} \qquad \frac{\begin{array}{ll}\text{a.} & p \Leftrightarrow q\end{array}}{q \Rightarrow p}\ \Leftrightarrow\text{-elim, a}$$

Bi-implication introduction encodes the intuition that to prove $p \Leftrightarrow q$ we need to prove both $p \Rightarrow q$ and $q \Rightarrow p$. Bi-implication elimination encodes the intuition that if we have shown $p \Leftrightarrow q$, then we can directly conclude $p \Rightarrow q$ and $q \Rightarrow p$.

Example 4.9. **$\wedge$-Commutativity alternative**
We show that $\vdash p \wedge q \Leftrightarrow q \wedge p$.

1.	$p \wedge q$	assumption
2.	q	$\wedge$-elim, 1
3.	p	$\wedge$-elim, 1
4.	$q \wedge p$	$\wedge$-intro, 2,3
5.	$p \wedge q \Rightarrow q \wedge p$	$\Rightarrow$-intro, 1–4
6.	$q \wedge p$	assumption
7.	p	$\wedge$-elim, 6
8.	q	$\wedge$-elim, 6
9.	$p \wedge q$	$\wedge$-intro, 7,8
10.	$q \wedge p \Rightarrow p \wedge q$	$\Rightarrow$-intro, 6-9
11.	$p \wedge q \Leftrightarrow q \wedge p$	$\Leftrightarrow$-intro, 5,10

As before, the structure of the sentence that we want to prove gives us a clue about the structure of the proof. In particular, since $\Leftrightarrow$ is the outermost connective, we would expect to use $\Leftrightarrow$-intro, and are therefore led to two sub-proofs of $p \wedge q \Rightarrow q \wedge p$ (line 5) and $q \wedge p \Rightarrow p \wedge q$ (line 10). $\square$

4.4.4 Disjunction

- Disjunction introduction:

$$\frac{\text{a. } p}{p \vee q}\ \vee\text{-intro, a} \qquad\qquad \frac{\text{a. } q}{p \vee q}\ \vee\text{-intro, a}$$

- Disjunction elimination:

a.	$p \vee q$	
b.	p	assumption
	$\vdots$	
d.	r	
e.	q	assumption
	$\vdots$	
g.	r	
	r	$\vee$-elim, a,b–d,e–g

Disjunction introduction says that in order to derive $p \vee q$ it is sufficient to derive one of the disjuncts.

The disjunction elimination rule justifies the reasoning strategy known as "case analysis." Consider the following informal example: We would like to prove a property about all integers. We could do this by showing a proof of the property for an arbitrary negative number, a proof for zero, and a proof for an arbitrary positive number. Having exhausted all the possibilities (an integer is either positive, negative, or zero) we have a proof of the property in question for all integers. Similarly, in order for a derivation to be possible under a disjunction, it is sufficient to have a derivation under each disjunct.

Example 4.10. **$\vee$-Associativity**
Associativity of $\vee$ can be expressed as $p \vee (q \vee r) \dashv\vdash (p \vee q) \vee r$ — it expresses that the meaning of disjunctions of three or more propositions does not depend on how the disjuncts are grouped together. Conjunction $\wedge$, and bi-implication $\Leftrightarrow$ are also associative.

We show that $p \vee (q \vee r) \vdash (p \vee q) \vee r$; the other direction can be shown similarly.

1.	$p \vee (q \vee r)$	premise
2.	p	assumption
3.	$p \vee q$	$\vee$-intro, 2
4.	$(p \vee q) \vee r$	$\vee$-intro, 3
5.	$q \vee r$	assumption
6.	q	assumption
7.	$p \vee q$	$\vee$-intro, 6
8.	$(p \vee q) \vee r$	$\vee$-intro, 7
9.	r	assumption
10.	$(p \vee q) \vee r$	$\vee$-intro, 9
11.	$(p \vee q) \vee r$	$\vee$-elim, 5,6–8,9–10
12.	$(p \vee q) \vee r$	$\vee$-elim, 1,2–4,5–11

We briefly describe the structure of the derivation. To derive $(p \vee q) \vee r$ from $p \vee (q \vee r)$ we use the disjunction elimination rule as follows: we try to derive $(p \vee q) \vee r$ from p and $q \vee r$ separately. The first case is simple: if we assume that p is derivable, then we can apply the (first) disjunction introduction rule to derive $p \vee q$, and then apply the rule once more (in the rule we substitute $p \vee q$

for the schema variable p and r for the schema variable q). Deriving $(p \vee q) \vee r$ from $q \vee r$ requires applying the disjunction elimination rule once more: we derive $(p \vee q) \vee r$ from q and r separately. This requires two applications of disjunction introduction to derive the sentence from q, and one application of disjunction introduction to derive it from r. Thus, we arrive at derivation line 11. Now, having derived the conclusion from both p and $q \vee r$ we finish the derivation. □

4.4.5 Negation

- Negation introduction:

a.	p	assumption
	⋮	
c.	q	
	⋮	
e.	$\neg q$	
	$\neg p$	$\neg$-intro, a,c,e

- Negation elimination:

a.	$\neg p$	assumption
	⋮	
c.	q	
	⋮	
e.	$\neg q$	
	p	$\neg$-elim, a,c,e

The negation rules justify "proofs by contradiction." In order to derive $\neg p$ we show that assuming otherwise (that is, assuming that p is derivable) leads to a contradiction (that is, the derivation of both q and $\neg q$, where q may be any sentence in propositional logic). Similarly, any sentence p can be proved by showing that its negation leads to a contradiction.

Example 4.11. We show that $\vdash p \wedge \neg p \Rightarrow q$.

1.	$p \wedge \neg p$	assumption
2.	$\neg q$	assumption
3.	p	$\wedge$-elim, 1
4.	$\neg p$	$\wedge$-elim, 1
5.	q	$\neg$-elim, 2,3,4
6.	$p \wedge \neg p \Rightarrow q$	$\Rightarrow$-intro, 1–5

□

Although the negation rules are intuitive, figuring out what contradiction to derive (that is, how to instantiate q in the rules) can be challenging in practice.

Example 4.12. **Double negation**
We show that $\neg\neg p \dashv\vdash p$.

First we show $\neg\neg p \vdash p$.

1.	$\neg\neg p$	premise
2.	$\neg p$	assumption
3.	$\neg p$	copy from 2
4.	$\neg\neg p$	copy from 1
5.	p	$\neg$-elim, 2,3,4

We instantiated q with $\neg p$ in the negation elimination rule. Therefore $\neg q$ becomes $\neg\neg p$.

Now we show $p \vdash \neg\neg p$.

1.	p	premise
2.	$\neg p$	assumption
3.	p	copy from 1.
4.	$\neg p$	copy from 2.
5.	$\neg\neg p$	$\neg$-intro, 2,3,4

We instantiated q with p in the negation elimination rule.

The proofs use "copy from" as justification for several of the derivation lines. This justification simply indicates that we are using a sentence derived earlier in the proof. Although it is not strictly necessary to copy the sentence in the context it is being used doing so makes the proof more readable. But when is it valid to use a previously derived sentence at some point in a proof? We can use a sentence r at a derivation line k in the proof if r occurs in a derivation line i prior to k *and* no assumption scope that encloses the derivation line i has been closed already. □

A special case of a proof by contradiction arises when the premises are *contradictory*: premises are contradictory if any of the individual premises or the conjunction of two or more premises is a contradiction. When this occurs, *any* sentence q (as well as $\neg q$) can be derived from those premises. (See Section 4.6 for a more formal justification.) This situation is the logical counterpart of "garbage in; garbage out."

For example, a proof of $p, \neg p \vdash q$ would go like this:

1.	$\neg q$	assumption	|
2.	p	premise	|
3.	$\neg p$	premise	|
4.	q	$\neg$-intro, 1,2,3	

Notice that the assumption ($\neg q$) played no role in the derivation.

4.5 Derived Inference Rules

Once a theorem is proved we may use it to derive other results; every theorem of a formal system becomes a new inference rule in the system. In fact, it is the ability to build on previously proved results that makes the task of formal reasoning about software systems possible in practice: as we introduce more and more machinery in the book we will be able to prove complex properties about systems using derived properties and rules of inference, without having to explicitly break down reasoning to the underlying primitive inference rules.

What justifies using proved theorems as inference rules? Why, for example, can we use $\vdash p \wedge q \Leftrightarrow q \wedge p$ ($\wedge$-Commutativity alternative, Example 4.9) to derive $\neg t \wedge \neg r \Leftrightarrow \neg r \wedge \neg t$? The justification is that appealing to "$\wedge$-Commutativity alternative" stands for appealing to the proof of the theorem. In the specific case in which we use $\wedge$-Commutativity alternative to derive $\neg t \wedge \neg r \Leftrightarrow \neg r \wedge \neg t$ we are appealing to a copy of the proof of the theorem in which $\neg t$ is substituted for p and $\neg r$ is substituted for q.

More generally, how can we reuse derivations with premises? To answer this question we first discuss how theorems can be created from derivations with premises.

Deduction Theorem Theorems can be created from derivations with premises thanks to the *deduction theorem*.[2] Informally, the deduction theorem says that if $p \vdash q$ (that is, if q is derivable from p) then $\vdash p \Rightarrow q$ is a theorem in propositional logic.[3] The technique generalizes for derivations with more than one premise:

[2]The deduction theorem is a *meta* theorem — it is a theorem *about* propositional logic.

[3]Proof sketch: assume $p \vdash q$. We would like to show that $\vdash p \Rightarrow q$. Apply implication introduction; this requires deriving q under the assumption p. Since the scope of assumption p is the entire derivation, p can be used in the derivation just like it would be used if it were a premise. But from the assumption $p \vdash q$ we have a derivation of q from p.

given derivation $p_1, p_2, \ldots, p_n \vdash q$, the following is a theorem: $\vdash p_1 \Rightarrow (p_2 \Rightarrow (\ldots \Rightarrow (p_n \Rightarrow q)))$.

Now we discuss how we can reuse derivations with premises. Given a derivation $p \vdash q$ we may introduce the following rule in propositional logic:

$$\frac{\text{a.}\quad p}{q}$$ rule name, a

The justification is simple. From the deduction theorem $\vdash p \Rightarrow q$ is a theorem, and we can always appeal to it in derivations. From implication elimination, to derive q it is sufficient to have a derivation for p, and this is exactly what the newly introduced rule says.

The technique generalizes for derivations with more than one premise. Given a derivation $p_1, p_2, \ldots, p_n \vdash q$ we may introduce the following rule in propositional logic:[4]

$$\frac{\begin{array}{ll}\text{a}_1. & p_1 \\ \text{a}_2. & p_2 \\ & \vdots \\ \text{a}_n. & p_n\end{array}}{q}$$ rule name, a_1,a_2,..., a_n

Example 4.13. In Example 4.12 we proved $\neg\neg p \dashv\vdash p$. We called this rule (which allows us to derive p and $\neg\neg p$ from each other) "Double negation."

We show that $\vdash r \vee \neg r$.

1.	$\neg(r \vee \neg r)$	assumption
2.	r	assumption
3.	$r \vee \neg r$	$\vee$-intro, 2
4.	$\neg(r \vee \neg r)$	copy from 1
5.	$\neg r$	$\neg$-intro, 2,3,4
6.	$r \vee \neg r$	$\vee$-intro, 5
7.	$\neg(r \vee \neg r)$	copy from 1
8.	$\neg\neg(r \vee \neg r)$	$\neg$-intro, 1,6,7
9.	$r \vee \neg r$	Double negation, 8

[4]The justification for the generalization also relies on the following fact:

$$p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q \dashv\vdash p_1 \Rightarrow (p_2 \Rightarrow (\ldots \Rightarrow (p_n \Rightarrow q)))$$

This is an expression of the Shunting rule from Figure 4.2.

Note that we have only needed the $\neg\neg p \vdash p$ side of the rule. $\square$

Figure 4.2 summarizes some useful derived rules of propositional calculus. These include many of the common propositional rules such as DeMorgan's Laws, commutativity and associativity of conjunction, disjunction, and bi-implication, and rules for reasoning with contrapositives.

$\vdash p \vee \neg p$	Excluded Middle
$p \vee q \vdash q \vee p$	$\vee$-Commutativity
$p \wedge q \vdash q \wedge p$	$\wedge$-Commutativity
$p \Leftrightarrow q \vdash q \Leftrightarrow p$	$\Leftrightarrow$-Commutativity
$(p \vee q) \vee r \dashv\vdash p \vee (q \vee r)$	$\vee$-Associativity
$(p \wedge q) \wedge r \dashv\vdash p \wedge (q \wedge r)$	$\wedge$-Associativity
$(p \Leftrightarrow q) \Leftrightarrow r \dashv\vdash p \Leftrightarrow (q \Leftrightarrow r)$	$\Leftrightarrow$-Associativity
$p \vee (q \wedge r) \dashv\vdash (p \vee q) \wedge (p \vee r)$	$\vee\wedge$-Distributivity
$p \wedge (q \vee r) \dashv\vdash (p \wedge q) \vee (p \wedge r)$	$\wedge\vee$-Distributivity
$(p \Rightarrow q) \wedge (q \Rightarrow r) \vdash p \Rightarrow r$	$\Rightarrow$-Transitivity
$(p \Leftrightarrow q) \wedge (q \Leftrightarrow r) \vdash p \Leftrightarrow r$	$\Leftrightarrow$-Transitivity
$p \Rightarrow q \dashv\vdash \neg p \vee q$	$\Rightarrow$-Alternative
$p \Rightarrow q \dashv\vdash \neg q \Rightarrow \neg p$	Contrapositives
$\neg\neg p \dashv\vdash p$	Double Negation
$(p \Leftrightarrow q) \dashv\vdash (\neg p \Leftrightarrow \neg q)$	$\Leftrightarrow$-Alternative
$\neg(p \wedge q) \dashv\vdash \neg p \vee \neg q$	De Morgan
$\neg(p \vee q) \dashv\vdash \neg p \wedge \neg q$	De Morgan
$p \wedge q \Rightarrow r \dashv\vdash p \Rightarrow (q \Rightarrow r)$	Shunting

Figure 4.2: Useful derived rules of propositional calculus

4.6 Soundness and Completeness

An important property is that the propositional calculus that we have presented is sound and complete with respect to the semantics of propositions as truth values.

Soundness means that every theorem of propositional calculus is a valid sentence in propositional logic. That is to say, if $\vdash p$ can be proved using the inference rules of propositional calculus then p is valid – i.e., p is true under all interpretations of the atomic propositions (see Section 4.3). More generally, soundness expresses the fact that if $p \vdash q$ is derivable then every interpretation (of the propositional symbols appearing in p) that makes p true, also makes q true.

Completeness means that every valid sentence in propositional logic is a theorem of propositional calculus. That is to say, if p is a valid sentence of propositional logic, then $\vdash p$ can be proved in propositional calculus. More generally, if every interpretation that makes p true also makes q true, then the derivation $p \vdash q$ is possible. A special degenerate case arises from a contradictory p. In this situation no interpretation makes p true (since p is a contradiction), therefore, the condition for $p \vdash q$ to be possible is said to hold *vacuously*.

4.7 Translating English into Propositional Logic

One of the important skills we are going to need when modeling and reasoning about software systems is the ability to translate system descriptions and requirements from informal English into sentences in formal logic. In many cases this is straightforward once we have picked a suitable collection of symbols to represent basic facts about the domain of interest. Sometimes, however, such translation can be tricky due to the ambiguity of natural language (often manifested in overloaded meanings for certain words), and situations in which context influences the meaning of sentences. In this section we provide guidelines for translating common English constructs into propositional logic and clarify how you can resolve some of the ambiguities.

The process of translation usually involves three steps:

1. Determine which facts about our world to represent as atomic propositions and introduce suitable symbols for them.
2. Replace the parts of English description corresponding to atomic propositions with the introduced symbols.
3. Translate English connective words into the connectives of propositional logic.

4.7.1 Choosing Atomic Propositions

When we approach a translation problem we usually have control over the amount of detail to represent in our models. One important decision is which facts about the world should be treated as atomic propositions. Often there is a tradeoff: If we choose to represent complex real-world situations as atomic propositions we simplify the description. But at the same time we may reduce our ability to reason at a detailed level about the phenomena that we are trying to capture.

Consider, for example, the software to control a medical device. At one extreme we might represent "the system is functioning correctly" as an atomic proposition. On the other hand, we might represent the logical constituents of "correctness" (such as that the power is on, the patient is connected to the machine, etc.) as individual propositions, each of which might be true or false. In this case "functioning correctly" would probably be the conjunction of those other propositions. While the second option is more detailed, and hence more complex, it also allows us to reason about situations where parts of the machine are running correctly, but other parts are not.

Traffic Lights

Let's consider another example.

> *Traffic lights control traffic of two intersecting roads: the North-South direction, and the East-West direction. Each direction has a single light associated with it. A light can be green, yellow, or red. To function properly the traffic lights must obey the constraint that if the light of one direction is green or yellow, the light of the other direction is red.*

A simple description of the traffic lights can be created using the following sentences as atomic propositions:

> *nsl*: "the North-South light is green, yellow, or red"
> *ewl*: "the East-West light is green, yellow, or red"
> *r*: "if the North-South light is green or yellow, the East-West light is red, and if the East-West light is green or yellow, the North-South light is red"

We could then write

$$traffic\_lights_1 == nsl \wedge ewl \wedge r$$

where $==$ is used to represent the fact that $traffic\_lights_1$ is a name for the proposition on the right hand side of $==$. We pick $\wedge$ as the connective since we would like to say that all the mentioned propositions should simultaneously hold in the world of traffic lights.

This description is simple, but not particularly useful. What can we say about the truth or falsity of proposition "if the North-South light is green or yellow, the

East-West light is red"? Not much. However, by revealing more of the structure of r we can do better.

We might start by introducing r_1 and r_2 as atomic propositions for the two parts of r:

r_1: "if the North-South light is green or yellow, the East-West light is red"
r_2: "if the East-West light is green or yellow, the North-South light is red"

We can then characterize the collection of facts about our traffic lights as:

$$traffic\_lights_2 == nsl \wedge ewl \wedge r_1 \wedge r_2$$

$traffic\_lights_2$ is more detailed, and the proposition "if the North-South light is green or yellow, the East-West light is red" is one of its primitive facts. But many of the questions that might arise would remain unanswered in this model. For example, what can we say about the East-West light in the situation that "the North-South light is green"? It surely must be red, but we cannot reason about this given our current representation of traffic lights. We will return to this example with a more expressive representation after discussing how English connective words are translated into logic connectives.

4.7.2 Connectives

The process of translating English connective words into logic connectives is often straightforward. Figure 4.3 summarizes how some of the most frequent English connective words are translated.

In the translations of Figure 4.3 a couple of points are worth noting. First, the word "or" is sometimes used in English in an inclusive and sometimes in an exclusive sense. The usage that we have shown is inclusive, meaning that it remains true when both p and q are true. In contrast, we treat the expression "either p or q" as being exclusive: if both p and q are true the statement is false. Another point worth noting is that we interpret the word "unless" to mean that p and q cannot be true at the same time.

There are also some words that signal logical connectives in various ways depending on context. One of these is "but." In some cases it indicates conjunction. For instance the sentence

"The dog was small but fierce."

p and q	becomes	$p \wedge q$
both p and q	becomes	$p \wedge q$
p or q	becomes	$p \vee q$
either p or q	becomes	$(p \vee q) \wedge \neg(p \wedge q)$
not	becomes	$\neg$
it is not the case that	becomes	$\neg$
neither p nor q	becomes	$\neg(p \vee q)$
if p then q	becomes	$p \Rightarrow q$
p if q	becomes	$q \Rightarrow p$
p only if q	becomes	$p \Rightarrow q$
p if and only if q	becomes	$p \Leftrightarrow q$
p unless q	becomes	$p \Leftrightarrow \neg q$

Figure 4.3: Translation of English connective words

would be translated into propositional logic as $s \wedge f$ where s stands for proposition "*the dog was small*", and f stands for proposition "*the dog was fierce*".

One exception to translating "not" into $\neg$ is in expressions of the form "not only ... but also" as follows:

> *"Not only did the cat jump over the fence, but he also scratched the paint."*

In this case the sentence has the same meaning as

> *"The cat jumped over the fence and scratched the paint."*

Implication: Sufficiency and Necessity

We call p a *sufficient condition* for q to be true if $p \Rightarrow q$. We call p a *necessary condition* for q to be true if $q \Rightarrow p$.

Spotting sufficiency and necessity separately in colloquial English is not particularly challenging. They are often expressed in terms of "it is enough" (sufficiency), "it is sufficient" (sufficiency), "it is necessary" (necessity), "must" (necessity). For example

> *"For two straight lines to be parallel, the lines must not cross."*

is translated as $spl \Rightarrow ncl$ where spl stands for "two straight lines are parallel", and ncl stands for "the lines do not cross." The lines not crossing is a necessary

condition for *spl*. However, *ncl* is not a sufficient condition for *spl* because $ncl \Rightarrow spl$ is not true when the straight lines are on different planes.

Bi-implication

In informal English descriptions we rarely see the use of "if and only if." Therefore, it can be challenging to recognize when we are dealing with bi-implication.

Bi-implication is associated with facts that are both sufficient and necessary for some other fact to be true. Such conditions usually arise in definitions. For example, the definition of equilateral triangle can be expressed as:

> *"The triangle is equilateral if its sides are equal."*

We translate the definition as $te \Leftrightarrow se$ where *te* stands for "the triangle is equilateral" and *se* for "the sides of the triangle are equal." If we know that a triangle is equilateral, by definition, we know that its sides are equal; on the other hand if we show that the sides of a triangle are equal we can conclude, by definition, that the triangle is equilateral.

Outside definitions, information needed to determine that we are dealing with bi-implication is often either implicit or completely missing from the English descriptions. Therefore, examining the context and domain of interest is often necessary to determine whether we are dealing with bi-implication. For example, for this statement

> *"For two straight lines to be parallel, the lines must not cross and the lines must be on the same plane."*

the direct translation would be:

$$spl \Rightarrow ncl \wedge samepl$$

where *samepl* stands for "the lines are on the same plane." However, we know from Euclidean geometry that we are dealing with conditions that are both sufficient and necessary, therefore, we write:

$$spl \Leftrightarrow ncl \wedge spl$$

4.7.3 Example: More Traffic Lights

We now return to the traffic lights example with a more-detailed formal representation. We introduce the following atomic propositions:

nsg: "the North-South light is green"
nsy: "the North-South light is yellow"
nsr: "the North-South light is red"
ewg: "the East-West light is green"
ewy: "the East-West light is yellow"
ewr: "the East-West light is red"

We then define:

$$\begin{array}{lll} nsl_1 & == & nsg \vee nsy \vee nsr \\ nsl_2 & == & \neg(nsg \wedge nsy) \wedge \neg(nsg \wedge nsr) \wedge \neg(nsy \wedge nsr) \end{array}$$

nsl_1 follows directly from the informal description. While nsl_2 is missing from the description, it reflects our knowledge that a light cannot be both red and green at the same time, etc. When formalizing a domain, such as traffic light systems, it is often the case that there exist such implicit facts that will need to be made explicit in order to reason about those systems. In many cases we may not realize that we need them until we discover that we cannot prove some property that we would intuitively expect to hold. We then go back and add those additional facts.

Similarly, we write:

$$\begin{array}{lll} ewl_1 & == & ewg \vee ewy \vee ewr \\ ewl_2 & == & \neg(ewg \wedge ewy) \wedge \neg(ewg \wedge ewr) \wedge \neg(ewy \wedge ewr) \end{array}$$

So far so good, but a question might occur to the reader: what if we were talking about lights that could have more than three colors? Enumerating all the possibilities would surely result in lengthy sentences. Similarly, it may seem redundant to have two sets of facts (nsl_i, and ewl_i) expressing the same properties for each direction (North-South, and East-West). In propositional logic we have no way of avoiding enumeration of color possibilities, or stating the same properties for each direction but, fortunately, in the next chapter we will be able to express such collection of sentences much more concisely.

We now detail the rules the traffic lights must obey.

$$\begin{array}{lll} r_1 & == & nsg \vee nsy \Rightarrow ewr \\ r_2 & == & ewg \vee ewy \Rightarrow nsr \end{array}$$

The resulting facts about traffic lights would be expressed as:

$$\mathit{traffic\_lights}_3 == \mathit{nsl}_1 \wedge \mathit{nsl}_2 \wedge \mathit{ewl}_1 \wedge \mathit{ewl}_2 \wedge r_1 \wedge r_2$$

We started this more-detailed representation of traffic lights with the intention of being able to say that if we know that the North-South light is green then the East-West light must be red. We can express this proposition easily as $\mathit{nsg} \Rightarrow \mathit{ewr}$. We can also formally derive this proposition *within* the traffic lights world; in this world we can use the facts of $\mathit{traffic\_lights}_3$ as premises of our derivations. We show that

$$\mathit{nsl}_1, \mathit{nsl}_2, \mathit{ewl}_1, \mathit{ewl}_2, r_1, r_2 \vdash \mathit{nsg} \Rightarrow \mathit{ewr}$$

1.	$\mathit{nsg} \vee \mathit{nsy} \Rightarrow \mathit{ewr}$	premise (r_1)
2.	nsg	assumption
3.	$\mathit{nsg} \vee \mathit{nsy}$	$\vee$-intro, 2
4.	ewr	$\Rightarrow$-elim, 1,3
5.	$\mathit{nsg} \Rightarrow \mathit{ewr}$	$\Rightarrow$-intro, 2–4

Another interesting property to prove about our traffic lights is that the lights cannot both be green at the same time. A situation in which both lights were green could be catastrophic indeed. We would like, therefore, to make sure that it cannot happen in our model. In later chapters we will call such properties *safety properties*, because as the name suggests they are meant to show that "bad situations" cannot happen.

Let us show that the lights cannot be green at the same time, that is

$$\mathit{nsl}_1, \mathit{nsl}_2, \mathit{ewl}_1, \mathit{ewl}_2, r_1, r_2 \vdash \neg(\mathit{nsg} \wedge \mathit{ewg})$$

We use a proof by contradiction to derive this property: we assume that both lights can be green at the same time. But from the property we just proved (and which we call "nsg-ewr"), the North-South light's greenness implies that the East-West light is red. So we have that the East-West light is simultaneously green and red, which contradicts ewl_2.

1.	$nsg \wedge ewg$	assumption
2.	nsg	$\wedge$-elim, 1
3.	$nsg \Rightarrow ewr$	nsg-ewr
4.	ewr	$\Rightarrow$-elim, 3,2
5.	ewg	$\wedge$-elim, 1
6.	$ewg \wedge ewr$	$\wedge$-intro, 5,4
7.	$\neg(ewg \wedge ewy) \wedge \neg(ewg \wedge ewr) \wedge \neg(ewy \wedge ewr)$	premise (ewl_2)
8.	$\neg(ewg \wedge ewy) \wedge \neg(ewg \wedge ewr)$	$\wedge$-elim, 7
9.	$\neg(ewg \wedge ewr)$	$\wedge$-elim, 8
10.	$\neg(nsg \wedge ewg)$	$\neg$-elim, 1,6,9

A note about line 8 of the derivation: we are assuming that the parenthesized version of

$$\neg(ewg \wedge ewy) \wedge \neg(ewg \wedge ewr) \wedge \neg(ewy \wedge ewr)$$

is

$$(\neg(ewg \wedge ewy) \wedge \neg(ewg \wedge ewr)) \wedge \neg(ewy \wedge ewr)$$

Chapter Notes

[TBD]

Further Reading

[TBD]

4.8 Exercises

1. Which of the following are well-formed sentences in propositional logic?

   (a) $p\neg q$
   (b) $\neg\neg\neg(p \wedge r \wedge q)$
   (c) $\Rightarrow (s \Leftrightarrow r)$
   (d) $((q \Leftrightarrow r) \rightarrow s)$

2. The following sentences have no parentheses and are parsed according to the precedence rules.

   (a) $\neg q \vee r \Rightarrow s$
   (b) $q \Leftrightarrow r \Rightarrow s$
   (c) $p \vee s \wedge \neg\neg q \Leftrightarrow p \wedge s$
   (d) $p \Rightarrow q \Rightarrow r$
   (e) $q \Leftrightarrow r \Leftrightarrow s$

   Questions:

   (i) Add parentheses as intended by the precedence rules.
   (ii) Is there more than one way of adding parentheses to the sentences? Is that a problem?

3. Construct truth tables for each of the following:

   (a) $p \vee (p \wedge q)$
   (b) $p \wedge (p \vee q)$
   (c) $\neg p \Rightarrow (p \wedge (q \Rightarrow p))$
   (d) $\neg p \wedge (p \vee (q \Rightarrow p))$
   (e) $(p \Rightarrow q) \Rightarrow (\neg p \vee q)$
   (f) $(p \Rightarrow q) \Leftrightarrow (\neg p \vee q)$

4. Which of the sentences in Exercise 3 are:

   (a) valid?
   (b) consistent?
   (c) contingent?
   (d) inconsistent?

5. Show that the following sentences are valid using truth tables:

   (a) $p \Rightarrow q \Leftrightarrow \neg p \vee q$
   (b) $p \Rightarrow q \Leftrightarrow \neg q \Rightarrow \neg p$
   (c) $\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$
   (d) $\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$

6. Prove the following statements using propositional calculus:

   (a) $p \Leftrightarrow q \vdash q \Rightarrow p$

(b) $p, \neg p \vdash q$
(c) $q \wedge \neg q \vdash p \Rightarrow r$
(d) $(p \wedge q) \wedge r \vdash p \wedge (q \wedge r)$
(e) $\neg\neg q \vdash q \vee r$
(f) $p \wedge q, p \Rightarrow s, q \Rightarrow t \vdash s \wedge t$
(g) $\vdash (p \wedge q) \Rightarrow q$
(h) $q \Rightarrow \neg p, p \wedge q \vdash r$
(i) $p \wedge q \vdash p \Rightarrow q$
(j) $\neg p \wedge \neg q \vdash p \Leftrightarrow q$

7-8. **Normal Forms**

There are many equivalent ways to represent a sentence in propositional logic. Sometimes it is useful to represent sentences using a standard structure, called a *normal form*. Normal forms are particularly useful in the context of tool construction for automating reasoning about propositional sentences. First, having to work only on a subset of structures makes tool construction more efficient. Second, solutions for certain problems can be determined with remarkable efficiency for some normal forms. However, when sentences are not in the desired normal form to start with, it can be computationally expensive to transform the sentences into the desired normal form.

We now define three normal forms. Sentences in these normal forms are represented using only $\neg, \wedge$ and $\vee$.

If p is an atomic proposition, we call p and $\neg p$ *literals*.

A sentence is in *conjunctive normal form* (CNF) if it is a conjunction of disjunctions of literals. Formally, sentences in CNF have the following syntax:

$$
\begin{aligned}
\mathit{atom} &= \text{“}p\text{”} \mid \text{“}q\text{”} \mid \text{“}r\text{”} \mid \ldots \mid \text{“}p_1\text{”} \mid \text{“}q_1\text{”} \mid \text{“}r_1\text{”} \mid \ldots \\
\mathit{literal} &= \mathit{atom} \mid \text{“}\neg\text{”}, \mathit{atom} \\
\mathit{clause} &= \mathit{literal} \\
&\quad \mid \text{“(”}, \mathit{clause}, \text{“}\vee\text{”}, \mathit{clause}, \text{“)”} \\
\mathit{CNF} &= \mathit{clause} \\
&\quad \mid \text{“(”}, \mathit{CNF}, \text{“}\wedge\text{”}, \mathit{CNF}, \text{“)”}
\end{aligned}
$$

CNF is useful when reasoning about satisfiability of propositional sentences. For example, for a sentence in CNF showing satisfiability boils down to showing that for each clause there exists a literal that is satisfiable.

A sentence is in *disjunctive normal form* (DNF) if it is a disjunction of conjunctions of literals.

A sentence is in *negation normal form* (NNF) if negation appears only in the context of a literal (that is, next to an atomic propositional symbol). There is no restriction on how conjunction and disjunction are used. Formally, sentences in NNF have the following syntax:

$$\begin{aligned} \mathit{atom} &= \text{“}p\text{”} \mid \text{“}q\text{”} \mid \text{“}r\text{”} \mid \ldots \mid \text{“}p_1\text{”} \mid \text{“}q_1\text{”} \mid \text{“}r_1\text{”} \mid \ldots \\ \mathit{literal} &= \mathit{atom} \mid \text{“}\neg\text{”}, \mathit{atom} \\ \mathit{NNF} &= \mathit{literal} \\ &\quad \mid \text{“(”}, \mathit{NNF}, \text{“}\vee\text{”}, \mathit{NNF}, \text{“)”} \\ &\quad \mid \text{“(”}, \mathit{NNF}, \text{“}\wedge\text{”}, \mathit{NNF}, \text{“)”} \end{aligned}$$

Example

- $(p \vee \neg q) \wedge (q \vee r \vee \neg p)$ is in CNF and NNF.
- $(q \wedge \neg q \wedge p) \vee p \vee (\neg r \wedge p)$ is in DNF and NNF.
- $((p \vee \neg q) \wedge (q \vee r \vee \neg p)) \vee \neg r$ is in NNF but not in CNF or DNF.
- $\neg(p \vee q)$ is not in normal form.

7. **NNF**

   (a) Describe an algorithm that transforms an arbitrary propositional sentence into an *equivalent* sentence in NNF.
   Two sentences are called equivalent if their truth values coincide for every interpretation of atomic propositions.
   (b) How did you eliminate implications and bi-implications?
   (c) What is the complexity of your algorithm?

8. **CNF**

   (a) Describe an algorithm that transforms an arbitrary propositional sentence in NNF into an *equivalent* sentence in CNF.
   (b) What is the complexity of your algorithm?

Chapter 5

Predicate Logic

In Chapter 4 we described the formal system of propositional logic, which enables reasoning about propositions. We now extend both the language of propositional logic and its inference system to enable reasoning about more complex statements, called *predicates*. The resulting language is called *predicate logic* and an inference system for it is called a *predicate calculus*.

5.1 Predicates

Sometimes we want to assert the proposition that all members of some set satisfy a property. Suppose we have a set of friends $\{Larry, Joe, Moe\}$. If you wanted to state that they are all tall you might do this in propositional logic using three propositions and conjunction:

"Larry is tall" $\wedge$ *"Joe is tall"* $\wedge$ *"Moe is tall"*

While this achieves the desired effect, for large sets enumeration becomes unwieldy. Worse, for sets that have an infinite number of elements, such enumeration is not possible.

An alternative is to introduce a property template "$Tall()$" — we call such a property template a *predicate*. Given a friend, x, $Tall(x)$ would be either true or false depending on whether that person is tall or not. For our small set of friends we could then write:

$$Tall(Larry) \wedge Tall(Joe) \wedge Tall(Moe)$$

This helps since the use of a predicate eliminates the need to define distinct propositions for each element of the set. But we still have the problem that we

must write down the property for each element of the set. To avoid the need for explicit enumeration we introduce the following new syntax:

$$\forall x : Friends \bullet Tall(x)$$

to represent

$$Tall(friend1) \wedge Tall(friend2) \wedge \ldots$$

where the set *Friends* is defined as $\{friend1, friend2, \ldots\}$. The notation "$x : Friends$" indicates that x is a *variable* whose values are drawn from the set *Friends*. That is, x stands for any object of the set *Friends*.

Similarly we introduce the syntax:

$$\exists x : Friends \bullet Tall(x)$$

to represent:

$$Tall(friend1) \vee Tall(friend2) \vee \ldots$$

where, again, "$x : Friends$" indicates that x stands for any object in the set *Friends*.

The notational limitations of propositional logic are problematic, but not particularly serious: after all, we can often capture our intent using single propositions, such as "All roads lead to Rome." However, as we noted earlier, such propositions limit our ability to use their structure to derive new facts. For example, consider the following propositions:

1. *"All roads lead to Rome."*
2. *"X is a road."*
3. *"X leads to Rome."*

What can we say about the truth or falsity of the last proposition? Although our intuition suggests that the last proposition should follow from the first two, in propositional logic we cannot deduce anything useful about the last proposition from the first two. As we will see shortly, predicate logic and its inference system will enable us to express the structure of propositions so that we can reason as follows: "given that all roads lead to Rome, X's being a road implies that X leads to Rome."

A predicate like *Tall* is an example of a *unary* predicate: it has just one place for an object name to be put. We will also allow *n-ary* predicates, which can be thought of as representing some relationship among n objects. For example, the predicate "$ParentOf(x, y)$" can be used to express the fact that "y is a parent of x".

5.2 Syntax

The sentences of predicate logic are defined by extending the grammar for propositional logic as follows:

$$
\begin{array}{ll}
\textit{sentence} & = \textit{atomic proposition} \mid \textit{predicate} \\
& \mid \text{“}\neg\text{”}, \textit{sentence} \\
& \mid \text{“(”}, \textit{sentence}, \text{“}\vee\text{”}, \textit{sentence}, \text{“)”} \\
& \mid \text{“(”}, \textit{sentence}, \text{“}\wedge\text{”}, \textit{sentence}, \text{“)”} \\
& \mid \text{“(”}, \textit{sentence}, \text{“}\Rightarrow\text{”}, \textit{sentence}, \text{“)”} \\
& \mid \text{“(”}, \textit{sentence}, \text{“}\Leftrightarrow\text{”}, \textit{sentence}, \text{“)”} \\
& \mid \text{“(”}, \text{“}\forall\text{”}, \textit{variable}, \text{“ :”}, \textit{setname}, \text{“}\bullet\text{”}, \textit{sentence}, \text{“)”} \\
& \mid \text{“(”}, \text{“}\exists\text{”}, \textit{variable}, \text{“ :”}, \textit{setname}, \text{“}\bullet\text{”}, \textit{sentence}, \text{“)”} \\
\textit{atomic proposition} & = \text{“}p\text{”} \mid \text{“}q\text{”} \mid \text{“}r\text{”} \mid \ldots \mid \text{“}p_1\text{”} \mid \text{“}q_1\text{”} \mid \text{“}r_1\text{”} \mid \ldots \\
\textit{predicate} & = \textit{predicate name}, \text{“(”}, \textit{termlist}, \text{“)”} \\
\textit{term list} & = \textit{term} \mid \textit{term}, \text{“,”}, \textit{term list} \\
\textit{term} & = \textit{constant} \mid \textit{variable} \mid \textit{function application} \\
\textit{constant} & = \text{“}a\text{”} \mid \text{“}b\text{”} \mid \text{“}c\text{”} \mid \ldots \mid \text{“}a_1\text{”} \mid \text{“}b_1\text{”} \mid \text{“}c_1\text{”} \mid \ldots \\
\textit{variable} & = \text{“}x\text{”} \mid \text{“}y\text{”} \mid \text{“}z\text{”} \mid \ldots \mid \text{“}x_1\text{”} \mid \text{“}y_1\text{”} \mid \text{“}z_1\text{”} \mid \ldots \\
\textit{function application} & = \textit{function name}, \text{“(”}, \textit{term list}, \text{“)”}
\end{array}
$$

Usually we use upper case letters $P, Q, R, \ldots$ to denote generic predicates. We also allow predicate names that suggest a particular property or relationship, such as *Tall* and *ParentOf*.

The “parameters” of a predicate are called *terms*. Terms are *constants*, *variables*, or formed by *function application*. A constant represents a specific, fixed object in a set of objects. A variable represents an undetermined object in a set of objects — it can be instantiated with the name of any object belonging to the set in question. Function application allows us to create expressions such as $5x+3$, and $\sqrt{a}$. [1] We will discuss functions more formally in Chapter 6. For now suffice it to say that function application must obey well-formedness rules regarding the number of parameters and the sets associated with them. For example, for addition of numbers $+$ to make sense it must be applied to two parameters and the parameters must be numbers.

We assume that each predicate symbol has a fixed *arity*, representing the number of places for terms. For a sentence to be well-formed the term list associated with each predicate must have the same length as the arity of that predicate.

[1] Formally these expressions would be written as $\sqrt{}(a,2)$, and $+((\times(5,x)),3)$, but we will typically use the more familiar forms.

The symbols $\forall$ and $\exists$ are called *quantifiers*. $\forall$ is called the *universal quantifier*, and $\exists$ is called the *existential quantifier*. In sentences of the form $(\mathit{quantifier}\ x : S \bullet \ldots)$, x is called the *quantified variable*, and the part after "$\bullet$" is called the *scope* of the quantifier. The quantifier is said to *range over* S, where S is a set of values.

Example 5.1. The following are sentences in predicate logic:

- $(\exists y : S \bullet (P(x) \wedge Q(x)))$
  The scope of $\exists$ is $P(x) \wedge Q(x)$.
- $(p \Leftrightarrow (\forall x : T \bullet Q(x)))$
  The scope of $\forall$ is $Q(x)$.
- $(\exists x : S \bullet ((\forall y : \mathbb{N} \bullet R(x,y)) \vee \neg Q(x,z)))$
  The scope of $\exists$ is $((\forall y : \mathbb{N} \bullet R(x,y)) \vee \neg Q(x,z))$; the scope of $\forall$ is $R(x,y)$.
- $((\forall z : \mathbb{Z} \bullet (\forall y : S \bullet P(z,y,x))) \Rightarrow \neg r)$
  The scope of the outermost $\forall$ is $(\forall y : S \bullet P(z,y,x))$; the scope of the innermost $\forall$ is $P(z,y,x)$.

□

Conventions The precedence rules for propositional sentences can also be used in predicate logic to eliminate parentheses. Additionally, we will use the following conventions to improve sentence readability:

– When parentheses are missing around a quantified sentence we will assume that the quantifier scope stretches as far to the right as possible. Specifically, the scope is taken to be the first unmatched right parenthesis, '$)$', to the right of the '$\bullet$', or the end of the sentence if none exists. For example,

$$(p \vee \forall x : S \bullet \neg Q(x) \Rightarrow P(x)) \wedge q$$

should be read as:

$$(p \vee (\forall x : S \bullet (\neg Q(x) \Rightarrow P(x)))) \wedge q$$

– We will use the following notation:

$$\forall x, y : T;\ z : S \bullet \ldots$$

as syntactic sugaring for:

$$\forall x : T \bullet (\forall y : T \bullet (\forall z : S \bullet \ldots))$$

Similarly, we may combine several $\exists$ appearing one after the other. However, we may not combine $\forall$ and $\exists$ together.

Bound and Free Variables The grammar allows one to use any variable name as a term in a predicate. In many cases such a variable will have been introduced as the quantified variable in a scope that includes the predicate. But this is not strictly necessary. To distinguish between variables that are within the scope of a quantifier and those that aren't we introduce the following terminology:

- A variable x is said to be *bound* if it occurs within the scope of a quantifier whose variable is x.
- A variable x is said to be *free* if it is not bound.
- A sentence that does not have any free variables is said to be *closed*.

A variable can occur both free and bound in the same sentence.

Example 5.2. In the following sentences x is bound, y is free, and z is both free and bound.

- $\forall x : S \bullet (\exists z : T \bullet (P(x,z) \land Q(z,y))) \lor Q(z,x)$
- $z \in \mathbb{Z} \land (y \leq z) \land (\exists x : \mathbb{Z} \bullet (\forall z : \mathbb{Z} \bullet x \leq z))$

□

5.3 Semantics

The semantics of predicate logic extends that of propositional logic. As before, each sentence will be mapped to the domain of *true* or *false*. Moreover, atomic propositions and propositional connectives are interpreted exactly as in propositional logic.

A variable is interpreted as the objects whose names may be used to instantiate it; so a variable has as many interpretations as the number of objects in the set the variable is drawn from. Each constant has a single interpretation — that of the object that it denotes. Informally, function application $f(t_1, t_2, \ldots, t_n)$ is interpreted by first interpreting the function name "f," and all the variables and constants appearing in its parameter list. Application is then interpreted as "the value" of the function for the given arguments (in the sense elaborated in Chapter 6). For example, $f(x,3)$ will be interpreted as the value 5 when f is interpreted as addition and x is interpreted as 2, but as 12 when f is interpreted as multiplication and x is interpreted as 4. We often associate a fixed interpretation with some function names; for example, $+$ is usually interpreted as addition.

A predicate $P(t_1, t_2, \ldots, t_n)$ is interpreted by first interpreting the predicate name "P" and the term list $t_1, t_2, \ldots, t_n$. Interpreting "P" means deciding which relationship it is intended to capture. For example, in $P(x, y)$ we might interpret P to represent the father-child relationship. Alternatively, we might interpret it to represent the "smaller-than" relationship. As with function names, some predicate names suggest a fixed interpretation. For example, in $NumericalLessThan(x, y)$ we interpret *NumericalLessThan* to represent the less-than relationship.

Associating meanings with predicate names leads to *typing* restrictions on the arguments of a predicate. For example, *FatherOf*(*Joe*, *Moe*) makes sense only if both Joe and Moe are persons (or at least members of the same species). On the other hand, *FatherOf*(*Moe*, *Finland*) is meaningless (assuming that Finland stands for the country of Finland and that we are not talking about some abstract notion of fathering). Predicates whose arguments satisfy typing restrictions are called *well-typed*.

A predicate is interpreted by evaluating its truth value for each interpretation of its argument list. For example, $Positive(x)$, where x is an integer variable, is interpreted as false for x instantiated by 0 or any negative integer, and as true otherwise.

Finally, quantified sentences are interpreted as follows: First we interpret the predicate and propositional symbols appearing within the scope of the quantifier. Then $\forall x : T \bullet P(x)$ is interpreted as the proposition: "$P(x)$ interprets to true for every value of x from T." Similarly, $\exists x : T \bullet P(x)$ gives rise to the following proposition "$P(x)$ interprets to true for at least one value of x from T." The rules for predicates of higher arities are similar.

Note that when T is a finite set with elements $t_1, t_2, ..., t_n$ the above interpretation of $\forall$ is the same as conjunction of $P(x)$ over all interpretations of x: $P(t_1) \wedge P(t_2) \wedge \ldots \wedge P(t_n)$. Similarly, the interpretation of $\exists$ is the same as the disjunction of $P(x)$ over all interpretations of x: $P(t_1) \vee P(t_2) \vee \ldots \vee P(t_n)$.

The notions of *satisfiability, validity, and unsatisfiability* from propositional logic extend naturally to predicate logic:

- A predicate sentence is said to be *satisfiable* if there exists an interpretation of its atomic propositions, terms, and predicate names such that the sentence is true.
- A predicate sentence is said to be *valid* if it is true for every interpretation of its atomic propositions, terms, and predicate names.
- A predicate sentence is said to be *unsatisfiable* if it is false for every interpretation of its atomic propositions, terms, and predicate names.

Variable Renaming A useful consequence of the above semantics is that under certain conditions we can change the name of a quantified variable. Specifically, suppose y is not a free variable in $P(x)$, which appears in a sentence of the form:

$$(\textit{quantifier}\ x : T \bullet P(x))$$

Renaming the quantified variable from x to y and replacing all occurrences of x with y does not change the semantics of the quantified sentence. This allows us to change the quantified variable to another. Normally when we do renaming we will pick a y that does not appear at all in $P(x)$, and hence is not a free variable.

Example 5.3.

- y may be renamed to z in the following sentence:

  $$\exists x : \mathbb{Z} \bullet (\forall y : \mathbb{Z} \bullet P(x, y))$$

  to give:

  $$\exists x : \mathbb{Z} \bullet (\forall z : \mathbb{Z} \bullet P(x, z))$$

- y may *not* be renamed to x in the following sentence:

  $$\exists x : \mathbb{Z} \bullet (\forall y : \mathbb{Z} \bullet P(x, y))$$

  Renaming y to x here would result in the free occurrence of x in $\forall y : \mathbb{Z} \bullet P(x, y)$ to become bound (this is also known as *variable capturing*). We could, however, achieve a correct renaming of y to x by first renaming x to z.

□

Some Important Predicates We assume that each set comes equipped with two special predicates: the *element of* predicate "$\in$" where $a \in S$ holds when a is a member of set S, and the *equality* predicate "$=$" where $a = b$ holds when a and b denote the same element of set S. (Sets will be discussed more thoroughly in Chapter 6.) We will also use some well-known predicates on integers ($\mathbb{Z}$), such as "$<$" (interpreted as $x < y$ if "x is smaller than y"), and some of their properties, without explicitly defining them.

Textual Substitution Sometimes we would like to change a sentence S by replacing all the free occurrences of a variable x with a term t. We write $S[x \leftarrow t]$ to denote such substitution. For example $(x^2 \leq y)[y \leftarrow 2z]$ results in the new sentence being $x^2 \leq 2z$.

When performing textual substitution we need to be careful not to change the meaning of the sentence. For example, if the sentence was valid before the substitution it should be so after the substitution. The key to not changing the meaning of the sentence is to not allow any free variables in t to become bound after the substitution $S[x \leftarrow t]$. To avoid capturing of variables we rename bound variables in S prior to substitution choosing *fresh* variable names. This usually means that we use variable names that do not appear in S or t. For example, substitution $(\exists x : \mathbb{N} \bullet y \leq x)[y \leftarrow x+1]$ is carried out by first renaming the bound x to z to get $(\exists z : \mathbb{N} \bullet y \leq z)[y \leftarrow x+1]$ and then replacing y with $x+1$ to get $\exists z : \mathbb{N} \bullet x+1 \leq z$. This sentence is valid (like the original): regardless of the value of x we can find a larger natural number. In contrast, incorrect substitution that does not avoid variable capturing would give rise to $\exists x : \mathbb{N} \bullet x+1 \leq x$ — an unsatisfiable sentence.

5.4 Predicate Calculus

As with the language for predicate logic, we will obtain a predicate calculus by augmenting propositional calculus: specifically, we define rules for introducing and eliminating quantifiers.

5.4.1 The Universal Quantifier

- Universal introduction:

a.	$a \in S$	assumption	|
	$\vdots$		|
c.	$P(a)$		|
	$\forall x : S \bullet P(x)$	$\forall$-intro, a–c	[a is a fresh variable]

  $P(a)$ denotes $P(x)$ with free occurrences of x having been replaced everywhere by a. The rule expresses the intuition that if we can derive $P(a)$ for an *arbitrary* element a of S (that is, the proof cannot depend on any specific details of a), then it must hold for *all* elements of S.

This rule includes a *side condition:* "a is a fresh variable." This constraint is included to ensure that a is indeed an arbitrary element of S. Specifically, "fresh variable" here means that a should not appear free in $\forall x : S \bullet P(x)$ or in any of the undischarged assumptions.

- Universal elimination:

  a. $\forall x : S \bullet P(x)$
  b. $a \in S$

  $P(a)$ $\forall$-elim, a,b

  The intuition behind this rule is that if we know that $P(x)$ holds for all values x in S, and a is a specific element of S, then it must hold for a.

To see how these rules are used, consider the following derivation:

Example 5.4. We show that $\vdash (\forall x : S \bullet P(x)) \Rightarrow (\forall x : S \bullet P(x) \vee Q(x))$.

1.	$\forall x : S \bullet P(x)$	assumption
2.	$x \in S$	assumption
3.	$P(x)$	$\forall$-elim, 1,2
4.	$P(x) \vee Q(x)$	$\vee$-intro, 3
5.	$\forall x : S \bullet P(x) \vee Q(x)$	$\forall$-intro, 2–4
6.	$(\forall x : S \bullet P(x)) \Rightarrow (\forall x : S \bullet P(x) \vee Q(x))$	$\Rightarrow$-intro, 1–5

□

Example 5.5. An **incorrect** derivation using $\forall$-intro.

1.	$z \in S$	assumption	← **correct**
2.	$P(z,z)$	assumption	
3.	$z \in S$	assumption	← **incorrect!**
4.	$P(z,z)$	copy from 2.	
5.	$\forall x : S \bullet P(x,z)$	$\forall$-intro, 3–4	
6.	$P(z,z) \Rightarrow (\forall x : S \bullet P(x,z))$	$\Rightarrow$-intro, 2–5	
7.	$\forall y : S \bullet P(y,y) \Rightarrow (\forall x : S \bullet P(x,y))$	$\forall$-intro, 1–6	

This derivation erroneously "shows" that if a binary predicate P is true when its two arguments are the same, then it must hold for any values of its variables — clearly not something that we would expect to be valid. The incorrect usage results from introducing the assumption $z \in S$, violating the side condition of $\forall$-introduction: z occurs free in $\forall x : S \bullet P(x,z)$, and the undischarged assumptions $z \in S$ and $P(z,z)$. □

Special cases of the universal rules arise when S is *empty*: that is to say, S contains no elements. When S is empty then the statement $\forall x : S \bullet P(x)$ is always true — the statement is said to hold *vacuously*. Formally, since the assumption $a \in S$ in the introduction rule contradicts S's being empty, anything can be derived from this contradiction, including $\forall x : S \bullet P(x)$. On the other hand, for an empty S we can never derive $a \in S$ in the elimination rule, so we can never prove $P(a)$ in this case.

5.4.2 The Existential Quantifier

- Existential introduction:

  a. $a \in S$
  b. $P(a)$

  $\exists x : S \bullet P(x)$ $\exists$-intro, a,b

  The existential introduction rule is similar to $\vee$ introduction: to derive $\exists x : S \bullet P(x)$ it is sufficient to show that P is satisfiable by one member of S.

- Existential elimination:

  a. $\exists x : S \bullet P(x)$
  b. $a \in S \wedge P(a)$ assumption
  $\vdots$
  d. R

  R $\exists$-elim, a,b–d [a is a fresh variable]

  The intuition behind the elimination rule is that $\exists x : S \bullet P(x)$ says that P holds for at least one member of S. Thus we are trying to derive R when at least one element of S has the property P. By picking a to be an *arbitrary* element of S that makes P true, and showing that R must follow, we know that the proof would work for whichever a makes $P(a)$ true.

  The constraint that a is a fresh variable is included to ensure that a is an arbitrary element of S. a should not appear free in $\exists x : S \bullet P(x)$, R, or any undischarged assumptions.

To see how these rules work, consider the following derivation:

Example 5.6. We show that $\exists x : S \bullet \exists y : T \bullet P(x,y) \vdash \exists y : T \bullet \exists x : S \bullet P(x,y)$

1.	$\exists x : S \bullet \exists y : T \bullet P(x,y)$	premise
2.	$z \in S \wedge \exists y : T \bullet P(z,y)$	assumption
3.	$\exists y : T \bullet P(z,y)$	$\wedge$-elim, 2
4.	$w \in T \wedge P(z,w)$	assumption
5.	$z \in S$	$\wedge$-elim, 2
6.	$P(z,w)$	$\wedge$-elim, 4
7.	$\exists x : S \bullet P(x,w)$	$\exists$-intro, 5,6
8.	$w \in T$	$\wedge$-elim, 4
9.	$\exists y : T \bullet \exists x : S \bullet P(x,y)$	$\exists$-intro, 8,7
10.	$\exists y : T \bullet \exists x : S \bullet P(x,y)$	$\exists$-elim, 3,4–9
11.	$\exists y : T \bullet \exists x : S \bullet P(x,y)$	$\exists$-elim, 1,2–10

Recall that we have let $\exists x : S, y : T \bullet P(x,y)$ be syntactic sugaring for $\exists x : S \bullet \exists y : T \bullet P(x,y)$. Thanks to this derivation, it does not matter in what order we combine the variables of existential quantifiers appearing next to each other. □

The condition of choosing a fresh variable in the existential elimination rule is important: using a variable that appears free in $\exists x : S \bullet P(x)$, R, or assumptions that have not been discharged yet, would mean assuming erroneously that there is a relationship between the element(s) of S that satisfy P and that free variable. To illustrate, consider the following invalid derivation of $\forall x : S \bullet P(x)$ from $\exists y : S \bullet P(y)$:

1.	$a \in S$	assumption	
2.	$\exists y : S \bullet P(y)$	premise	
3.	$a \in S \wedge P(a)$	assumption	← **incorrect!**
4.	$P(a)$	$\wedge$-elim, 3	
5.	$P(a)$	$\exists$-elim, 2,3–4	
6.	$\forall x : S \bullet P(x)$	$\forall$-intro, 1–5	

a appears free in the assumption $a \in S$ in line 1 — this assumption is considered undischarged throughout its scope (lines 1–5). Clearly, assuming $a \in S \wedge P(a)$ in line 3, means that the object represented by a in line 1 satisfies P, which contradicts the fact that we do not know which objects from S satisfy P.

Special cases of the existential rules arise when S is empty. We can never prove the $a \in S$ premise of the introduction rule. On the other hand, a contradiction arises in the premises of the elimination rule, allowing us to derive any sentence.

5.5 Equality

We mentioned *equality* "=" as a special predicate in Section 5.3. This special predicate denotes that two values from a set T are the same. Comparing two values for equality makes sense only if they are from the same set. Every set has an equality predicate associated with it; however, we do not generally distinguish between the equality symbols, writing $=$ to stand for $=_T, =_U$, etc. We also write $x \neq y$ as a shorthand for $\neg(x = y)$.

The properties of equality are captured as the following inference rules.

- Reflexivity:

$$\frac{}{t = t} \text{ eq-refl}$$

- Symmetry:

$$\frac{\text{a. } t_1 = t_2}{t_2 = t_1} \text{ eq-sym, a}$$

- Transitivity:

$$\frac{\begin{array}{ll}\text{a.} & t_1 = t_2 \\ \text{b.} & t_2 = t_3\end{array}}{t_1 = t_3} \text{ eq-trans, a,b}$$

- Substitution of equals for equals:

$$\frac{\begin{array}{ll}\text{a.} & t = u \\ \text{b.} & S[x \leftarrow t]\end{array}}{S[x \leftarrow u]} \text{ eq-sub, a,b}$$

  This final inference rule is simple, but extremely powerful. It says that if we know that two terms (t and u) are equal then whenever a property (S) holds about t (expressed as $S[x \leftarrow t]$) it also holds with u substituted for t (which is expressed as $S[x \leftarrow u]$. Sometimes this is called "substituting equals for equals."

The resulting formal system is known as *predicate logic with equality*. In Chapter 7 we present *equational reasoning* — a style of reasoning based on the properties of equality and substitution of equals for equals.

Example 5.7. As an example involving reasoning about equality let us derive $\vdash \forall x : S \bullet (\exists y : S \bullet y = x \wedge P(y)) \Rightarrow P(x)$.

1.	$x \in S$	assumption
2.	$\exists y : S \bullet y = x \wedge P(y)$	assumption
3.	$y \in S \wedge (y = x \wedge P(y))$	assumption
4.	$y = x \wedge P(y)$	$\wedge$-elim, 3
5.	$y = x$	$\wedge$-elim, 4
6.	$P(y)$	$\wedge$-elim, 4
7.	$P(x)$	eq-sub, 5,6
8.	$P(x)$	$\exists$-elim, 2,3–7
9.	$(\exists y : S \bullet y = x \wedge P(y)) \Rightarrow P(x)$	$\Rightarrow$-intro, 2–8
10.	$\forall x : S \bullet (\exists y : S \bullet y = x \wedge P(y)) \Rightarrow P(x)$	$\forall$-intro, 1–9

□

As we have noted, the meaning of equality will depend on the kinds of entities being compared, and whenever we use it we must be careful to indicate how equality is determined. In Chapter 6 we will, for example, define how equality on sets can be determined.

5.6 Derived Inference Rules

As with propositional calculus, predicate calculus has a meta-theorem that allows us to create theorems from derivations with premises.

Deduction Theorem Suppose that adding $P_1, P_2, \ldots, P_n$ as axioms of predicate logic, with the free variables of the P_i considered to be constants, allows Q to be proved. Then $\vdash P_1 \wedge P_2 \wedge \ldots \wedge P_n \Rightarrow Q$ is a theorem.

Example 5.8. Suppose we have shown that $0 < x \vdash \exists y : \mathbb{Z} \bullet y < 0 \wedge x + y = 0$. The side condition that x (in $0 < x$) be considered a constant, simply means that x denotes the same number in both $0 < x$ and $\exists y : \mathbb{Z} \bullet y < 0 \wedge x + y = 0$. From the deduction theorem we can conclude that $\vdash 0 < x \Rightarrow (\exists y : \mathbb{Z} \bullet y < 0 \wedge x + y = 0)$.
□

5.7 Soundness and Incompleteness

In Section 4.6 we stated that propositional calculus is both sound and complete with respect to the semantics of propositional logic in terms of truth values.

We have been somewhat informal in our discussion of what facts, or theories, we assume to be part of predicate calculus. We have explicitly included the introduction and elimination rules for predicate operators, axioms and inference rules related to equality, and assumed the inclusion of elementary arithmetic facts.

Predicate calculus, as presented here, is sound but *incomplete*. Soundness means that every theorem proved using predicate calculus (including, for example, the axioms related to arithmetic facts) is a true statement of predicate logic. Incompleteness means that there exist true statements of predicate logic for which no proof can be constructed using predicate calculus.

In fact, Gödel [4] proved in his first incompleteness theorem that it is impossible to create a formal system that is both sound and capable of encompassing *all* of the arithmetic facts: There will always exist true arithmetic facts that cannot be proved in a sound system. Other theories, such as the theory of sets that we discuss in Chapter 6, also give rise to incomplete formal systems.

5.8 Translating English into Logic

In Section 4.7 we talked about translating informal English descriptions into formal propositional logic sentences. Here we expand on that discussion with guidelines that deal with translations involving quantifiers.

5.8.1 Propositions Versus Predicates

As with propositional logic, one of the first things to decide is which facts about the domain of interest to represent as primitive elements. In the context of predicate logic, primitives are either (atomic) propositions, or primitive predicates. But how do we determine whether we should use propositions or predicates?

Let us reconsider the traffic lights example of Section 4.7. Recall that we introduced the following atomic propositions:

nsg: “the North-South light is green”
nsy: “the North-South light is yellow”
nsr: “the North-South light is red”
ewg: “the East-West light is green”
ewy: “the East-West light is yellow”
ewr: “the East-West light is red”

We then expressed the possible states of the North-South light as

$$\begin{array}{lll} nsl_1 & == & nsg \lor nsy \lor nsr \\ nsl_2 & == & \neg(nsg \land nsy) \land \neg(nsg \land nsr) \land \neg(nsy \land nsr) \end{array}$$

(As before, we use $==$ to indicate the introduction of a new name to represent the expression on the right-hand side of the symbol.)

An alternative would be to consider green, yellow, and red as the (only) elements of a set *Colors*, and introduce the predicate $NS(x)$, where x is a color variable, and $NS(x)$ holds if the North-South light is showing color x. We would then write:

$$\begin{array}{lll} NSL_1 & == & \exists x : Colors \bullet NS(x) \\ NSL_2 & == & \forall x, y : Colors \bullet x \neq y \Rightarrow \neg(NS(x) \land NS(y)) \end{array}$$

Similarly we can introduce $EW(x)$ for the East-West light, and express EWL_1 and EWL_2 analogously to NSL_1 and NSL_2. The resulting description is slightly more concise than the first one. However, a more important advantage of using predicates is that the expressions are immune to changes in the color possibilities: if traffic light design was to change to allow more than three colors, we would only need to change the definition of the colors set.

Another alternative for the traffic lights is to introduce a new set *Directions*, containing the elements *North-South* and *East-West*, and a binary predicate $L(x,y)$ where x represents the direction and y the color. We could then describe the traffic light rules as:

$$\begin{array}{lll} L_1 & == & \forall y : Directions \bullet \exists x : Colors \bullet L(y,x) \\ L_2 & == & \forall z : Directions;\ x, y : Colors \bullet x \neq y \Rightarrow \neg(L(x) \land L(y)) \\ R & == & \forall x, y : Directions \bullet \\ & & \quad x \neq y \land (L(x, green) \lor L(x, yellow)) \Rightarrow L(y, red) \end{array}$$

and the resulting traffic lights representation as

$$traffic\_lights_4 == L_1 \land L_2 \land R$$

This approach would be helpful if we should want to extend our description to also handle traffic lights that indicate whether a left or right turn is possible.

As a meta observation, as we have illustrated, the process of deciding how to use predicate logic to model a system of interest, depends significantly on the kind of reasoning that we plan to do, as well as our expectations about how the

model may need to be changed over time. By encapsulating certain concepts using sets and predicates, we can often insulate ourselves from later changes, in a way similar to how abstract data types and object-oriented classes have been used in programming to reduce the impact of changes on a software system.

5.8.2 Quantifiers

The universal quantifier $\forall$ is typically signaled by words such as "all", "every", "each", and "any." For example, "any" indicates that the property under consideration characterizes arbitrary members of a particular set; therefore, it characterizes *all* members of the set.[2] Similarly, the article "a" when used in the sense of "any" gives rise to a universally quantified sentence. For example, the first "a" in the sentence

"*A friend in need is a friend indeed.*"

indicates that *all* those who help you in difficult times are true friends.

The existential quantifier $\exists$ is signaled by words such as "there is", "there exist(s)", and "some." For example,

"*Some students have a background in logic.*"

translates to

$$\exists x : \mathit{Students} \bullet L(x)$$

where *Students* is the set of students, and $L(x)$ the predicate that holds if student x has a background in logic.

Quantifiers and Negation

The word "none" signals negation in connection with a quantifier. The sentence

"*None of the students has a background in logic.*"

would be translated into

$$\neg(\exists x : \mathit{Students} \bullet L(x))$$

or

$$\forall x : \mathit{Students} \bullet \neg L(s)$$

[2]Formally, the justification for introducing a universal quantifier in this case is given by the $\forall$-intro rule above.

Uniqueness and Exactness

Often we need to express that a specific number of objects have a certain property. Consider these examples:

"*One student failed the final.*"
"*Two students got A on the final.*"

Regardless of what the number of objects in question is — let us call it n — the formalization of such sentences is normally split into two parts:

1. Showing that there exist *at least n* objects with the property in question.
2. Showing that *at most n* objects have the property in question.

The sentence about one of the students failing the final is formalized as follows:

$$\exists x : \mathit{Students} \bullet F(x) \land \forall y : \mathit{Students} \bullet F(y) \Rightarrow y = x$$

Similarly, the sentence about the two students that got an A on the final can be formalized as:

$$\begin{array}{l} \exists x, y : \mathit{Students} \bullet \\ \quad x \neq y \land A(x) \land A(y) \land \forall z : \mathit{Students} \bullet A(z) \Rightarrow z = x \lor z = y \end{array}$$

Restrictive and Non-restrictive Clauses

We often add relative clauses in English sentences, which may or may not restrict the set of objects the sentence refers to.

The following use is restrictive.

"The students that had a background in logic did well on the final."

The restrictive clause "that had a background in logic" specifies which students are under discussion. In other words, this sentence captures that we only have information about how the students with background in logic did on the finals. Using $W(x)$ as the predicate that holds if x did well on the finals, we formalize the sentence as:

$$\forall x : \mathit{Students} \bullet L(x) \Rightarrow W(x)$$

The students without a logic background may or may not have done well on the final, and hence the use of implication is appropriate.

The following use is non-restrictive:

"The students, who had a background in logic, did well on the final."

The non-restrictive clause "who had a background in logic" *adds* information about the students.

$$\forall x : Students \bullet W(x) \land L(x)$$

The situation is a bit different when dealing with sentences that require existential quantification. In almost all such cases we would use conjunction when translating the sentences. For example, the sentence

"Some students, who shall remain nameless, failed the final."

is translated as

$$\exists x : Students \bullet F(x) \land N(x)$$

where $N(x)$ expresses that x "shall remain nameless." Moreover, the sentence

"Some of the crocodiles that Jim saw at the zoo looked menacing."

would also be translated using conjunction:

$$\exists x : Crocodiles \bullet Z(x) \land M(x)$$

where $Z(x)$ holds if Jim saw x at the zoo, and $M(x)$ if x looks menacing.

To understand why this sentence could not be translated as

$$\exists x : Crocodiles \bullet Z(x) \Rightarrow M(x)$$

notice that the sentence would be true if Jim did not see all of the crocodiles (making the antecedent of $Z(x) \Rightarrow M(x)$ false for some crocodile, and therefore the implication true), or if some crocodile looks menacing (making the consequent of $Z(x) \Rightarrow M(x)$ true for some crocodile, and therefore the implication true).[3] Most people would agree that this is not the intended meaning of the original sentence.

[3]Formally, this follows from the property

$$\exists x : S \bullet P(x) \Rightarrow Q(x) \dashv\vdash \neg(\forall x : S \bullet P(x)) \lor (\exists x : S \bullet Q(x))$$

5.8.3 Beyond Predicate Logic

There exist a variety of extensions of predicate logic. One important class are so-called *modal logics*, the most well-known of which are called *temporal logics*. Temporal logics are suitable for expressing temporal modalities such as "always," "eventually," "before." They extend predicate logic by describing how the passage of time affects the truth evaluation of predicates. Other useful modal logics include *epistemic logics*, which are suitable for expressing modalities such as "know" and "believe."

5.9 Fathers and Sons: A Formal Riddle System

So far we have used predicate calculus in its full generality. However, we are often interested in describing and reasoning about specific phenomena and their properties. To do this we build formal systems, based on predicate logic, that are tailored to the phenomena we are interested in.

The process of creating a formal system based on predicate logic normally involves the following steps:

1. Identify the sets of interest and the primitive predicates relevant for the phenomena under consideration.
2. Identify a set of axioms that capture properties of primitive predicates. Such properties are expressed as sentences of predicate logic and capture relationships among primitive predicates.
3. Use the inference rules of predicate calculus and axioms of the systems to build a a collection of derived facts.

5.9.1 Fathers and Sons

Consider the following two riddles:

Riddle 1 *"Brothers and sisters have I none,*
but this man is my father's son."

Riddle 2 *"Brothers and sisters have I none,*
but this man's father is my father's son."

For each riddle, what can we say about the relationship between the person telling the riddle and the unidentified man? Moreover, if we know the solution to the riddle how can we prove that formally?

We now show how to create a formal system to express and reason about such riddles. We call our system the Riddle System.

Sets and Predicates of Interest

First we need to identify the set of *Persons* as the set of interest. Two obvious predicates related to this domain are:

$FatherOf(x,y)$, which holds when y is the father of x
$SonOf(x,y)$, which holds when y is the son of x

Less obvious is whether we need separate predicates for brothers and sisters. One simplifying alternative is to introduce a single predicate $Siblings(x,y)$, which holds when x and y are siblings, and $x \neq y$.

Another decision we need to make is whether to introduce predicates for mothers and daughters. Neither seem to be directly needed in the riddle formalization. However, we opt for not including mothers, but include a predicate $DaughterOf(x,y)$. One reason for including daughters is that knowing that y is the father of x does not give us any information about what x is to y (since x is not necessarily y's son). With daughters in the picture we can express that x is the son or daughter of y.

Reasoning this way might seem rather arbitrary, or at best an acquired skill. This is true in the sense that creating a formal system is often an iterative process: when we hit a point where we cannot deduce what we think we should be able to, we go back and add both primitives and axioms to the system.

Axioms

We now introduce a collection of axioms for the Riddle System to clarify the properties of the predicates and sets that we have introduced.

A1: $\forall x : Persons \bullet \neg FatherOf(x,x)$
This axiom states that no one can be their own father.

A2: $\forall x : Persons \bullet \exists y : Persons \bullet FatherOf(x,y)$
This axiom states the existence of a father for every person.

A3: $\forall x,y,z : Persons \bullet FatherOf(x,y) \wedge FatherOf(x,z) \Rightarrow y = z$
This axiom states that a person cannot have more than one (biological) father.

A4: $\forall x : Persons \bullet \neg SonOf(x,x) \wedge \neg DaughterOf(x,x)$
This axiom states that no one can be their own son or daughter.

A5: $\forall x, y : Persons \bullet \neg(SonOf(x,y) \land DaughterOf(x,y))$
The axiom states that a person cannot be both someone's son and daughter.

A6: $\forall x, y : Persons \bullet FatherOf(x,y) \Leftrightarrow SonOf(y,x) \lor DaughterOf(y,x)$
This axiom characterizes the father-child relationship.

A7: $\forall x, y : Persons \bullet Siblings(x,y) \Leftrightarrow$
$x \neq y \land \exists z : Persons \bullet FatherOf(x,z) \land FatherOf(y,z)$
This axiom says that siblings are any two different persons who have the same father. (Note that since we are not including mothers in our model, we do not have similar axioms related to them.)

An important obligation when creating the axioms of a formal system is to make sure that they are consistent. That is to say, the axioms should not introduce contradictions in the system. (Otherwise we could prove anything from them.) Demonstrating the consistency of a set of axioms typically boils down to providing an interpretation of the sets and symbols for which all of the axioms are satisfied.

Another form of reasoning that we often do with respect to our axioms is to see if they are *minimal*. Informally this means that no axiom is derivable by the other axioms. For example, we might be tempted to include an axiom stating "if x is the son of y then y is the father of x." However, this is unnecessary, since the new fact can be derived from $A6$. Although minimality is not strictly required, it is often desirable, because it reduces the effort required to show that a set of axioms is consistent. Similarly, other meta-theorems that we might like to prove about our system are simplified if the set of axioms is minimal.

Derived Facts

Having defined the primitives and the axioms, the next task is to build a collection of derived facts. Derived facts can be used like any of the axioms of the Riddle System or the theorems that hold more generally for predicate logic (cf., Section 4.5). As before, the choice of which facts to derive is driven by the specific needs of the formalization. For example, as we will see below, to proof that a particular kind of relationship exists between the riddle teller and the unidentified man in the riddles above, is assisted by the introduction of a collection of lemmas or theorems.

Let us now consider some derived facts in the Riddle System.

Theorem 1. *Son-Father*

We show that

$$A1, A2, \ldots, A7 \vdash \forall x, y : Persons \bullet SonOf(y,x) \Rightarrow FatherOf(x,y)$$

See Figure 5.1 for the proof.

1.	$x \in Persons \wedge y \in Persons$	assumption
2.	$SonOf(y,x)$	assumption
3.	$\forall x, y : Persons \bullet$ $FatherOf(x,y) \Leftrightarrow SonOf(y,x) \vee DaughterOf(y,x)$	premise $A6$
4.	$FatherOf(x,y) \Leftrightarrow SonOf(y,x) \vee DaughterOf(y,x)$	$\forall$-elim, 3,1
5.	$SonOf(y,x) \vee DaughterOf(y,x) \Rightarrow FatherOf(x,y)$	$\Leftrightarrow$-elim, 4
6.	$SonOf(y,x) \vee DaughterOf(y,x)$	$\vee$-intro, 2
7.	$FatherOf(x,y)$	$\Rightarrow$-elim, 5,6
8.	$SonOf(y,x) \Rightarrow FatherOf(x,y)$	$\Rightarrow$-intro, 2–7
9.	$\forall x, y : Persons \bullet SonOf(y,x) \Rightarrow FatherOf(x,y)$	$\forall$-intro, 1–8

Figure 5.1: Proof of Theorem 1

Theorem 2. *Daughter-Father*
We show that

$$A1, A2, \ldots, A7 \vdash \forall x, y : Persons \bullet DaughterOf(y,x) \Rightarrow FatherOf(x,y)$$

The proof is similar to that of Theorem 1.

Regarding siblings we observe that the relationship is symmetric.

Theorem 3. *Symmetric Siblings*

$$A1, A2, \ldots, A7 \vdash \forall x, y : Persons \bullet Siblings(x,y) \Leftrightarrow Siblings(y,x)$$

Proof: Exercise for the reader.

Next we prove that no one can be their own sibling.

Theorem 4. *Not Own Sibling*

$$A1, A2, \ldots, A7 \vdash \forall x : Persons \bullet \neg Siblings(x,x)$$

See Figure 5.2 for the proof.

1.	$x \in Persons$	assumption
2.	$Siblings(x,x)$	assumption
3.	$\forall x,y : Persons \bullet$ $Siblings(x,y) \Leftrightarrow$ $x \neq y \wedge \exists z : Persons \bullet FatherOf(x,z) \wedge FatherOf(y,z)$	premise $A7$
4.	$x \in Persons \wedge x \in Persons$	$\wedge$-intro 1,1
5.	$Siblings(x,x) \Leftrightarrow$ $x \neq x \wedge \exists z : Persons \bullet FatherOf(x,z) \wedge FatherOf(x,z)$	$\forall$-elim, 3,4
6.	$Siblings(x,x) \Rightarrow$ $x \neq x \wedge \exists z : Persons \bullet FatherOf(x,z) \wedge FatherOf(x,z)$	$\Leftrightarrow$-elim, 5
7.	$x \neq x \wedge \exists z : Persons \bullet FatherOf(x,z) \wedge FatherOf(x,z)$	$\Rightarrow$-elim, 6,2
8.	$x \neq x$	$\wedge$-elim, 7
9.	$x = x$	eq-refl
10.	$\neg Siblings(x,x)$	$\neg$-intro, 2,8,9
11.	$\forall x : Persons \bullet \neg Siblings(x,x)$	$\forall$-intro, 1–10

Figure 5.2: Proof of Theorem 4

We also prove one more result about siblings: that two different persons that are not siblings cannot have the same father.

Theorem 5. *Different Fathers*

$$
\begin{array}{l}
A1, A2, \ldots, A7 \vdash \\
\quad \forall x,y : Persons \bullet \\
\quad\quad \neg Siblings(x,y) \wedge x \neq y \Rightarrow \\
\quad\quad\quad \forall z : Persons \bullet \neg(FatherOf(x,z) \wedge FatherOf(y,z))
\end{array}
$$

See Figure 5.3 for the proof.

Another direct consequence of $A7$ is the following theorem, which states that two different persons that have the same father are siblings.

Theorem 6. *Same Father*

$$
\begin{array}{l}
A1, A2, \ldots, A7 \vdash \\
\quad \forall x,y,z : Persons \bullet \\
\quad\quad x \neq y \wedge FatherOf(x,z) \wedge FatherOf(y,z) \;\Rightarrow\; Siblings(x,y)
\end{array}
$$

Proof: Exercise for the reader.

1.	$x \in \mathit{Persons} \wedge y \in \mathit{Persons}$	assumption
2.	$\neg \mathit{Siblings}(x,y) \wedge x \neq y$	assumption
3.	$z \in \mathit{Persons}$	assumption
4.	$\mathit{FatherOf}(x,z) \wedge \mathit{FatherOf}(y,z)$	assumption
5.	$\forall x,y : \mathit{Persons} \bullet$ $\mathit{Siblings}(x,y) \Leftrightarrow$ $x \neq y \wedge \exists z : \mathit{Persons} \bullet \mathit{FatherOf}(x,z) \wedge \mathit{FatherOf}(y,z)$	premise $A7$
6.	$\mathit{Siblings}(x,y) \Leftrightarrow$ $x \neq y \wedge \exists z : \mathit{Persons} \bullet \mathit{FatherOf}(x,z) \wedge \mathit{FatherOf}(y,z)$	$\forall$-elim, 5,1
7.	$(x \neq y \wedge$ $\exists z : \mathit{Persons} \bullet \mathit{FatherOf}(x,z) \wedge \mathit{FatherOf}(y,z)) \Rightarrow$ $\mathit{Siblings}(x,y)$	$\Leftrightarrow$-elim, 6
8.	$x \neq y$	$\wedge$-elim, 2
9.	$\exists z : \mathit{Persons} \bullet \mathit{FatherOf}(x,z) \wedge \mathit{FatherOf}(y,z)$	$\exists$-intro, 3,4
10.	$x \neq y \wedge \exists z : \mathit{Persons} \bullet \mathit{FatherOf}(x,z) \wedge \mathit{FatherOf}(y,z)$	$\wedge$-intro, 8,9
11.	$\mathit{Siblings}(x,y)$	$\Rightarrow$-elim, 7,10
12.	$\neg \mathit{Siblings}(x,y)$	$\wedge$-elim, 2
13.	$\neg(\mathit{FatherOf}(x,z) \wedge \mathit{FatherOf}(y,z))$	$\neg$-intro, 4,11,12
14.	$\forall z : \mathit{Persons} \bullet \neg(\mathit{FatherOf}(x,z) \wedge \mathit{FatherOf}(y,z))$	$\forall$-intro, 3–13
15.	$\neg \mathit{Siblings}(x,y) \wedge x \neq y \Rightarrow$ $\forall z : \mathit{Persons} \bullet \neg(\mathit{FatherOf}(x,z) \wedge \mathit{FatherOf}(y,z))$	$\Rightarrow$-intro, 2–14
16.	$\forall x,y : \mathit{Persons} \bullet$ $\neg \mathit{Siblings}(x,y) \wedge x \neq y \Rightarrow$ $\forall z : \mathit{Persons} \bullet \neg(\mathit{FatherOf}(x,z) \wedge \mathit{FatherOf}(y,z))$	$\forall$-intro, 1–15

Figure 5.3: Proof of Theorem 5

Deriving facts once the axioms of a system are identified is also useful as a sanity check: if we are unable to prove properties that we expect to hold this might be an indication that our axioms are too *weak*. For example, can you derive that "a person cannot be his father's father" in the Riddle System? Probably not. Informally the reason for not being one's own grandfather is that every father is older than his son. But there is nothing in the axioms of the Riddle System that allows us to reason in this way.

Riddle Formalization

Let us now go back to the riddles that we started with. We will formalize the first riddle in the Riddle System; moreover, we will stipulate a relationship that we expect to hold between the riddle teller and the unidentified man and try to derive

that from the formalized riddle.

Let x and y be two variables of type *Persons*, standing for the riddle teller, and the unidentified man, respectively. We formalize $R(x,y)$, where R captures the relationship between x and y as expressed in the riddle. Let us now detail R.

The first line of the riddle says that x has no brothers or sisters — in other words, x has no siblings. We formalize it as follows:

$$\forall z : Persons \bullet \neg Siblings(x,z)$$

The second line, "this man is my father's son," can be translated in several ways. For example,

$$\exists v : Persons \bullet FatherOf(x,v) \wedge SonOf(v,y)$$

expresses that one of x's father's sons is y. Alternatively, we could write:

$$\forall v : Persons \bullet FatherOf(x,v) \Rightarrow SonOf(v,y)$$

which says that if v is x's father then y must be one of v's sons. The careful reader will notice that these two sentences do not generally express the same thing in predicate logic. So why can we use them interchangeably here? The answer is that *within* the Riddle System the sentences express the same fact.

Lemma 1.

$$\begin{array}{l} A1, A2, \ldots, A7 \vdash \\ \quad \forall x, y : Persons \bullet \\ \quad\quad (\forall v : Persons \bullet FatherOf(x,v) \Rightarrow SonOf(v,y)) \Leftrightarrow \\ \quad\quad (\exists v : Persons \bullet FatherOf(x,v) \wedge SonOf(v,y)) \end{array}$$

See Figure 5.4 for the proof.

We now put together the two lines of the riddle as follows:

$$\begin{array}{l} (\forall z : Persons \bullet \neg Siblings(x,z)) \wedge \\ (\forall v : Persons \bullet FatherOf(x,v) \Rightarrow SonOf(v,y)) \end{array}$$

By now the reader has probably guessed that the solution to the puzzle is that x and y are the same person. Now we express this relationship formally and prove it in the Riddle system.

1.	$x \in Persons \wedge y \in Persons$	assumption
2.	$\forall v : Persons \bullet FatherOf(x,v) \Rightarrow SonOf(v,y)$	assumption
3.	$\forall x : Persons \bullet \exists y : Persons \bullet FatherOf(x,y)$	premise $A2$
4.	$x \in Persons$	$\wedge$-elim, 1
5.	$\exists y : Persons \bullet FatherOf(x,y)$	$\forall$-elim, 3,4
6.	$z \in Persons \wedge FatherOf(x,z)$	assumption
7.	$z \in Persons$	$\wedge$-elim, 6
8.	$FatherOf(x,z) \Rightarrow SonOf(z,y)$	$\forall$-elim, 2,7
9.	$FatherOf(x,z)$	$\wedge$-elim, 6
10.	$SonOf(z,y)$	$\Rightarrow$-elim, 8,9
11.	$FatherOf(x,z) \wedge SonOf(z,y)$	$\wedge$-intro, 9,10
12.	$\exists v : Persons \bullet FatherOf(x,v) \wedge SonOf v,y)$	$\exists$-intro, 7,11
13.	$\exists v : Persons \bullet FatherOf(x,v) \wedge SonOf v,y)$	$\exists$-elim, 5,6–12
14.	$(\forall v : Persons \bullet FatherOf(x,v) \Rightarrow SonOf(v,y)) \Rightarrow$ $(\exists v : Persons \bullet FatherOf(x,v) \wedge SonOf v,y))$	$\Rightarrow$-intro, 2–13
15.	$\exists v : Persons \bullet FatherOf(x,v) \wedge SonOf v,y)$	assumption
16.	$w \in Persons \wedge FatherOf(x,w) \wedge SonOf(w,y)$	assumption
17.	$z \in Persons$	assumption
18.	$FatherOf(x,z)$	assumption
19.	$\forall x,y,z : Persons \bullet$ $FatherOf(x,y) \wedge FatherOf(x,z) \Rightarrow y = z$	premise $A3$
20.	$x \in Persons$	$\wedge$-elim, 1
21.	$w \in Persons$	$\wedge$-elim, 16
22.	$z \in Persons$	copy from 17
23.	$x \in Persons \wedge w \in Persons \wedge z \in Persons$	$\wedge$-intro, 20,21,22
24.	$FatherOf(x,w) \wedge FatherOf(x,z) \Rightarrow w = z$	$\forall$-elim, 19,23
25.	$FatherOf(x,w)$	$\wedge$-elim, 16
26.	$FatherOf(x,w) \wedge FatherOf(x,z)$	$\wedge$-intro, 25,18
27.	$w = z$	$\Rightarrow$-elim, 24,26
28.	$SonOf(w,y)$	$\wedge$-elim, 16
29.	$SonOf(z,y)$	subst, 27,28
30.	$FatherOf(x,z) \Rightarrow SonOf(z,y)$	$\Rightarrow$-intro, 18–29
31.	$\forall v : Persons \bullet FatherOf(x,v) \Rightarrow SonOf(v,y)$	$\forall$-intro, 17–30
32.	$\forall v : Persons \bullet FatherOf(x,v) \Rightarrow SonOf(v,y)$	$\exists$-elim, 15,16–31
33.	$(\exists v : Persons \bullet FatherOf(x,v) \wedge SonOf v,y)) \Rightarrow$ $(\forall v : Persons \bullet FatherOf(x,v) \Rightarrow SonOf(v,y))$	$\Rightarrow$-intro, 15–32
34.	$(\forall v : Persons \bullet FatherOf(x,v) \Rightarrow SonOf(v,y)) \Leftrightarrow$ $(\exists v : Persons \bullet FatherOf(x,v) \wedge SonOf(v,y))$	$\Leftrightarrow$-intro, 14,33
35.	$\forall x,y : Persons \bullet$ $(\forall v : Persons \bullet FatherOf(x,v) \Rightarrow SonOf(v,y)) \Leftrightarrow$ $(\exists v : Persons \bullet FatherOf(x,v) \wedge SonOf(v,y))$	$\forall$-intro, 1–34

Figure 5.4: Proof of Lemma 1

Theorem 7. *Riddle 1 Solution*

$$\begin{array}{l}\forall x, y : \mathit{Persons} \bullet \\ \quad (\forall z : \mathit{Persons} \bullet \neg \mathit{Siblings}(x, z)) \wedge \\ \quad (\forall v : \mathit{Persons} \bullet \mathit{FatherOf}(x, v) \Rightarrow \mathit{SonOf}(v, y)) \Rightarrow \\ \qquad x = y\end{array}$$

Proof Sketch: Intuitively the argument for why x and y must be the same person is that they are children of the same father (since y is the son of x's father) and x's father has only one child (since x has no brothers or sisters). We prove that $x = y$ by contradiction: we assume that $x \neq y$. Then from the second line of the riddle we deduce that x and y have the same father. Moreover, since $x \neq y$ then they must be siblings. But now we have a contradiction: the first line says that x has no siblings. The contradiction arose from assuming $x \neq y$; therefore, x and y must be the same person.

The full proof of the theorem is presented in Figure 5.5.

Chapter Notes

[TBD]

Further Reading

[TBD]

5.10 Exercises

1. Which of the following sentences are well-formed sentences in predicate logic?

   (a) $\forall P(x) \bullet Q(x)$
   (b) $(\exists y \bullet P(y)) \wedge Q(y)$
   (c) $\forall z : T \bullet \exists x : S \bullet P(x, z)$

2. Which occurrences of the variables x and y are free and which are bound in each of the following?

1.	$x \in Persons \land y \in Persons$	assumption
2.	$(\forall z : Persons \bullet \neg Siblings(x,z)) \land$ $(\forall v : Persons \bullet FatherOf(x,v) \Rightarrow SonOf(v,y))$	assumption
3.	$x \neq y$	assumption
4.	$\forall x : Persons \bullet \exists y : Persons \bullet FatherOf(x,y)$	premise $A2$
5.	$x \in Persons$	$\land$-elim, 1
6.	$\exists y : Persons \bullet FatherOf(x,y)$	$\forall$-elim, 4,5
7.	$z \in Persons \land FatherOf(x,z)$	assumption
8.	$z \in Persons$	$\land$-elim, 7
9.	$\forall v : Persons \bullet FatherOf(x,v) \Rightarrow SonOf(v,y)$	$\land$-elim, 2
10.	$FatherOf(x,z) \Rightarrow SonOf(z,y)$	$\forall$-elim, 9,8
11.	$FatherOf(x,z)$	$\land$-elim, 7
12.	$SonOf(z,y)$	$\Rightarrow$-elim, 10,11
13.	$\forall x,y : Persons \bullet SonOf(y,x) \Rightarrow FatherOf(x,y)$	Theorem 1
14.	$y \in Persons$	$\land$-elim, 1
15.	$y \in Persons \land z \in Persons$	$\land$-intro, 14,8
16.	$SonOf(z,y) \Rightarrow FatherOf(y,z)$	$\forall$-elim, 13,15
17.	$FatherOf(y,z)$	$\Rightarrow$-elim, 16,12
18.	$\forall x,y,z : Persons \bullet$ $x \neq y \land FatherOf(x,z) \land FatherOf(y,z) \Rightarrow$ $Siblings(x,y)$	Theorem 6
19.	$x \in Persons \land y \in Persons \land z \in Persons$	$\land$-intro, 5,15
20.	$x \neq y \land FatherOf(x,z) \land FatherOf(y,z) \Rightarrow$ $Siblings(x,y)$	$\forall$-elim, 18,19
21.	$x \neq y \land FatherOf(x,z) \land FatherOf(y,z)$	$\land$-intro, 3,11,17
22.	$Siblings(x,y)$	$\Rightarrow$-elim, 20,21
23.	$\forall z : Persons \bullet \neg Siblings(x,z)$	$\land$-elim, 2
24.	$\neg Siblings(x,y)$	$\forall$-elim, 23,14
25.	$Siblings(x,y) \land \neg Siblings(x,y)$	$\land$-intro, 22,24
26.	$Siblings(x,y) \land \neg Siblings(x,y)$	$\exists$-elim, 6,7–25
27.	$Siblings(x,y)$	$\land$-elim, 26
28.	$\neg Siblings(x,y)$	$\land$-elim, 26
29.	$x = y$	$\neg$-elim, 3,27,28
30.	$(\forall z : Persons \bullet \neg Siblings(x,z)) \land$ $(\forall v : Persons \bullet FatherOf(x,v) \Rightarrow SonOf(v,y)) \Rightarrow$ $x = y$	$\Rightarrow$-intro, 2–29
31.	$\forall x,y : Persons \bullet$ $(\forall z : Persons \bullet \neg Siblings(x,z)) \land$ $(\forall v : Persons \bullet FatherOf(x,v) \Rightarrow SonOf(v,y)) \Rightarrow$ $x = y$	$\forall$-intro, 1–30

Figure 5.5: Proof of Riddle 1 Theorem 7

(a) $(\exists y : \mathbb{N} \bullet y > 2) \land (\forall x : \mathbb{N} \bullet x + 1 > x)$
(b) $x = 2 * y$
(c) $(\exists y : \mathbb{N} \bullet y > 2) \land (\forall x : \mathbb{N} \bullet x > y)$
(d) $\forall x : \mathbb{N} \bullet ((\exists y : \mathbb{N} \bullet y > x) \land x = 2 * y)$

3. Formalize the following statements using predicate logic. Define appropriate sets and predicate symbols.

(a) *"There is a green elephant."*
(b) *"The elephant, which is green, jumps over the fence."*
(c) *"The elephant that is green jumps over the fence."*
(d) *"The elephant is green and jumps over the fence."*
(e) *"The green elephants jump over the fence."*
(f) *"Elephants that jump over the fence are green."*
(g) *"All elephants are green."*
(h) *"All green elephants jump over the fence."*
(i) *"Only one elephant is green, and it jumps over the fence."*

4. **The Riddle of the Potions**

The Riddle of the Potions [5] is the last challenge faced by Harry Potter and Hermione Granger before entering the chamber where the Philosopher's Stone is kept safely hidden in the Mirror of Erised. From seven differently shaped bottles standing in a line, Harry and Hermione have to choose the one that brings them safely through the fire that blocks the door.

Danger lies before you, while safety lies behind,
Two of us will help you, whichever you would find,
S1. *One among us seven will let you move ahead*,
Another will transport the drinker back instead,
S2. *Two among our number hold only nettle wine*,
Three of us are killers, waiting hidden in line.
Choose, unless you wish to stay here forevermore,
To help you in your choice, we give you these clues four:
First, however slyly the poison tries to hide
S3. *You will ... find some [poison] on nettle wine's left side*;
Second, different are those who stand at either end,
But if you would move onward, neither is your friend;
Third, as you see clearly, all are different size,
Neither dwarf nor giant holds death in their insides;

S4. Fourth, *the second left and the second on the right*
Are twins once you taste them, though different at first sight.

(a) Translate the emphasized sentences of the riddle into predicate logic using the translation key provided below:

B	...	is the set of the seven bottles
$F(x)$	...	x holds a potion that allows us to move forward
$W(x)$	...	x holds (only) wine
$P(x)$	...	x is filled with poison (i.e. is deadly)
$L(x,y)$	...	x is the left neighbor of y

5. Show using predicate calculus the following properties:

(a) $(\forall x : T \bullet P(x) \land Q(x)) \dashv\vdash (\forall x : T \bullet P(x)) \land (\forall x : T \bullet Q(x))$
(b) $(\forall x : T \bullet P(x)) \lor (\forall x : T \bullet Q(x)) \vdash (\forall x : T \bullet P(x) \lor Q(x))$
(c) $(\exists x : T \bullet P(x)) \land Q(x)) \vdash (\exists x : T \bullet P(x)) \land (\exists x : T \bullet Q(x))$
(d) $(\exists x : T \bullet P(x) \lor Q(x)) \dashv\vdash (\exists x : T \bullet P(x)) \lor (\exists x : T \bullet Q(x))$
(e) $\neg(\forall x : T \bullet P(x)) \dashv\vdash (\exists x : T \bullet \neg P(x))$
(f) $\neg(\exists x : T \bullet P(x)) \dashv\vdash (\forall x : T \bullet \neg P(x))$

6. Assume x does not occur free in Q. Show using predicate calculus that:

(a) $(\forall x : T \bullet Q \Rightarrow P(x)) \dashv\vdash Q \Rightarrow (\forall x : T \bullet P(x))$
(b) $(\exists x : T \bullet P(x) \Rightarrow Q) \dashv\vdash (\forall x : T \bullet P(x)) \Rightarrow Q$
(c) $(\forall x : T \bullet P(x) \Rightarrow Q) \dashv\vdash (\exists x : T \bullet P(x)) \Rightarrow Q$
(d) $(\exists x : T \bullet Q \Rightarrow P(x)) \dashv\vdash Q \Rightarrow (\exists x : T \bullet P(x))$

7. Must dogs wear shoes? [4]

A sign displayed on an escalator says:

"Shoes must be worn!"
"Dogs must be carried!"

Can you explain this confusion using the formal notations covered in this chapter?

8-10. **Riddle System** (Section 5.9)

[4] Courtesy of Michael Jackson.

8. (a) Formalize each of the following riddles in the Riddle System, introducing new primitive predicates if necessary.
   i. *"Brothers and sisters have I none,*
   but this man's father is my father's son."
   ii. *"Brothers and sisters have I none,*
   but this man's son is my son."
   iii. *"Brothers and sisters have I none,*
   but this man's son is my father's son."
   iv. *"Brothers and sisters have I none,*
   but this man's father's son is my son.
   v. *"Brothers and sisters have I none,*
   but this man's father's son is my father's son."

   (b) If new primitive predicates were introduced augment the Riddle System with a set of axioms relating the newly-introduced predicates to those of the original Riddle System.

9. For each of the riddles in Exercise 8:

   (a) Characterize the relationship between the riddle teller and the unidentified man.
   (b) Prove that the relationship is a logical consequence of the formalized riddle in the (augmented) Riddle System.

10. Extend the Riddle System so that one can derive the fact that no one is their own grandfather, great-grandfather, or great-great-grandfather.

11. **Infusion Pump**

    An infusion pump is a medical device used to feed fluids intravenously to patients through one of several "infusion lines." Each line is a physical tube connected to a patient.

    Consider the following excerpt from a requirements description of a typical pump.

    (a) An infusion line may become pinched causing the flow to be blocked. This will be recognized by the pump as an occlusion and will cause the pump to alarm.
       i. The mitigation is to straighten the line and re-start the pump.
       ii. A caregiver may silence the alarm during the procedure.

(b) The infusion line may become plugged. The pump will recognize an occlusion and alarm.
   i. The mitigation is to clear the infusion lines and re-start the pump.
   ii. The caregiver may silence the alarm during the procedure.

(c) Electrical failure may occur causing the pump to switch to battery operation.
   i. The pump will switch over to battery power and notify the caregiver visually.
   ii. The switch may not occur if the battery is not properly charged.

Questions:

(1) Define some sets and predicates appropriate to this domain.
(2) Using the sets and predicates you defined express the following statements in predicate logic:
   i. An alarm sounds whenever the line is “pinched” or “plugged.”
   ii. If there is an electrical failure the battery power will be on unless the battery is not properly charged.
(3) Does your set of predicates allow you to state “The alarm will continue to sound until the care giver turns it off.” Why or why not?

Chapter 6

Structures and Relations

In Chapter 5 we described ways to talk about properties of things of interest and deduce consequences of those properties. What we need now is a way to model the "things" themselves. To do this we will usually start by introducing some named collections that represent the primitive objects in a universe of discourse. We will then build up more complex structures using operators for constructing new collections of model elements from existing ones. In this chapter we describe the building blocks for doing this using mathematical concepts such as sets, relations, functions, records, sequences, and trees.

6.1 Sets

A *set* is simply a collection of objects. Examples include the set of prime numbers, the set of positive integers, the set of countries in Europe, the set of strings of letters and numbers, and the set of possible vehicle license plate numbers in Pennsylvania.

When working with sets, we will assume that there exists a predicate, *element of*, that allows us to assert that an element is a member of a set. Notationally, we write $e \in S$, which is true when e is an element of the set S. We will abbreviate $\neg(e \in S)$ by $e \notin S$.

An element can, of course, be a member of several sets. For example, the number 3 is an element of both the sets of prime numbers and the positive integers. Similarly, "$ABC123$" is both a possible license plate number in Pennsylvania and a string of letters and numbers.

Types One important feature of our approach is that we will require that sets be homogeneous, in the sense that all their elements have the same "shape." To make this idea precise, we will associate a *type* with each element in a model, and insist that a set contain elements of only one type. This approach to sets is called *typed set theory*.

As with programming languages, the use of types has a number of engineering benefits. First, it permits us to make definitional distinctions between different kinds of elements in the universe, thereby allowing us to represent important semantic differences in the elements of systems that we are modeling. For example, we might distinguish calendar dates from employee identification numbers, even though both could in principle be represented as numbers or strings. Second, it serves as a sanity check on expressions that we write, and allows tools to help make sure that we are not writing down nonsense. For example, typing rules would prohibit the application of a function to an element that has the incorrect type. Third, it eliminates certain kinds of mathematical paradoxes that would otherwise occur in a less constrained world.

There are many possible type systems that one might use. In this book we adopt a simple scheme: the type of an object is the maximal set in which it is an element.[1] This approach has the virtue that every element e has a unique type — the largest set S for which $e \in S$.

In addition to ensuring that sets are homogeneous, the use of types also allows us to check that expressions are well-formed. For example, we can rule out expressions of the form $a = b$ if a and b do not have the same type. Similarly, we can rule out an expression of the form $e \in S$ if the type of e is not the same as the type of elements contained in S.

In the remainder of this chapter, as we introduce new ways to construct sets, we will also explain how we assign types to their elements.

Basic Sets The starting point for constructing models is to define a set of primitive, or basic, sets. A *basic set* is defined simply by providing a name for that set.

Basic sets are primitive in the sense that we have no knowledge about the internal structure of their elements. We will, however, assume that two different basic sets are *disjoint*: that is, no two basic sets have common elements. We will also assume that each basic set comes with an equality predicate that allows us to

[1] Of course, for this definition to be sound we need to be sure that such a maximal set exists. See the chapter notes for a brief discussion of this issue.

determine whether two elements in that set are the same object.

Syntactically, we declare a basic set by enclosing its name in square brackets. For example,

$[Persons]$
$[Plants]$

declare *Persons* and *Plants* to be basic sets. By virtue of the fact that they are different basic sets, we can assume that no person is also a plant, or the other way around.

We may declare more than one basic set in the same place. So, an equivalent declaration for the examples above would be

$[Persons, Plants]$

Given a basic set B and an element $x \in B$, the *type* of x is simply the set B.

The Integers We will also include one *built-in set* — the set of *integers*. Informally, this is the set containing

$$\ldots, -2, -1, 0, 1, \ldots$$

We use $\mathbb{Z}$ to denote this set.[2] We also assume that we know the standard arithmetic facts about the elements of $\mathbb{Z}$, such as the facts about addition, subtraction, less than, and so on. Equality between integers is the usual notion of numerical equality.

Variable Declarations When we want to introduce a variable that represents an element in some set, we will do it as follows:

$$x : S$$

where x is the variable name that we are introducing and S is a set or any expression that represents a set. The type of x is the type of elements in S.

Note that ':' and '$\in$' represent different concepts. The former is used to *declare* a variable, while the latter is a *predicate*, which will be true or false depending on whether the value of x is in S or not.

[2]Formally, this is a basic set for which the properties of its elements have been axiomatized in predicate logic.

Variable declarations such as this can be global (i.e., available throughout a specification), or they may occur in the context of some expression. For example, as we have seen, they appear in quantified expressions in predicate logic. We will see other examples later.

Sometimes we may wish to introduce a variable together with a constraint on its value. This can be done using an *axiomatic declaration*, which has the following form:

$$\begin{array}{|l} x : S \\ \hline P(x) \end{array}$$

Here the variable x is introduced globally together with a predicate P that the value of x must satisfy. More than one variable may be declared and more than one constraint may be defined in the same definition block. For example,

$$\begin{array}{|l} Max : \mathbb{Z} \\ Unlucky : \mathbb{Z} \\ \hline Max \geq 100 \\ Unlucky = 13 \end{array}$$

declares *Max*, whose value must be greater than or equal to 100, and *Unlucky*, whose value is 13.

Multiple predicates in axiomatic definitions are conjoined together. For example,

$$\begin{array}{|l} x : S \\ \hline P_1(x) \\ P_2(x) \end{array}$$

constrains the variable x to satisfy $P_1(x) \land P_2(x)$.

6.1.1 Set Enumeration

One way of defining a set is simply to enumerate the elements that it contains; we use curly brackets to enclose the elements. When we do this we say that the set is *defined by enumeration*. To make sure that such a set is well-defined, we must be careful to enumerate elements that are of the same type.

Example 6.1. The following are examples of sets defined by enumeration. (As in Chapters 4-5, we use the notation $S == e$ to mean that S is *by definition* the same as e: wherever we use S we could have used e.)

- $SmallEvens == \{2,4,6,8\}$
  The elements of *SmallEvens* are of type $\mathbb{Z}$.
- $Primary == \{red, green, blue\}$
  The elements of *Primary* are of type *Colors* (presumably declared elsewhere as a basic set).
- $Primes == \{2,3,5,7,11,13,\ldots\}$
  This is the set of prime numbers; its elements are of type $\mathbb{Z}$. Note that here we informally use "..." to represent that the set also contains other elements that fit the pattern observed in the elements that have been listed. We will see later how we can define such patterns formally.

□

Example 6.2. The following is *not* a well-formed set.

- $ColorNumbers == \{2,4,red,green,10,15,1\}$
  The set is not properly typed: we cannot mix color elements with number elements in the same set.

□

One common form of set enumeration is a *number range* — the set of all integers in some range. For these enumerations we use the syntactic abbreviation "$x..y$" to denote the set that includes integers from x up to (and including) y. When $y < x$ the set $x..y$ is empty.

Example 6.3.

- 1..3 represents the set $\{1,2,3\}$.
- -2..2 represents the set $\{-2,-1,0,1,2\}$.
- 0..9 represents the set of digits $\{0,1,2,3,4,5,6,7,8,9\}$.
- 1..100 represents the set of the first 100 positive integers.
- 1..0 represents the empty set of integers.

□

The notion of defining a set by enumeration can be captured formally through an axiom that tells us how to determine whether an object is a member of such a set: an object is an element of a set defined by enumeration if and only if it is equal to one of the elements of the set:

Set Membership: $a \in \{s_1, s_2, \ldots, s_n\} \Leftrightarrow a = s_1 \vee a = s_2 \vee \ldots \vee a = s_n$

6.1.2 Set Equality

A set is determined solely by its elements, in the sense that two sets are equal if and only if they have the same elements. Formally, this is expressed as the axiom

$$\text{Set Equality:} \quad S = T \;\Leftrightarrow\; (\forall x : U \bullet x \in S \Leftrightarrow x \in T)$$

where U is the type of elements of S and T. (Implicitly, because of our typing restrictions, we require that the type of elements in S and T be the same for an expression of the form $S = T$ to be well-formed.)

One consequence of the definition of set equality is that two sets are equal if they differ only in the number of times an element is enumerated in a set definition, or the order in which those elements are listed. This is because all that matters for two sets to be equal is that an element is either in both sets or in neither. For example, the sets $\{2,4\}$, $\{4,2\}$, $\{2,4,4\}$, and $\{2,4,2,4,4\}$ are equal to each other, since each contains exactly the elements 2 and 4.

Inference Rules An axiom phrased in terms of $\Leftrightarrow$, such as *Set Membership* or *Set Equality*, allows us to introduce two inference rules: one for deriving the left-hand side of $\Leftrightarrow$ from its right-hand side, and one for the other way around.[3] In the case of set equality we get the following rules: "set-eq" for deriving when two sets are equal, and "set-ext" for unfolding what it means for two sets to be equal:

$$\frac{\text{a.}\quad \forall x : U \bullet x \in S \Leftrightarrow x \in T}{S = T}\ \text{set-eq, a}$$

$$\frac{\text{a.}\quad S = T}{\forall x : U \bullet x \in S \Leftrightarrow x \in T}\ \text{set-ext, a}$$

6.1.3 Subsets

A set S is said to be a *subset* of set T, written $S \subseteq T$, if and only if every element of S is an element of T. Formally:

$$\text{Subset:} \quad S \subseteq T \;\Leftrightarrow\; (\forall x : U \bullet x \in S \Rightarrow x \in T)$$

[3] Since $\vdash P \Leftrightarrow Q$ then we also know that $\vdash P \Rightarrow Q$ and $\vdash Q \Rightarrow P$. From $\vdash P \Rightarrow Q$ and Modus Ponens, to derive Q we need a derivation of P. And that is what the following rule says

$$\frac{\text{a.}\quad P}{Q}\ \text{R1, a}$$

Similarly, we can introduce a rule from $\vdash Q \Rightarrow P$.

where U is the type of elements of S and T. T is said to be a *superset* of S.

We can also define the *superset* relationship as:

$$T \supseteq S \Leftrightarrow S \subseteq T$$

S is called a *proper subset* of T, written $S \subset T$, if $S \subseteq T$ and $\neg(S = T)$:

$$S \subset T \Leftrightarrow S \subseteq T \land \neg(S = T)$$

$T \supset S$ is defined symmetrically.

6.1.4 The Empty Set

We introduce a special symbol $\emptyset$ to represent a set with no elements.[4] The axiom for empty sets is:

$$\text{Empty Set} : \forall x : U \bullet x \notin \emptyset$$

where U is the type of the elements of the empty set in question.

One subtle point about empty sets is that we use the same symbol for empty sets of different types, and rely on context to clarify their types. For example:

- If we write $\emptyset \subseteq \{1, -3, 5\}$ we can infer that we are talking about the empty set of integers.
- If we write $\emptyset \subseteq \{red, blue\}$ we can infer that we are talking about the empty set of colors.

6.1.5 Set Cardinality

A set with a finite number of elements is said to be *finite*; a set with an infinite number of elements is said to be *infinite*. The number of elements in a finite set is called that set's *cardinality*. We write $\#S$ for the cardinality of a finite S.

Example 6.4. Some examples of set cardinality.

- $\#\{red, green, blue\} = 3$
- $\#\emptyset = 0$
- $\#\{8, -3, -2, -1, 0\} = 5$
- $\#\{1, 2, 2\} = 2$
- $\#\mathbb{Z}$ is not defined, because $\mathbb{Z}$ is not a finite set.

□

[4] Sometimes the empty set is represented by the symbols $\{\}$.

6.1.6 Set Comprehension

One particularly useful way of defining a set is to specify a property that determines which elements, drawn from some larger set, are included in the new set. We say that the set is *defined by comprehension* and write:

$$S == \{x : T \mid P(x)\}$$

where T is the set from which the elements are drawn, and $P(x)$ is the sentence in predicate logic that captures the property that each element of S must possess.

For example, the *natural numbers* are defined as the set of non-negative integers:

$$\mathbb{N} == \{x : \mathbb{Z} \mid x \geq 0\}$$

Example 6.5. The following are examples of sets defined by comprehension.

- $Digits == \{x : \mathbb{Z} \mid x > -1 \land x < 10\}$
  The digits 0..9.
- $Evens == \{x : \mathbb{N} \mid \exists y : \mathbb{N} \bullet x = 2 * y\}$
  The set of even numbers.
- $Odds == \{x : \mathbb{N} \mid x \notin Evens\}$
  The set of odd numbers.
- $NegativePrimes == \{x : Primes \mid x < 0\}$.
  (The set *Primes* was defined in Example 6.1.) The set *NegativePrimes* is empty since the predicate $x < 0$ is false for every $x : Primes$.
- $PositivePrimes == \{x : Primes \mid 0 < x\}$.
  The set *PositivePrimes* is the same as the set *Primes* since the predicate $0 < x$ is true for every $x : Primes$.
- $S == \{x : \mathbb{N} \mid 0 = 1\}$.
  S is the empty set.

□

In the definition of set comprehension x is considered a bound variable, and, as with predicate logic, can be renamed provided no variables occurring free in $P(x)$ are captured during renaming. So, for example, an equivalent definition of the natural numbers would be

$$\mathbb{N} == \{y : \mathbb{Z} \mid y \geq 0\}$$

The type of the elements in a set defined by comprehension is the type of the elements in T. Note that the set T from which elements are drawn can be any set, and not necessarily a type (i.e., a maximal set).

Example 6.6. Consider the set $\mathit{SmallNats} == \{x : \mathbb{N} \mid x < 5\}$. Since the values of x under consideration are drawn from $\mathbb{N}$, the type of elements in *SmallNats* is $\mathbb{Z}$, the type of elements in $\mathbb{N}$. □

Set comprehension introduces the following axiom:

$$\text{Set Comprehension:} \quad \forall y : U \bullet y \in \{x : U \mid P(x)\} \quad \Leftrightarrow \quad y \in U \land P(y)$$

where $P(y)$ denotes $P(x)[x \leftarrow y]$.

Example 6.7. Let us prove that if a natural number is even then it is not odd, for *Evens* and *Odds* defined as in Example 6.5. In other words, we show that $\vdash \forall x : \mathbb{N} \bullet x \in \mathit{Evens} \Rightarrow x \notin \mathit{Odds}$.

1.	$x \in \mathbb{N}$	assumption
2.	$x \in \mathit{Evens}$	assumption
3.	$\neg(x \notin \mathit{Evens})$	Double negation, 2
4.	$x \notin \mathbb{N} \lor \neg(x \notin \mathit{Evens})$	$\lor$-intro, 3
5.	$\neg(x \in \mathbb{N} \land x \notin \mathit{Evens})$	De Morgan, 4
6.	$\neg(x \in \mathit{Odds})$	Set Comprehension, 5
7.	$x \in \mathit{Evens} \Rightarrow x \notin \mathit{Odds}$	$\Rightarrow$-intro, 2–6
8.	$\forall x : \mathbb{N} \bullet x \in \mathit{Evens} \Rightarrow x \notin \mathit{Odds}$	$\forall$-intro, 1–7

Line 5 makes use of one of *De Morgan's laws*, which show how negation distributes over disjunction and conjunction. The specific law used here is $\vdash \neg(P \land Q) \Leftrightarrow \neg P \lor \neg Q$. (Cf., Section 4.5.)

Line 6 uses the Set Comprehension axiom. Our instantiation of the axiom has been somewhat informal: We have replaced $x \in \mathbb{N} \land x \notin \mathit{Evens}$ from line 5 with $x \in \mathit{Odds}$. (A more formal derivation would have required using $\forall$-elim, $\Leftrightarrow$-elim, and $\Rightarrow$-elim.) In general, for a set $\mathit{SetName} == \{x : U \mid P(x)\}$ we will use the comprehension axiom by interchanging a derivation line $y \in \mathit{SetName}$ with $y \in U \land P(y)$. □

6.2 Powerset

The *powerset* of a set T, denoted $\mathbb{P}\, T$, is the set of all of its subsets.

Example 6.8. Powerset examples.

- $\mathbb{P}\{5,0\} = \{\emptyset, \{5\}, \{0\}, \{0,5\}\}$
- $\mathbb{P}\{red, blue, yellow\} =$
  $\{\emptyset, \{red\}, \{blue\}, \{yellow\},$
  $\{red, blue\}, \{red, yellow\}, \{blue, yellow\}, \{red, blue, yellow\}\}$
- $\mathbb{P}\,\emptyset = \{\emptyset\}$

□

A set defined as a powerset of some other set cannot be empty, since it contains at least one element, the empty set. In fact, for a finite set S the powerset of S contains exactly $2^{\#S}$ elements; for this reason, powersets are sometimes denoted 2^S.

The powerset operator allows us to introduce new types into our models. If T is a set whose elements have type U, then the type of elements in the set $\mathbb{P}T$ is $\mathbb{P}U$. Thus with powersets in the picture we can define sets whose elements are sets.

Example 6.9. Well-formed sets of sets.

- $IntegerSets == \{\{-5\}, \{2,0\}, \{-12,-1,-20\}\}$
  The elements of this set are of type $\mathbb{P}\,\mathbb{Z}$.
- $PrimarySets == \{\{red, blue\}, \{blue\}, \{red, green, blue\}\}$
  The elements of this set are of type $\mathbb{P}\,Colors$.
- $PrimarySetsSets == \{\{\{green, blue\}, \{red, green, blue\}\}, \{\emptyset\}\}$
  The elements of this set are of type $\mathbb{P}(\mathbb{P}\,Colors)$.
- $SmallEvenSets == \{S : \mathbb{P}\,Evens \mid (\forall x : S \bullet x \leq 20)\}$
  This is the set of all sets of even numbers smaller than 20. The elements of this set are of type $\mathbb{P}\mathbb{Z}$.

□

Example 6.10. The following sets of sets are *not* well-formed.

- $\{2, \{3\}\}$ is badly-formed because one of its elements has type $\mathbb{Z}$ and the other has type $\mathbb{P}\mathbb{Z}$.
- $\{\{\{red, blue\}\}, \{green\}, \{blue, white\}\}$ is badly-formed because one of its elements has type $\mathbb{P}\mathbb{P}\,Colors$ and the other two have type $\mathbb{P}\,Colors$.
- $\{\{2,3\}, \{blue, yellow\}\}$ is badly formed because one if its elements has type $\mathbb{P}\mathbb{Z}$ and the other has type $\mathbb{P}\,Colors$.

□

Finite Subsets Sometimes we would like to talk about the finite subsets of a set. $\mathbb{F}\, S$ denotes the set of all *finite subsets* of S.

The powerset, and finite subset operators bind tighter than any of the other set operators (such as union, intersection, and product, which we will discuss shortly).

6.3 Generic Set Definitions

As we have already seen, we can introduce new names of sets using the following form:

$$NewName == SetExpression$$

One extension of this form allows us to define a family of such definitions using a *generic definition*:

$$NewName[Set] == SetExpression$$

For example, suppose we wish to talk about the non-empty subsets of a variety of sets. We could define this for each set as needed, but a more general way would be to declare

$$NonEmptySets[S] == \{ss : \mathbb{P}\, S \mid ss \neq \emptyset\}$$

Then, for example, $NonEmptySets[\mathbb{Z}]$ would represent the set of non-empty sets of integers, and $NonEmptySets[COLOR]$ would represent the set of non-empty sets of colors,

6.4 Union, Intersection, Difference

6.4.1 Union

The *union* of two sets S and T, denoted $S \cup T$, is the set containing exactly those elements that appear in S or in T, or in both:

$$S \cup T == \{x : U \mid x \in S \lor x \in T\}$$

where the type of the elements of both S and T is U.

Example 6.11. Examples of unions.

- For $S == \{4,5,6,7\}$ and $T == \{2,3,4,5\}$, $S \cup T = \{2,3,4,5,6,7\}$
- For $S == \{red\}$ and $T == \emptyset$, $S \cup T = \{red\}$
- $Evens \cup Odds = \mathbb{N}$
- $\mathbb{P}\,\mathbb{N} \cup \mathbb{P}\,Evens = \mathbb{P}\,\mathbb{N}$

□

Distributed Union Let S be a set of sets whose elements are of type U. The *distributed union over S* is defined as:

$$\bigcup S \;==\; \{x : U \mid \exists s : S \bullet x \in s\}$$

That is, an element is in the distributed union over a set of sets if and only if it appears in at least one of the member sets.

Example 6.12. Suppose *PrimarySets* and *PrimarySetsSets* are defined as in Example 6.9.

- $\bigcup PrimarySets = \{green, blue, red\}$
- $\bigcup PrimarySetsSets = \{\{green, blue\}, \{red, green, blue\}, \emptyset\}$
- $\bigcup(\mathbb{P}\,\mathbb{N}) = \mathbb{N}$

□

6.4.2 Intersection

The *intersection* of two sets S and T, denoted $S \cap T$, is the set containing exactly those elements that appear both in S and in T:

$$S \cap T \;==\; \{x : U \mid x \in S \land x \in T\}$$

where the type of the elements of both S and T is U.

Example 6.13. Here are some intersection examples.

- For $S == \{4,5,6,7\}$ and $T == \{2,3,4,5\}$, $S \cap T = \{4,5\}$
- For $S == \{red\}$ and $T == \emptyset$, $S \cap T = \emptyset$
- $Evens \cap Odds = \emptyset$
- $\mathbb{P}\,Evens \cap \mathbb{P}\,Odds = \{\emptyset\}$

□

Set union and intersection satisfy many useful properties. Figure 6.1 lists some of these. Note that the last two cardinality-related properties apply only to finite sets.

$S \cap T = T \cap S$	$\cap$-Commutativity
$S \cup T = T \cup S$	$\cup$-Commutativity
$S \cap \emptyset = \emptyset$	$\cap$-Empty
$S \cup \emptyset = S$	$\cup$-Empty
$(S \cap T) \cap U = S \cap (T \cap U)$	$\cap$-Associativity
$(S \cup T) \cup U = S \cup (T \cup U)$	$\cup$-Associativity
$S \cap (T \cup U) = (S \cap T) \cup (S \cap U)$	$\cap\cup$-Distributivity
$S \cup (T \cap U) = (S \cup T) \cap (S \cup U)$	$\cup\cap$-Distributivity
$\#(S \cap T) \leq \#S \ \land \ \#(S \cap T) \leq \#T$	$\cap$-Cardinality
$\#S \leq \#(S \cup T) \ \land \ \#T \leq \#(S \cup T)$	$\cup$-Cardinality

Figure 6.1: Properties of Union and Intersection

Distributed Intersection Let S be a set of sets whose elements are of type U. The *distributed intersection over S* is defined as:

$$\bigcap S \ == \ \{x : U \mid \forall s : S \bullet x \in s\}$$

Example 6.14. *PrimarySets* and *PrimarySetsSets* are defined as in Example 6.9.

- $\bigcap PrimarySets = \{blue\}$
- $\bigcap PrimarySetsSets = \emptyset$. Note that here we are talking about the empty set of sets of colors.
- $\bigcap (\mathbb{P}\,\mathbb{N}) = \emptyset$ Note that here we are talking about the empty set of integers.

□

6.4.3 Difference

The *difference* of sets S and T, denoted $S \setminus T$, is the set containing exactly those elements of S that do not appear in T:

$$S \setminus T \ == \ \{x : U \mid x \in S \land x \notin T\}$$

where the type of the elements of both S and T is U.

Example 6.15. Difference examples.

- For $S == \{4,5,6,7\}$ and $T == \{2,3,4,5\}$, $S \setminus T = \{6,7\}$
- For $S == \{red\}$ and $T == \emptyset$, $S \setminus T = \{red\}$
- $Evens \setminus Odds = Evens$
- $\mathbb{N} \setminus Evens = Odds$

□

6.5 Pairs, Tuples, and Products

Pairs and Tuples Sets allow us to define unordered collections of elements that have the same type. However, when creating models of real phenomena we need to talk about collections of elements of different types. For example, to model the information associated with some employee, we might want to include a name, date of birth, employee id, and salary. Moreover, the order in which we represent that information may be significant, for example to distinguish the employee id from the salary. To model such structures we will need to enrich our vocabulary of types of model elements.

Starting with the simplest case, *a pair*, denoted (x,y), represents an ordering of its components x and y, where x and y can be of different types. Two pairs are equal if their first components are equal and their second components are equal:

$$\text{Pair Equality:} \quad (x_1,y_1) = (x_2,y_2) \;\Leftrightarrow\; x_1 = x_2 \wedge y_1 = y_2$$

More generally, an *n-tuple*, denoted $(x_1,x_2,\ldots,x_n)$, represents an ordering of its n components. Two n-tuples are equal if their corresponding components are equal.

A 3-tuple is also called a *triple*; similarly, a 4-tuple is also called a *quadruple*.

6.5.1 Cartesian Product

We can also create sets of tuples. The *Cartesian product* of two sets S and T, denoted $S \times T$, is the set of all pairs with first component from S and second component from T.

Example 6.16. Examples of Cartesian products using two sets.

- Let $S == \{1,3,5\}$ and $T == \{\text{“}a\text{”},\text{“}b\text{”}\}$
  $S \times T = \{(1,\text{“}a\text{”}),(1,\text{“}b\text{”}),(3,\text{“}a\text{”}),(3,\text{“}b\text{”}),(5,\text{“}a\text{”}),(5,\text{“}b\text{”})\}$
- Let $S == \{\{1,2\},\{3\}\}$ and $T == \{-1,-2\}$
  $S \times T = \{(\{1,2\},-1),(\{1,2\},-2),(\{3\},-1),(\{3\},-2)\}$
- Let $S == \{11,12\}$ and $T == \emptyset$, $S \times T = \emptyset$.

□

For finite sets S and T, if $\#S = m$ and $\#T = n$ the Cartesian product, $S \times T$, has $m * n$ elements. If either of the sets are infinite, the product is also an infinite set.

Cartesian products allow us to to define the types of tuples. If s has type U and t has type V, then the tuple (s,t) has type $U \times V$. In other words, if S is a set

whose elements have type U, and T a set whose elements have type V, then the elements of $S \times T$ have type $U \times V$.

Generalizing, the Cartesian product of n sets $S_1, S_2, \ldots, S_n$, denoted $S_1 \times S_2 \times \ldots \times S_n$, is a set of n-tuples $(x_1, x_2, \ldots, x_n)$ where $x_1 \in S_1, x_2 \in S_2, \ldots, x_n \in S_n$. If the elements of set S_i have type U_i (for $1 \leq i \leq n$) then the type of the elements of $S_1 \times S_2 \times \ldots \times S_n$ is $U_1 \times U_2 \times \ldots \times U_n$.

Example 6.17. Let $S_1 == \{red, green\}$, $S_2 == \{3\}$, and $S_3 == \{Paul, Ron\}$

$$S_1 \times S_2 \times S_3 = \{(red, 3, Paul), (red, 3, Ron), (green, 3, Paul), (green, 3, Ron)\}$$

□

6.6 Relations and Functions

When modeling a system it is often necessary to describe the relationships between various types of model elements. As we will see, this can be done using the mathematical building blocks that we have already presented.

6.6.1 Binary Relations

A *binary relation*, or simply *relation*, R between two sets S and T is a set of pairs from $S \times T$. That is to say, $R \subseteq S \times T$, or equivalently $R \in \mathbb{P}(S \times T)$. S and T are said to be R's *source* and *target*, respectively.

As an example, consider a database of car owners that relates cars (identified using vehicle numbers) and their owners. If vehicle numbers are drawn from the natural numbers, $\mathbb{N}$, and $[Persons]$ is the set of persons, an entry in the car-owner database is an element of $\mathbb{N} \times Persons$. As a set of number-person pairs, the database itself is a subset of $\mathbb{N} \times Persons$:

$$Owners \subseteq (\mathbb{N} \times Persons)$$

Another way to say the same thing is:

$$Owners \in \mathbb{P}(\mathbb{N} \times Persons)$$

As a shorthand, we introduce $S \leftrightarrow T$ for $\mathbb{P}(S \times T)$, and we write:

$$Owners \in \mathbb{N} \leftrightarrow Persons$$

An example of a car-owner database could be:

$$Owners == \{(1234, John\ S.), (3251, Peter\ M.), (5132, Mary\ P.)\}$$

If $(a,b) \in R$ we say that R *maps a to b*. An alternative notation for a relational pair (a,b) is $a \mapsto b$. So, the car-owner database above could be written equivalently as:

$$Owners == \{1234 \mapsto John\ S., 3251 \mapsto Peter\ M., 5132 \mapsto Mary\ P.\}$$

(Note that when using the "map" notation we do not enclose the map expression in parentheses: that is, we use $a \mapsto b$ and not $(a \mapsto b)$.)

We assume that "$\leftrightarrow$" associates to the right, so that $S \leftrightarrow T \leftrightarrow U$ is interpreted as $S \leftrightarrow (T \leftrightarrow U)$.

Domain and Range We identify two important sets in connection with a relation $R : S \leftrightarrow T$. Its *domain*, denoted $dom(R)$, is the set of values from S that appear as the first element of some pair in R. Its range, denoted $ran(R)$, is the set of values from T that appear as the second element of some pair in R. Formally, the domain and range of a relation are defined as follows:

$$\begin{array}{lcl} dom(R) & == & \{x : S \mid \exists y : T \bullet (x,y) \in R\} \\ ran(R) & == & \{y : T \mid \exists x : S \bullet (x,y) \in R\} \end{array}$$

For example, for the car-owner database defined above:

$$\begin{array}{lcl} dom(Owners) & = & \{1234, 3251, 5132\} \\ ran(Owners) & = & \{John\ S., Peter\ M., Mary\ P.\} \end{array}$$

Notice that although $dom(R) \subseteq S$, it may be the case that $dom(R)$ is not the same as S, since there can be elements of S that do not appear as the first element of *any* pair in R. Similarly for the range and target.

Example 6.18. Consider the relation *Div2* defined as follows

$$Div2 == \{(x,y) : \mathbb{N} \times \mathbb{N} \mid x = 2y\}$$

Its elements are the pairs $(2,1), (4,2), (6,3)$.... The elements are of type $\mathbb{Z} \times \mathbb{Z}$. $\mathbb{N}$ is both the source and the target of the relation. The domain of the relation is *Evens*, and the range of the relation is $\mathbb{N}$. □

Domain and Range Restriction There are a number of operators that allow us to "filter" elements of a binary relation. The *domain restriction* operator, denoted $\lhd$, is defined as follows:

$$S_1 \lhd R == \{(x,y) : S \times T \mid (x,y) \in R \land x \in S_1\}$$

where S_1 has the same type as S. Informally, the restricted relation contains those elements from R whose first component appears in S_1.

Similarly, we define the *range restriction* operator, denoted $\rhd$, as:

$$R \rhd T_1 == \{(x,y) : S \times T \mid (x,y) \in R \land y \in T_1\}$$

where T_1 has the same type as T. Informally, the restricted relation contains those elements from R whose second component appears in T_1.

Example 6.19. Let

$$R == \{(2,red),(5,blue),(2,yellow),(3,red),(5,pink),(4,azure)\}$$
$$Primary == \{red,blue,green\}$$
$$Evens == \{x : \mathbb{N} \mid \exists y : \mathbb{N} \bullet x = 2 * y\}$$

Then

$$Evens \lhd R = \{(2,red),(2,yellow),(4,azure)\}$$
$$R \rhd Primary = \{(2,red),(5,blue),(3,red)\}$$

□

6.6.2 *n*-ary Relations

We can generalize the notion of a binary relation. An *n-ary relation R* is a subset of $S_1 \times S_2 \times \ldots \times S_n$.

Example 6.20. Examples of *n*-ary relations.

- A Pythagorean triple consists of three positive integers that can be the lengths of the sides of a right triangle. The set of all such triples can be defined as:

  $$Pythagorean == \{(a,b,c) : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \mid a^2 + b^2 = c^2\}$$

  For example, $(3,4,5) \in Pythagorean$.

- n-degree polynomials can be represented as $(n+1)$-ary relations.

  $$\mathit{5DegreePolynomials} \in \mathbb{P}(\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z})$$

  $4x^5 - 2x^4 + 7x^3 + 13x^2 - 3x + 8$ is an example of a 5-degree polynomial, and its representation $(4, -2, 7, 13, -3, 8)$ is an element of *5DegreePolynomials*.
- Birth certificates can be modeled as n-ary relations, with each person's name being associated with the date, place, and parents to whom they are born.

  $$\mathit{BirthCertificates} \in \mathbb{P}(\mathit{Persons}, \mathit{Dates}, \mathit{Places}, \mathit{Persons}, \mathit{Persons})$$

  As an example, consider John Smith's birth certificate
  $(\mathit{John\ S.}, 1\text{-}\mathit{Sep}\text{-}1973, \mathit{Portland}, \mathit{Mary\ S.}, \mathit{Joe\ S.}) \in \mathit{BirthCertificates}$.

□

Sometimes it is convenient to consider an n-ary relation $r_n : \mathbb{P}(S_1 \times S_2 \times \ldots \times S_n)$ as a binary relation $r_2 : (S_1 \times S_2 \times \ldots \times S_{n-1}) \leftrightarrow S_n$. For example, it may be convenient to work with terms like $(a,b,c) \mapsto 5$ (an element of r_2) instead of $(a,b,c,5)$ (an element of r_4).

Whenever we do so, we assume we are working with a relationship r_2 equivalent to r_n in the following sense:

$$\begin{array}{l} \forall s_1 : S_1;\ s_2 : S_2;\ \ldots;\ s_n : S_n \bullet \\ \quad ((s_1, s_2, \ldots, s_{n-1}), s_n) \in r_2 \quad \Leftrightarrow \quad (s_1, s_2, \ldots, s_n) \in r_n \end{array}$$

6.6.3 Functions

A *partial function* from S to T is a binary relation $f : S \leftrightarrow T$ such that f maps an element of S to *at most one* element T:

$$\forall x : S;\ y_1, y_2 : T \bullet (x, y_1) \in f \land (x, y_2) \in f \Rightarrow y_1 = y_2$$

We use the notation $f(x) = y$ when there is a y such that $(x,y) \in f$ and then say that $f(x)$ is *defined*; otherwise we say that $f(x)$ is *undefined*.

A *total* function from S to T is a partial function f from S to T such that $f(x)$ is defined for all $x \in S$. In other words, $\mathit{dom}(f) = S$.

Although a total function is a special kind of partial function it is customary to use the word *function* to mean a total function. We then say explicitly when we are dealing with partial functions.

Example 6.21. The following relations are functions.

- $f == \{red \mapsto 2, green \mapsto 4, blue \mapsto 6\}$
  If f is considered to map *Primary* to $\mathbb{N}$, where $Primary == \{red, green, blue\}$ then f is a total function. However, if f maps *Colors* to $\mathbb{N}$ (where $Primary \subset Colors$) then f is a partial function, since some colors are not mapped to numbers by f.
- $g == \{1 \mapsto 0, 2 \mapsto 1, 3 \mapsto 1, 4 \mapsto 2, \ldots\}$
  This function maps a positive integer x to $x/2$ where "/" is the integer division operator. g is a partial function from $\mathbb{Z}$ to $\mathbb{Z}$ — g is undefined for negative numbers and 0. Notice that some elements are mapped to the same element. For example, both 3 and 2 map to 1.

□

Terminology and Notation We introduce the following terminology and notation for functions:

- We write $f : S \to T$ for a total function from S to T.
- We write $f : S \nrightarrow T$ for a partial function from S to T.
- We say that a function f is *finite* if its domain is a finite set.We write $f : S \twoheadrightarrow T$ for a finite function.
- We say that a function f from S to T is *injective* or *one-to-one* if no two elements of S are mapped to the same element in T. We write $f : S \rightarrowtail T$.
- We say that a function f from S to T is *surjective* if $ran(f) = T$. We write $f : S \twoheadrightarrow T$.
- We say that a function f is *bijective* if it is both injective and surjective. We write $f : S \rightarrowtail\!\!\!\!\twoheadrightarrow T$.

Figure 6.2 depicts a graphical representation of the different classes of function and their relationships to each other.

We assume that function symbols associate to the right, so that, for example, $S \to T \to U$ is interpreted as $S \to (T \to U)$. Sometimes we will drop the parentheses in expressions involving function application. For example, we can write $f\,x$ instead of $f(x)$.

6.6.4 Composing Relations and Functions

Two binary relations $R_1 : S \leftrightarrow T$ and $R_2 : T \leftrightarrow U$ can be composed, so that elements in the domain of R_1 are mapped to elements in the range of R_2, provided there is

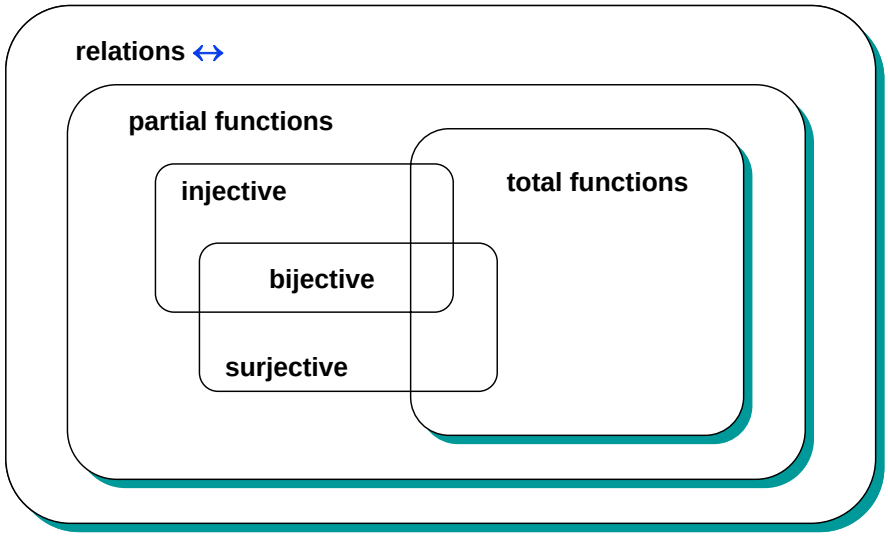


Figure 6.2: Functions

a common intermediary value in the range of R_1 and domain of R_2. Formally, *relational composition* is defined as follows:

$$(R_1; R_2) \;==\; \{(x,z) \in S \leftrightarrow U \mid \exists y : T \bullet (x,y) \in R_1 \land (y,z) \in R_2\}$$

An alternative notation, referred to as "backward relational composition," is sometimes used: we can write $R_2 \circ R_1$ instead of $R_1; R_2$. (Note that the order is reversed.)

Relational composition is associative:

$$\vdash (R_1; R_2); R_3 = R_1; (R_2; R_3) \quad \textit{and} \quad \vdash (R_3 \circ R_2) \circ R_1 = R_3 \circ (R_2 \circ R_1)$$

Since functions are also relations we can use relational composition to compose them, provided their types match up appropriately. In particular,

$$(g; f)(x) = (f \circ g)(x) = f(g(x))$$

Example 6.22. Figure 6.3 shows an example of relational composition. R_1 maps letters from the alphabet to numbers, which are in turn mapped to colors by R_2. The composition $R_1; R_2$ maps the letters directly to the colors. □

Example 6.23. Recall the car-owner database that we defined above (Section 6.6.1). Now consider the relation that maps persons to driver's license numbers, which

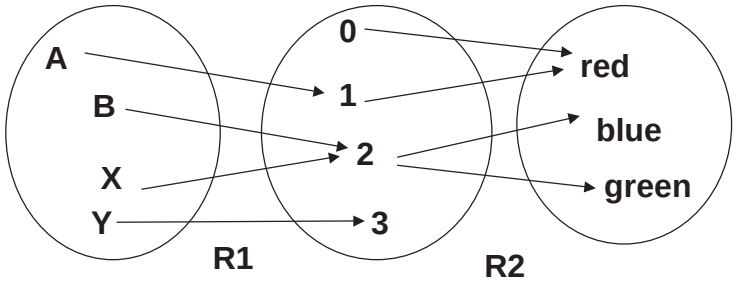


Figure 6.3: Example of Relational Composition

we denote by *Licenses*. Composing *Owners* with *Licenses* would give us the relation that maps vehicle numbers to the driver's license numbers of their owners. □

Example 6.24. Pipes and Filters. Relational composition can be thought of as modeling a "pipe-and-filter" style of computation: a filter represents a computing unit that transforms its inputs according to a relation. The outputs from the first filter are piped into a second filter, which also transforms the data according to another relation, and so on. For n filters, the initial inputs are then related to the final inputs according to $R_1; R_2; \ldots; R_n$. □

6.6.5 Defining Relations and Functions Axiomatically

As we have illustrated thus far, we can define a relation or a function in a variety of ways: using set enumeration, set comprehension, and so on. We can also define a function using an axiomatic declaration introduced in Section 6.1.

Consider the square function, which we might informally define as follows:

$$square(x) = x^2$$

Defining this axiomatically we would have:

$$\begin{array}{|l} square : \mathbb{N} \to \mathbb{N} \\ \hline \forall x : \mathbb{N} \bullet square(x) = x^2 \end{array}$$

Similarly, relations can be defined axiomatically, where the predicate specifies when two elements are in the defined relation. For example, the "integer square root relation" can be defined as follows:

$$\begin{array}{|l} root : \mathbb{Z} \leftrightarrow \mathbb{Z} \\ \hline \forall x, y : \mathbb{Z} \bullet (x, y) \in root \Leftrightarrow (x^2 = y) \end{array}$$

When defining functions and relations in this way we can also indicate that they are to be treated as an infix operator. For example, when defining the superset relation $\supset$ over sets we can declare that it is an *infix* operator by using the notation $\_ \supset \_$ as follows:

$$\begin{array}{|l} \_ \supset \_ : \mathbb{P} X \leftrightarrow \mathbb{P} X \\ \hline \forall S, T : \mathbb{P} X \bullet S \supset T \Leftrightarrow S \subset T \end{array}$$

So writing $S \supset T$ means $(S, T) \in \supset$.

6.7 Records

Another useful structure when creating models is one that allows us to keep track of mixed types of information (as we do with tuples), but also allows us to identify each part using a label, rather than its position in some ordering. For example, we might like to model information associated with an employee, such as social security number, salary, and date of employment. But rather than listing these in some fixed order we will refer to each part of the data record using an appropriate label.

This can be achieved with *records*, a construct familiar to programmers. Records are similar to tuples, but their components have names. We refer to the component names as *fields*. For example, a sample employee record might be

$$[ssn = 123456789;\ salary = 50000;\ startDate = 5/16/02]$$

which has fields *ssn*, *salary*, and *startDate*.

In contrast to tuples, the order of fields in a record does not matter, so that

$$[salary = 55000;\ ssn = 123456789;\ startDate = 5/16/02]$$

is the same as the previous record. That is to say, two records are equal if the values of their fields are the same.

The value of an individual field in a record can be accessed using the familiar "dot" notation. For example, if x refers to the record defined above, then $x.ssn = 123456789$, $x.salary = 55000$, and $x.startDate = 5/16/02$.

We can define a set of records using the following form:

$$RecordSet == [f_1 : S_1 ; f_2 : S_2 ; \ldots ; f_n : S_n]$$

This defines the set of records with fields $f_1, f_2, \ldots, f_n$, where the values are drawn from the sets $S_1, S_2, \ldots, S_n$, respectively. If any of the S_i are empty the set of records is also empty. The type of a record from this set is denoted

$$[f_1 : U_1 ; f_2 : U_2 ; \ldots ; f_n : U_n]$$

where U_i is the type of elements of S_i.

For example, the set of employee records might be defined as follows:

$$EmployeeInfo == [ssn : 9DigitNats ; salary : \mathbb{N} ; startDate : Date]$$

where $9DigitNats$ is the set of natural numbers with exactly nine digits. The type of a record in this set would be

$$[ssn : \mathbb{Z} ; salary : \mathbb{Z} ; startDate : Date]$$

6.8 Recursive Structures

In this section we describe how to model certain structures recursively. The basic idea is to define rules that show how such structures can be composed of "simpler" structures of the same kind.

6.8.1 Trees

Consider binary trees. A recursive definition for simple binary trees would be

$$TREE ::= leaf \mid node\langle\langle TREE \times TREE \rangle\rangle$$

This definition introduces a new type, *TREE*, for which any element of the type is either a *leaf* or it is made up of two subtrees glued together with a *node*. We refer to *leaf* and *node* as tree *constructors*.

To write down a particular element of a recursive type we treat the non-parameterized constructors of the recursive definition as constants, and the parameterized constructors as functions. For example, here are some *TREE* instances:

$$leaf$$
$$node(leaf, leaf)$$
$$node(node(leaf, leaf), leaf)$$
$$node(leaf, node(node(leaf, leaf), leaf))$$

A recursive structure can have any number of constructors. For example consider a kind of tree that can have both binary as well as ternary nodes:

$$\begin{aligned} &\mathit{MIXEDTREE} ::= \mathit{mixleaf} \\ &\qquad \mid \mathit{binnode}\langle\langle \mathit{MIXEDTREE} \times \mathit{MIXEDTREE} \rangle\rangle \\ &\qquad \mid \mathit{ternnode}\langle\langle \mathit{MIXEDTREE} \times \mathit{MIXEDTREE} \times \mathit{MIXEDTREE} \rangle\rangle \end{aligned}$$

Examples of mixed trees include:

$$\begin{aligned} &\mathit{mixleaf} \\ &\mathit{ternnode}(\mathit{binnode}(\mathit{mixleaf}, \mathit{mixleaf}), \mathit{ternnode}(\mathit{mixleaf}, \mathit{mixleaf}, \mathit{mixleaf}), \mathit{mixleaf}) \end{aligned}$$

6.8.2 Enumerated Types

A special case of a recursive structure occurs when all of the constructors are constants. Here are some examples:

$$\begin{aligned} &\mathit{Yes\_or\_No} ::= \mathit{YES} \mid \mathit{NO} \\ &\mathit{Primary\_Color} ::= \mathit{Red} \mid \mathit{Blue} \mid \mathit{Green} \\ &\mathit{Error} ::= \mathit{Overflow} \mid \mathit{Underflow} \mid \mathit{Div\_by\_Zero} \end{aligned}$$

Enumerations, such as these, guarantee that every value of the type must be exactly one of the distinct choices provided by the enumeration.

6.8.3 Engineering Considerations

You may notice that the recursive structure definitions resemble the rules of a grammar. Indeed, a grammar can be thought of as specifying recursively a set of well-formed formulas that define a formal language.

Another important connection to note is the relationship between recursive structure definitions and operational views of data structures. One general principle used in software engineering is the idea of specifying a type of data structure solely in terms of the operations that are used to create or modify instances of that type. This principle allows users of the data structure to ignore the internal representation of its data, and thereby permits modification of those representations without affecting its users.

As a simple example, we can think of a stack as an entity that is defined by operations *new*, *push*, and *pop*. Any instance of a stack can be represented as an expression involving these operations. For example, a stack with the elements 3

and 5 can be represented described as $(push(5,(push(3,new)))$. How the elements are stored internally is irrelevant to the user of the stack.

Beyond simple data structures, the general principle of characterizing an entity in terms of an interface specification is ubiquitous throughout software engineering. It represents one of the key ideas behind object-oriented programming, component-based systems, peer-to-peer computing, service-oriented architectures, and many other software engineering paradigms.

When encountering some phenomena that you would like to model, the decision of whether to use a recursive structure will often be dictated by the nature of the entities involved and the kind of reasoning that you would like to do. In some cases, such as trees, a recursive definition will be the most natural. In others, directly representing it in terms of other modeling structures (e.g., sets, relations, etc.) will be preferable. In some cases, as we will see shortly, both ways are commonly used – each offering certain advantages.

6.9 Sequences

Sequences are used to model ordered collections of objects. They are typically used to model queues, temporal orderings (such as the history of states of system), and indexed lists of elements. Unlike tuples and records, which have fixed length, the length of a sequence is variable. For example, one may append new elements to a sequence or concatenate two sequences together.

Instances of a sequence are denoted using angle brackets, for example, $\langle 4,2,8\rangle$. The *empty sequence* is the sequence with no elements, and is denoted by $\langle\rangle$.

In contrast to tuple types, which allow for elements of a tuple to have different types, the elements of a sequence must be of the same type. For example, $\langle\{a\},\emptyset,\{b,c\},\{a\}\rangle$ is a valid sequence – its elements are drawn from $\mathbb{P}\{a,b,c\}$ – whereas $\langle a,\{b\}\rangle$ is not a valid sequence. An element can occur more than once in a sequence; for example sequence $\langle b,a,a,c\rangle$ is different from sequence $\langle b,a,c\rangle$.

In the remainder of this section we describe two ways of modeling sequences: as relations and defined recursively. In addition to providing a useful modeling abstraction in its own right, this will help to illustrate how the various modeling concepts introduced in this chapter can be applied to create new kinds of modeling structures.

6.9.1 A Relational Model for Sequences

One way to model finite sequences containing elements from a set X is as partial functions from natural numbers to the set X. Specifically, we denote the set of finite sequences over a set X by $\text{seq}[X]$, defined as follows:

$$\text{seq}[X] \;==\; \{s : \mathbb{N} \pfun X \mid \exists n : \mathbb{N} \bullet dom(s) = 1..n\}$$

Here we use a generic definition (Section 6.3) to define a sequence constructor parameterized by the set of elements from which its members will be drawn. For example, $\text{seq}[\mathbb{Z}]$ is the set of sequences of integers, and $\text{seq}[\{0,1\}]$ is the set of sequences of binary numbers.

Empty sequences arise when $n = 0$ in the above definition, since we interpret $1..0$ as the empty set over natural numbers. Because sequences are simply functions, we can address individual elements by applying the function to the appropriate index: for every index i in the domain of a sequence s (that is, $1 \leq i \land i \leq n$), $s(i)$ corresponds to the i^{th} element of the sequence. For example, $\langle b,a,c,a\rangle(2) = a$.

There are a number of useful operators over sequences, summarized in Figure 6.4. These can be defined using axiomatic definitions (Section 6.6.5). For example, *length* and *concatenation* operators for sequences can be defined as follows:

$$\begin{array}{|l}
\# : \text{seq}[X] \pfun \mathbb{N} \\
_ \frown _ : \text{seq}[X] \times \text{seq}[X] \to \text{seq}[X] \\
\hline
\forall s : \text{seq}[X] \bullet \#s = \#(dom(s)) \\
\forall s,t : \text{seq}[X];\ i : dom(s);\ j : dom(t) \bullet \\
\qquad (s \frown t)(i) = s(i) \;\land\; (s \frown t)(\#s + j) = t(j) \\
\end{array}$$

The length of a sequence is defined to be the cardinality of the sequence's domain. (This is well-defined because we are dealing with finite sequences.) The concatenation operator appends the elements of the second sequence to the first sequence and adjusts their indexes accordingly.

6.9.2 A Recursive Model for Sequences

Another way to model finite sequences is as recursive structures. For example, we can define sequences over natural numbers as follows:

$$SEQ ::= \langle\rangle \mid cons\langle\langle \mathbb{N} \times SEQ\rangle\rangle$$

$head : \text{seq}[X] \nrightarrow X$	First element of a nonempty sequence.
$tail : \text{seq}[X] \nrightarrow X$	Subsequence after first element of a nonempty sequence.
$\# : \text{seq}[X] \nrightarrow \mathbb{N}$	Length of sequence.
$\frown : \text{seq}[X] \times \text{seq}[X] \rightarrow \text{seq}[X]$	Appends two sequences together.

Figure 6.4: Operations on Sequences

That is, a sequence is either empty or is formed by adding a natural number to the front of another sequence.

To avoid clutter we will pretty-print *cons* as an infix operator ' :: '. For example, $cons(n,s)$ will be written $(n :: s)$ for a number n and sequence s. Moreover, we let :: associate to the right, so $3 :: 4 :: 5 :: \langle\rangle$ means $3 :: (4 :: (5 :: \langle\rangle))$.

Example 6.25. These sequences are well-formed:

- $\langle\rangle$
- $1 :: (2 :: \langle\rangle)$
- $7 :: 10 :: \langle\rangle$
- $3 :: (6 :: (1 :: (2 :: \langle\rangle)))$

□

Example 6.26. These sequences are badly-formed:

- $\langle 3\rangle :: \langle\rangle$
- $(\langle\rangle :: 1) :: 4$

□

In order to define operations over such sequences, we need to describe the effect of the operator for each construct. How we do that, and also how we reason about structures defined recursively will be detailed in Chapter 7.

6.10 Specifying Models

Let us recap where we are. So far we have introduced a variety of ways to define mathematical structures. These are summarized in Figure 6.5. The question to address now is "How do we use all of these as building blocks for modeling a system?"

The answer to this question depends on a variety of things, including the kind of system to be modeled, the kinds of properties of that system that we care about,

Ways of defining different types of structures:	
Given sets	$[ANIMALS, VEGETABLES, MINERALS]$
Powersets	$\mathbb{P} S$
Products	$S_1 \times S_2 \times \ldots \times S\_n$
Records	$[x_1 : S_1; x_2 : S_2; \ldots x_n : S_n]$
Recursive structures	
Ways to create new sets from existing sets:	
By enumeration	$\{el_1, el_2, \ldots, el_n\}$
By comprehension	$\{x : S \mid P(x)\}$
Using set operators (union, ...)	
Ways to introduce new names:	
Definitions	$==$
Generic definitions	setname$[X] == \ldots$
Variable declarations	$x : T$
Axiomatic definitions	

Figure 6.5: Summary of Structures

the amount of effort that we want to put into defining a model, the nature of the tools that we have at our disposal for automating the specification and analysis process, expectations about ways that the model may need to be changed in the future, and the existence of other models, and theories that we may wish to build on.

However, despite these differences, there are a number of steps that are typically followed to design a specification of a model.

The first step is to decide what kinds of entities are going to be included in the model. This can often be done without yet knowing *how* those entities are going to be modeled. For example, if the system is intended to support secure access to documents, we are likely to have entities such as documents, people, passwords, access rights, audit trails, etc.

The second step is to define what kinds of entities are to be primitive. These will typically be modeled as given sets. For example, we might decide that passwords are primitive entities, but documents will require more structured representations.

Next we need to define the more complex elements using given sets and the type constructors that we have discussed in this chapter and the relationships

among those elements. This is usually the hard part. In the end, it typically involves thinking through the design of the specification in terms of the constraints over the possible states of the the model, the kinds of properties that are important, and the kinds of reasoning that you would like to perform. Typically this is an iterative process: early models may turn out to be too complex or not detailed enough. Or, in the process of specification we may discover new properties and constraints that are relevant.

In the process of creating our model, it will also be important to document it, by adding in prose that explains the terminology and the rationale behind the choice of model elements and properties that are included.

[Future versions will include an example that illustrates the process and techniques.]

Chapter Notes

[TBD]

Further Reading

[TBD]

6.11 Exercises

1. Define the following sets by enumeration. What is the type of the elements of each set?

   (a) The set of names representing the months of the year.
   (b) The set of the seven dwarfs (from Disney's "Snow White and the Seven Dwarfs").
   (c) The set of the first five positive cubes.
   (d) The set of the perfect numbers smaller than 20. (A perfect number is equal to the sum of its full divisors.)

2. Define the following sets by comprehension. Is it possible to define these sets by enumeration?

   (a) The set of numbers divisible by 3.

(b) The set of full divisors of number 20.
(c) The set of full divisors of a natural number n.
(d) The set of palindromes of length 3. (A palindrome is a string of characters that reads the same in either direction.)
(e) The set of all positive numbers a, b, c that satisfy $a^n + b^n = c^n$ for a given $n > 2$.

3. Let $S == \{x : \mathbb{N} \mid x \leq y\}$. Since x is considered a bound variable, we can rename it to, for example, z (because z does not appear free in $x \leq y$). What would the problem be if we renamed x to y? Which set would we get in that case? Is that set equal to S?

4. Compute the powersets of each of the following.

(a) $\{0, 1\}$
(b) $\{5\}$
(c) $\{\emptyset\}$
(d) $\{\{\emptyset\}\}$

5. Compute the union, intersection, and difference of the following sets.

(a) $S_1 = \{(a,b),(b,c),(c,d)\}$ and $T_1 = \{(b,a),(c,b)\}$.
(b) $S_2 = \{a : \mathbb{N} \mid a \leq 5\}$ and $T_2 = \{a : \mathbb{N} \mid 4 \leq a\}$.
(c) $S_3 = \{a : \mathbb{Z} \mid \exists b : \mathbb{Z} \bullet a = 15b\}$ and $T_3 = \{a : \mathbb{Z} \mid \exists b : \mathbb{Z} \bullet a = 10b\}$.
(d) $S_4 = \{(a,b) : \mathbb{N} \times \mathbb{N} \mid a * b \leq a^2\}$ and $T_4 = \{(a,b) : \mathbb{N} \times \mathbb{N} \mid a * b \leq b^2\}$.

6. Compute the distributed union, and intersection of the following sets.

(a) $S_1 = \{5,6,7\}$, $S_2 = \{6,7,8,9\}$, $S_3 = \{7,8,9,10,11\}$, and $S_4 = \{8,9,10,11,12,13\}$.
(b) $S_i = \{(a,b) : \mathbb{N} \times \mathbb{N} \mid a * b \leq i\}$, for $1 \leq i \leq 5$.
(c) $S_i = \{a : \mathbb{N} \mid \exists b : \mathbb{N} \bullet i = a * b\}$, for $1 \leq i \leq 5$.
(d) $T_i = \mathbb{P} S_i$, for $1 \leq i \leq 3$, where S_i is as defined in 6c.
(e) $S_i = \{i\}$ for $i \in \mathbb{N}$.

7. (a) Show $(A \subseteq B) \wedge (B \subseteq A) \Leftrightarrow A = B$.
(b) Show $(A \subset B) \Rightarrow B \neq \emptyset$.
(c) Show $A \setminus B \subseteq A$.
(d) Show that set difference is not commutative.

(e) Show that union and intersection are *idempotent*:

$$A \cup A = A$$
$$A \cap A = A$$

8. Give examples of two sets S and T such that

(a) $\#(S \cap T) < \#S$ and $\#(S \cap T) < \#T$
(b) $\#S < \#(S \cup T)$ and $\#T < \#(S \cup T)$

9. Write out in full the following cartesian products.

(a) $\{0,1\} \times \{0,1\}$
(b) $\emptyset \times \emptyset$
(c) $\{1,2\} \times \{a\}$
(d) $\{\emptyset\} \times \{a\}$

10. **Properties of Cartesian Products**

Prove the following properties of the Cartesian product of two sets.

(a) Distributivity of $\times$ over $\cup$:
  i. $S \times (T \cup U) = (S \times T) \cup (S \times U)$
  ii. $(S \cup T) \times U = (S \times U) \cup (T \times U)$
(b) Distributivity of $\times$ over $\cap$:
  i. $S \times (T \cap U) = (S \times T) \cap (S \times U)$
  ii. $(S \cap T) \times U = (S \times U) \cap (T \times U)$
(c) Distributivity of $\times$ over $\setminus$:
  $S \times (T \setminus U) = (S \times T) \setminus (S \times U)$
(d) Monotonicity:
  i. $T \subseteq U \Rightarrow S \times T \subseteq S \times U$
(e) $S \subseteq U \land T \subseteq V \Rightarrow S \times T \subseteq U \times V$
(f) $S \times T \subseteq S \times U \land S \neq \emptyset \Rightarrow T \subseteq U$
(g) $(S \cap T) \times (U \cap V) = (S \times U) \cap (T \times V)$
(h) $\#(S \times T) = \#S * \#T$
  where S and T are finite and "$*$" is multiplication over integers.

11. Suppose *Let* and *Num* are defined as follows:

$$Let == \{a,b,c,d,e\}$$
$$Num == \{1,2,3,4,5\}$$

(a) Give an example of each of the following:
   i. A function whose declaration is $Let \rightarrow Num$
   ii. A function whose declaration is $Let \nrightarrow Num$
   iii. A total injection from *Let* to *Num*

(b) Is it possible to give an example of a total injection from *Let* to $\{1,2,3,4\}$? If so, provide one; if not, explain why not.

12. Should the car-owner database of Section 6.6.1 be a function or relation?

13. Can a finite function have an infinite range? Briefly explain why or why not.

14. Is the union/intersection of two functions a function? If yes, argue (informally) why this is; if no, show two functions that disprove the claim.

15. Prove that relational composition of two relations $R_1 : S \leftrightarrow T$ and $R_2 : T \leftrightarrow U$ is associative.

16. Let R, S_1, S_2, and T be defined as follows.

$$\begin{aligned} R == &\{1 \mapsto (1,-1), 4 \mapsto (2,-2), 9 \mapsto (3,-3), \\ &16 \mapsto (4,-4), 25 \mapsto (5,-5), 36 \mapsto (6,-6)\} \\ S_1 == &\{1,16,32\} \\ S_2 == &\{3,4,5,7\} \\ T == &\{1 \mapsto -1, 3 \mapsto -3, 6 \mapsto -6, 9 \mapsto -9\} \end{aligned}$$

Write out in full the following sets.

(a) $S_1 \lhd R$
(b) $(S_1 \cup S_2) \lhd R$
(c) $R \rhd T$
(d) $S_2 \lhd (R \rhd T)$

17. **Domain and Range Anti-restrictions**

The *domain anti-restriction* operator, denoted $\lhd$, is defined as follows:

$$S_1 \lhd R == \{(x,y) : S \times T \mid (x,y) \in R \land x \notin S_1\}$$

Informally, the anti-restricted relation contains those elements from R whose first component does not appear in S_1.

Similarly, the *range anti-restriction* operator, denoted $\rhd$, is defined as follows:

$$R \rhd T_1 == \{(x,y) : S \times T \mid (x,y) \in R \land y \notin T_1\}$$

Informally, the anti-restricted relation contains those elements from R whose second component does not appear in T_1.

Prove the following properties of the restriction and anti-restriction operators.

(a) $S_1 \lhd R \cup S_1 \lhd R = R$
(b) $S_1 \lhd R \cap S_1 \lhd R = \emptyset$
(c) $R \rhd T_1 \cup R \rhd T_1 = R$
(d) $R \rhd T_1 \cap R \rhd T_1 = \emptyset$

18. **Function Overwriting**
The *overwriting* of a function f by another function g, denoted $f \oplus g$, allows us to create a new function defined as follows:

$$f \oplus g == ((\mathrm{dom}\, g) \lhd f) \cup g$$

The new function has the same results as f for domain elements of f not appearing in the domain of g, and the same results as g for domain elements of g.

For example, for

$$f == \{1 \mapsto red, 2 \mapsto blue, 3 \mapsto green\}$$
$$g == \{1 \mapsto pink, 4 \mapsto mauve\}$$
$$f \oplus g = \{1 \mapsto pink, 2 \mapsto blue\,, 3 \mapsto green, 4 \mapsto mauve\}$$

(a) Compute $g \oplus f$ for f and g defined as above.
(b) Compute $(f \oplus g) \oplus f$ for f and g defined as above.
(c) Show that $\mathrm{dom}(f \oplus g) = \mathrm{dom} f \cup \mathrm{dom}\, g$.
(d) Show that $f \oplus f = f$.
(e) Show that $(f \oplus g) \oplus g = f \oplus g$.
(f) Define a new operator "$\odot$" that works like "$\oplus$" but does not introduce domain elements that are not in f.

19. Using axiomatic definitions (as described in Section 6.6.5) define the following operators on sets.

(a) An operator "singletons" that takes a set and returns all its subsets with one element.
(b) An operator "infinite subsets" that takes a set and returns its infinite subsets.

20. Using axiomatic definitions (as described in Section 6.6.5) and the definition of sequences of natural numbers from Section 6.9.1 specify the following.

(a) The relation "identic" between two sequences: s and t are identic if their elements correspond.
(b) The relation "doubled" between two sequences: s and t are related if each element of t is twice as large as the corresponding element of s.
(c) The relation "mapped" between two sequences and a function f (that takes a natural number and returns a natural number): s, t, and f are related if each element of t is obtained by applying f to the corresponding element of s.
(d) The function "identity" that takes a sequence and returns an identic sequence (in the sense of 20a).
(e) The function "double" that takes a sequence and returns its double (in the sense of 20b).
(f) The function "map" that takes a sequence, and a function f and returns the mapped sequence (in the sense of 20c).

21. Sometimes it is useful to "transform" a relation into a function. For example, a relation

$$r == \{(2, red), (2, blue), (3, red)\}$$

can be turned into a function

$$f == \{(2, \{red, blue\}), (3, \{red\})\}$$

Define axiomatically the following operators.

(a) An operator "turn into function" that takes a relation like r above and turns it into a function like f.
(b) An operator "turn into relation" that takes a function like f above and turns it into a relation like r.

22. **Partial Orders**

A *partial order* over a set S is a binary relation $\leq$ that is:

- *reflexive*: $\forall a : S \bullet a \leq a$,
- *antisymmetric*: $\forall a, b : S \bullet a \leq b \land b \leq a \Rightarrow (a = b)$, and
- *transitive*: $\forall a, b, c : S \bullet a \leq b \land b \leq c \Rightarrow a \leq c$.

Prove that the subset relation $\subseteq$ over a set S is a partial order.

23. **Complete Lattice of Power Sets**

(a) A *lattice* is a *partially-ordered set* $(S, \leq)$ (a set S equipped with a partial order $\leq$) in which any two elements have a *least upper bound* (also known as a *meet*) and a *greatest lower bound* (also known as a *join*), defined as:

- $c : S$ is a least upper bound of $a, b : S$ if and only if
  i. $a \leq c \land b \leq c$, and
  ii. $\forall d : S \bullet a \leq d \land b \leq d \Rightarrow c \leq d$.
- $c : S$ is a greatest lower bound of $a, b : S$ if and only if
  i. $c \leq a \land c \leq b$, and
  ii. $\forall d : S \bullet d \leq a \land d \leq b \Rightarrow d \leq c$.

Prove that $(\mathbb{P} S, \subseteq)$ is a lattice, with the *meet* and *join* operators being $\cap$ and $\cup$, respectively.

(b) A *complete lattice* is a lattice $(S, \leq)$ with a *top* and a *bottom* element, defined as:

- $\top$ is a top element if and only if $\forall a : S \bullet a \leq \top$
- $\bot$ is a bottom element if and only if $\forall a : S \bullet \bot \leq a$

Prove that $(\mathbb{P} S, \subseteq)$ is a complete lattice. What are its top and bottom elements?

24. **Prefix closure**

Let S be a set of sequences $\text{seq}[X]$ (see Section 6.9.1). S is said to be *prefix-closed* if the following condition is satisfied

$$\forall s : S;\ s_1, s_2 : \text{seq}[X] \bullet (s = s_1 \frown s_2) \Rightarrow (s_1 \in S)$$

Determine whether the following sets are prefix closed:

(a) $\{\langle a,b\rangle, \langle\rangle, \langle a\rangle, \langle a,c,b\rangle\}$
(b) $\{\langle 6,5,4,3\rangle, \langle 3\rangle, \langle\rangle, \langle 4,3\rangle, \langle 5,4,3\rangle\}$
(c) $\{\langle Ben, Tim, Ed\rangle, \langle Tim, Ed\rangle, \langle Ed\rangle\}$

25. We defined an object's type to be the maximal set in which the object is an element. That is, any other set that the object is an element of will have fewer elements than the maximal set. Argue (informally) that this implies that the type of an object is unique — that any two maximal sets must be the same set. (Hint: assume that two different maximal sets exist and show that this leads to a contradiction.)

26. **Russell's paradox**

    The set theory described in this chapter is typed: We insisted that a set have elements of the same type. An untyped set theory does not have such restrictions. However, untyped set theories exhibit certain anomalies, one of which is known as Russell's paradox.

    Russell observed that the formalization of "*The set of sets that are not members of themselves*" in an untyped set theory leads to a contradiction. To see why, let us formalize the set as

    $$R == \{A \mid A \notin A\}$$

    Now we try to ask the question of whether R is a member of itself. If R is a member of R then by the definition of R and the axiom of comprehension $R \notin R$, contradicting our assumption. On the other hand if R is not a member of R it satisfies $R \notin R$ and by the axiom of comprehension we have $R \in R$, contradicting our assumption.

    Briefly explain why introducing typing restrictions eliminates the paradox.

Chapter 7

Reasoning Techniques

In this chapter we look at a number of specialized reasoning techniques. Each of these techniques represents a specialized style of reasoning appropriate to certain classes of reasoning problems. Although each is based fundamentally on the basic inference rules of propositional and predicate logic, by taking advantage of the particular structure of the kind of reasoning problem, we can often provide a more streamlined and understandable form of proof.

7.1 Equational Reasoning

Equational reasoning is one of the first reasoning techniques you were introduced to in your math education. Often to solve a computational problem you performed a series of simplification steps to transform the problem into a simpler and more manageable one. For example, to compute 55×9999 you could write:

$$\begin{aligned}
& 55 \times 9999 \\
= \quad & && [9999 = 10000 - 1] \\
& 55 \times (10000 - 1) \\
= \quad & && [\text{Distributivity of } \times] \\
& 55 \times 10000 - 55 \times 1 \\
= \quad & && [\cdots] \\
& \vdots
\end{aligned}$$

Each simplification step was justified by properties of numbers, or properties of $-, \times$. Such properties are called *algebraic* properties; more on this later. The

understanding was that by rationalizing individual steps you were ensuring that once a simple enough expression was obtained, that expression would be the same as the original.

This reasoning approach is formally based on a subset of predicate logic called *equational logic*. In the rest of this section we describe equational logic and the formal justification for equational proofs.

7.1.1 Equational Logic

Syntax

Sentences of an equational logic are of the form

$$e_1 = e_2$$

where expressions e_1, e_2 are quantifier-free terms of predicate logic. Recall that for $=$ between two terms to be meaningful the terms to be compared must have the same type.

Semantics

In Chapter 5 we axiomatized three important properties of equality: reflexivity, symmetry, and transitivity. We also introduced an inference rule ("eq-sub") for replacing parts of a sentence with equal parts while preserving its truth value.

Now we introduce an inference rule that enables replacing part of an expression with an equal part without changing the value of the expression. This rule is known as *substitution of equals for equals*:

$$\frac{a = b}{e[x := a] = e[x := b]} \text{ =-sub}$$

7.1.2 Equational Proofs

Substitution of equals for equals and transitivity of equality give us a method for showing that two expressions are equal. The method consists of constructing a series of transformations $e_0 = e_1$, $e_1 = e_2, \ldots, e_{n-1} = e_n$, where each individual transformation aims at replacing part of an expression with an equal expression.

We use substitution of equals for equals to rationalize individual transformations and write:

$$e_i[x := a]$$

$$
\begin{array}{ll}
= & \text{[explanation of why } a = b\text{]} \\
e_{i+1}[x := b]
\end{array}
$$

for a single transformation step.

More generally we can use algebraic properties (such as commutativity of addition, associativity of multiplication, distributivity of multiplication over addition, etc.) to transform the entire expression under equality, writing:

$$
\begin{array}{ll}
e_k & \\
= & \text{[explanation of why } e_k = e_{k+1}\text{]} \\
e_{k+1} &
\end{array}
$$

Transitivity of $=$ allows us to chain individual transformations together and get a derivation of the form:

$$
\begin{array}{ll}
e_0 & \\
= & \text{[explanation of why } e_0 = e_1\text{]} \\
e_1 & \\
= & \text{[explanation of why } e_1 = e_2\text{]} \\
e_2 & \\
\vdots & \\
= & \text{[explanation of why } e_{n-2} = e_{n-1}\text{]} \\
e_{n-1} & \\
= & \text{[explanation of why } e_{n-1} = e_n\text{]} \\
e_n &
\end{array}
$$

Example 7.1. Let us simplify $(a+b)^2$ into $a^2 + 2 \times a \times b + b^2$ using equational reasoning.

$$
\begin{array}{ll}
(a+b)^2 & \\
= & \text{[definition of exponentiation } m^2 = m \times m\text{]} \\
(a+b) \times (a+b) & \\
= & \text{[right distributivity of } \times \text{ over } +\text{]} \\
a \times (a+b) + b \times (a+b) & \\
= & \text{[left distributivity of } \times \text{ over } +\text{]}
\end{array}
$$

$$
\begin{aligned}
&(a \times a + a \times b) + (b \times a + b \times b) \\
&= && \text{[definition of exponentiation } m \times m = m^2\text{]} \\
&(a^2 + a \times b) + (b \times a + b^2) \\
&= && \text{[associativity of } +\text{]} \\
&a^2 + (a \times b + b \times a) + b^2 \\
&= && \text{[commutativity of } \times\text{]} \\
&a^2 + (a \times b + a \times b) + b^2 \\
&= && [m + m = 2 \times m] \\
&a^2 + 2 \times a \times b + b^2
\end{aligned}
$$

We intentionally detailed the derivation steps in this example. In actual proofs we would instead appeal to "arithmetic" to directly write $(a+b)^2 = a^2 + 2 \times a \times b + b^2$. □

7.2 Generalized Equational Reasoning

We now show how the ideas behind equational reasoning can be applied in the context of reasoning about logical statements.

First, any transitive operator, not just $=$, can motivate a chain-like style of reasoning. Transitivity of $\Leftrightarrow$, for example, can be used to chain a number of transformations $P_0 \Leftrightarrow P_1, P_1 \Leftrightarrow P_2, \cdots, P_{n-1} \Leftrightarrow P_n$ into a single derivation of the form:

$$
\begin{aligned}
&P_0 \\
&\Leftrightarrow && \text{[explanation of why } P_0 \Leftrightarrow P_1\text{]} \\
&P_1 \\
&\vdots \\
&\Leftrightarrow && [\cdots] \\
&P_{n-1} \\
&\Leftrightarrow && \text{[explanation of why } P_{n-1} \Leftrightarrow P_n\text{]} \\
&P_n
\end{aligned}
$$

Similarly, a proof for $Q_0 \Rightarrow Q_k$ can be constructed as a series of transforma-

tions $Q_0 \Rightarrow Q_1, Q_1 \Rightarrow Q_2, \ldots, Q_{k-1} \Rightarrow Q_k$.[1] In fact some (or all) steps of the transformation can be of the form $Q_i \Leftrightarrow Q_{i+1}$ since that trivially implies $Q_i \Rightarrow Q_{i+1}$.

Example 7.2. We prove $\vdash P \Rightarrow (Q \Rightarrow P)$ in an equational style:

$$
\begin{array}{ll}
P & \\
\Rightarrow & [\vee\text{-intro}] \\
\neg Q \vee P & \\
\Rightarrow & [\Rightarrow\text{-Alternative}] \\
Q \Rightarrow P &
\end{array}
$$

Notice the use of $\vee$-intro to derive $P \Rightarrow (\neg Q \vee P)$. Recall that a rule of the form $R \vdash S$ can be used as theorem $\vdash R \Rightarrow S$ thanks to the Deduction Theorem for predicate logic. □

It is sometimes convenient to write $\Rightarrow$-preserving derivations "backwards," writing $P \Leftarrow Q$ instead of $Q \Rightarrow P$.

Example 7.3. The previous example can be written using $\Leftarrow$ as follows.

$$
\begin{array}{ll}
Q \Rightarrow P & \\
\Leftarrow & [\Rightarrow\text{-Alternative}] \\
\neg Q \vee P & \\
\Leftarrow & [\vee\text{-intro}] \\
P &
\end{array}
$$

□

When doing equational-style proofs that involve transforming a sentence under $\Leftrightarrow$ and $\Rightarrow$ we can use several rules that resemble substitution of equals for equals. We discuss such rules next.

7.2.1 $\Leftrightarrow$ Substitution

We give two rules that directly support equational-style reasoning about $\Leftrightarrow$. The first is an alternative formulation of rule "eq-sub" from Chapter 5, and the second allows replacing sub-sentences with equivalent sentences.

$$\frac{m = n}{S[x := m] \Leftrightarrow S[x := n]}\ \text{eq-sub}$$

[1] Compare the use of $\Rightarrow$ to the use of $\leq$ when concluding $a_0 \leq a_n$ from $a_0 \leq a_1 \leq a_2 \leq \ldots \leq a_{n-1} \leq a_n$.

$$\frac{P \Leftrightarrow Q}{S[x := P] \Leftrightarrow T[x := Q]} \Leftrightarrow\text{-sub}$$

It is obvious that the above rules can also be used in derivations involving $\Rightarrow$. For example, we could write:

$$\frac{P \Leftrightarrow Q}{S[x := P] \Rightarrow T[x := Q]} \Rightarrow\text{-sub}$$

However, we *cannot* generally use $P \Rightarrow Q$ to justify $S[x := P] \Rightarrow T[x := Q]$. We discuss what we can do in situations where S has a certain structure next.

7.2.2 Monotonicity

Transformations that involve $\Rightarrow$ sometimes require application of so-called *monotonicity rules*. Figure 7.1 lists some useful monotonicity rules.

Monotonicity rules allow restricting the argument of why certain sentences are related by logical implication to an argument involving parts of such sentences. For example, $\vee$-Mono is read as: to show a proof for $P \vee R \Rightarrow Q \vee R$ it is sufficient to show a proof for $P \Rightarrow Q$.[2] This is incredibly useful since a proof for $P \Rightarrow Q$ will more often than not be much simpler than a proof for $P \vee R \Rightarrow Q \vee R$.

Example 7.4. A direct application of $\vee$-Monotonicity is that it is sufficient to prove $P \wedge Q \Rightarrow P$ in order to prove $(P \wedge Q) \vee R \Rightarrow P \vee R$. $\square$

Rules such as $\forall$-Body Mono, which involve quantifiers, can be used to simplify sentences by moving quantifiers outwards, and reducing the number of quantifiers. By applying $\forall$-Body Mono, instead of having to show a proof involving two $\forall$ quantifiers we are allowed to produce an argument involving only one.

7.3 Proof by Reduction to Truth

An interesting application of the $\Leftrightarrow$- and $\Rightarrow$-preservation strategy is to prove a sentence P by showing that it is a consequence of the logical constant *true*, or more generally, a logical consequence of a tautology. In the rest of this section we

[2] Compare for example with the arithmetic rule that allows reducing the proof for $a \times c \leq b \times c$ to a proof for $a \leq b$, for $a, b, c \in \mathbb{N}$. Why is $a \leq b$ a sufficient, but not a necessary condition? Is $P \Rightarrow Q$ a necessary condition for $P \vee R \Rightarrow Q \vee R$?

$\vdash (P \Rightarrow Q) \;\Rightarrow\; (P \vee R \Rightarrow Q \vee R)$	$\vee$-Mono
$\vdash (P \Rightarrow Q) \;\Rightarrow\; (P \wedge R \Rightarrow Q \wedge R)$	$\wedge$-Mono
$\vdash (P \Rightarrow Q) \;\Rightarrow\; ((R \Rightarrow P) \Rightarrow (R \Rightarrow Q))$	Conseq. Mono
$\vdash (P \Rightarrow Q) \;\Rightarrow\; (\neg Q \Rightarrow \neg P)$	$\neg$-Antimono
$\vdash (P \Rightarrow Q) \;\Rightarrow\; ((Q \Rightarrow R) \Rightarrow (P \Rightarrow R)$	Ante. Antimono
$\vdash (P \Rightarrow P') \wedge (Q \Rightarrow Q') \;\Rightarrow\; (P \wedge Q \Rightarrow P' \wedge Q')$	
$\vdash (\forall x : T \bullet P(x) \Rightarrow Q(x)) \Rightarrow ((\forall x : T \bullet P(x)) \Rightarrow (\forall x : T \bullet Q(x)))$	$\forall$-Body Mono
$\vdash (\exists x : T \bullet P(x) \Rightarrow Q(x)) \Rightarrow ((\exists x \bullet P(x)) \Rightarrow (\exists x \bullet Q(x)))$	$\exists$-Body Mono

Figure 7.1: Monotonicity and Antimonotonicity

first formalize the logical constants *true* and *false*, and then show how they can be used in equational-style derivations.

We axiomatize *true* as a tautology, and *false* as a contradiction:

Truth:	$\vdash \mathit{true}$
Falsity:	$\vdash \mathit{false} \Leftrightarrow \neg \mathit{true}$

Given this axiomatization we can prove that *true* and *false* satisfy some interesting properties, which we list in Figure 7.2.

$\vdash \mathit{true} \wedge P \Leftrightarrow P$	$\vdash \mathit{false} \wedge P \Leftrightarrow \mathit{false}$
$\vdash \mathit{true} \vee P \Leftrightarrow \mathit{true}$	$\vdash \mathit{false} \vee P \Leftrightarrow P$
$\vdash (P \Rightarrow \mathit{true}) \Leftrightarrow \mathit{true}$	$\vdash (\mathit{false} \Rightarrow P) \Leftrightarrow \mathit{true}$
$\vdash (P \Rightarrow P) \Leftrightarrow \mathit{true}$	$\vdash (P \Rightarrow \mathit{false}) \Leftrightarrow \neg P$
$\vdash (\mathit{true} \Rightarrow P) \Leftrightarrow P$	$\vdash (\neg P \Rightarrow \mathit{false}) \Leftrightarrow P$

Figure 7.2: Truth and Falsity

In particular, the property $(\mathit{true} \Rightarrow P) \Leftrightarrow P$ means that if we have a proof for $\mathit{true} \Rightarrow P$ we have a proof for P; similarly if P is a tautology then we automatically have a proof for $\mathit{true} \Rightarrow P$. This property gives us a method for deriving a property Q in equational style. To derive a property Q we could construct an equational style proof for $\mathit{true} \Rightarrow Q$; a proof for $\mathit{true} \Leftrightarrow Q$ obviously works as well.

Example 7.5. We show $\vdash (P \wedge Q) \vee R \Rightarrow P \vee R$:

$$
\begin{array}{ll}
(P \wedge Q) \vee R \Rightarrow P \vee R & \\
\Leftarrow & [\vee\text{-Mono}]
\end{array}
$$

$P \wedge Q \Rightarrow P$
$\Leftarrow$ [$P \Leftrightarrow P \wedge true$ Figure 7.2]
$P \wedge Q \Rightarrow P \wedge true$
$\Leftarrow$ [$\wedge$-Mono]
$Q \Rightarrow true$
$\Leftarrow$ [$(P \Rightarrow true) \Leftrightarrow true$, Figure 7.2]
$true$

The last three steps could be replaced by an appeal to $\wedge$-elimination. □

7.4 Other Proof Techniques

In this section we show some other proof techniques inspired by how proofs are done in standard mathematics.

7.4.1 Assuming the Antecedent

It is common in mathematics to prove an implication $P \Rightarrow Q$ by assuming the antecedent P and proving the consequent Q. By "assuming the antecedent" we momentarily think of it as an axiom and thus equivalent to *true*. Another way to look at it is that P becomes a premise for deriving Q. We can generalize the strategy as follows: to prove $P_1 \wedge P_2 \wedge \ldots \wedge P_n \Rightarrow Q$ we can assume $P_1, P_2, \ldots, P_n$ and prove Q.

Example 7.6. Let us for example show a proof for $P \wedge Q \Rightarrow P \Leftrightarrow Q$.

Assume P, Q
Show $P \Leftrightarrow Q$
P
$\Leftrightarrow$ [Assumption P]
$true$
$\Leftrightarrow$ [Assumption Q]
Q

□

The formal justification of this proof technique relies on the Deduction Theorem for predicate logic (see Chapter 5), which says that if we have $P_1, P_2, \cdots, P_n \vdash Q$ then we also have $\vdash P_1 \wedge P_2 \wedge \cdots \wedge P_n \Rightarrow Q$.

7.4.2 Proof by Mutual Implication

A direct application of bi-implication introduction can be seen in the proof strategy known as "proof by mutual implication." It simply consists of proving a bi-implication $P \Leftrightarrow Q$ by showing that both directions $P \Rightarrow Q$ and $P \Leftarrow Q$ hold.

7.4.3 Proof by Case Analysis

To illustrate the proof-by-cases technique consider how we would prove that $-x \times y = x \times -y$, where x, y are integers. A typical proof consists of showing that regardless of whether x, y are positive or not the equality holds. We then argue for each combination of x and y:

Case $0 < x$ and $0 < y$	
$-x \times y$	
$=$	[property of absolute value, $-x \leq 0$ and $0 < y$]
$-\mid x \times y \mid$	
$=$	[property of absolute value, $0 < x$ and $-y \leq 0$]
$x \times -y$	
Case $x \leq 0$ and $0 < y$	
$-x \times y$	
$\vdots$	
$=$	$[\cdots]$
$x \times -y$	
Case $0 < x$ and $y \leq 0$	
$-x \times y$	
$\vdots$	
$=$	$[\cdots]$
$x \times -y$	

Case $x \leq 0$ and $y \leq 0$

$-x \times y$

$\vdots$

$=$ $[\cdots]$

$x \times -y$

Having exhausted all possibilities we could conclude the proof.

A proof by case analysis is a manifestation of the following property of implication:

$$\vdash ((P \vee Q) \wedge (P \Rightarrow R) \wedge (Q \Rightarrow R)) \Rightarrow R \tag{7.1}$$

To prove R we find the cases P and Q such that $P \vee Q$ holds. We then show that in each case R follows, that is, $P \Rightarrow R$, and $Q \Rightarrow R$.

Often it will be obvious that $P \vee Q$ holds, and in those cases we omit the formal proof for $P \vee Q$. One such example is when Q is instantiated with $\neg P$. This gives rise to the strategy known as "simple case analysis," and which can formally be expressed as:

$$\vdash (P \Rightarrow R) \wedge (\neg P \Rightarrow R) \Leftrightarrow R \tag{7.2}$$

The rule generalizes trivially to more than two cases. For example, for three cases the justification would be:

$$\vdash ((P \vee Q \vee R) \wedge (P \Rightarrow S) \wedge (Q \Rightarrow S) \wedge (R \Rightarrow S)) \Rightarrow S \tag{7.3}$$

We have encountered proof by cases in the form of disjunction elimination, which could be rephrased as:[3]

$$\vdash ((P \Rightarrow R) \wedge (Q \Rightarrow R)) \Leftrightarrow (P \vee Q \Rightarrow R) \tag{7.4}$$

7.4.4 Proof by Contradiction

[TBD]

[3]Exercise: Prove $(((P \vee Q) \wedge (P \Rightarrow R) \wedge (Q \Rightarrow R)) \Rightarrow R) \;\Leftrightarrow\; (((P \Rightarrow R) \wedge (Q \Rightarrow R)) \Leftrightarrow (P \vee Q \Rightarrow R))$.

7.4.5 Universal Introduction

Another common technique in mathematics is to prove a statement of the form $\forall x : T \bullet P(x)$ by letting x stand for an "arbitrary object" from T and showing that $P(x)$ holds. An x is considered arbitrary if it has not been seen in the proof so far. If, for example, the statement is being proved under premises then x should not appear free in those premises. The justification for this technique is $\forall$-introduction.

Example 7.7. We prove $\forall x : \mathbb{N} \bullet 0 < (x+1)^2$.

Let x be an arbitrary natural number
Show $0 < (x+1)^2$
0
$<$ [Arithmetic]
1
$\leq$ [Arithmetic]
$x^2 + 2x + 1$
$\leq$ [Arithmetic]
$(x+1)^2$

We used an equational style in the proof making use of a transitivity-like property of $<$ and $\leq$, namely that if $a < b$ and $b \leq c$ then $a < c$. □

7.4.6 Existential Introduction and Elimination

[TBD]

7.5 Induction

We often find ourselves in the situation of wanting to prove something of the form:

$$\forall x : S \bullet P(x)$$

That is, we want to prove that all elements of some set S have the property P. We could try using universal introduction, or do a proof by contradiction. Unfortunately, these techniques are sometimes insufficient. For example, trying to prove $2^m \times 2^n = 2^{m+n}$ (where m, n are natural numbers) for arbitrary m, n does not get

us anywhere; neither does trying to derive a contradiction from its negation. We need to somehow consider *all* the possibilities for m and n. Clearly, enumerating all the cases is impossible. Fortunately, when dealing with natural numbers and other sets whose elements are defined by some regular structural rules, we can use the technique of *induction* to prove the desired properties.

Examples of sets that can be defined by structural rules include:

- **natural numbers**: starting with the element 0, each element is the successor of some other element in the set.
- **sequences**: starting with the empty sequence, $\langle\rangle$, each element is built up by appending an element to an existing list.
- **parse trees**: starting with the "terminal" nodes of a grammar, each parse tree is a structure defined by one of the "non-terminal" productions of the grammar.

In the first of these examples, the proof technique is typically referred to as "natural induction." In the others, it is typically referred to as "structural induction."

The technique of proof by induction resembles a proof by case analysis in the following sense. Each structural rule used to define a set S describes a (proper) subset of S, and the union of the resulting subsets corresponds with S. Therefore, it is sufficient to argue that each of the constituent subsets of S have a property P in order to prove that S itself has that property. For example, for natural numbers these subsets would be the set with sole element 0 and the set of elements that can be expressed as the successor of some natural number. Therefore, it is sufficient to derive a proof for $P(0)$, and a proof for $P(m+1)$ where m is an arbitrary natural number (and $m+1$ is therefore described by the rule "is a successor of some element of $\mathbb{N}$").

Induction is much more powerful than case analysis however. We need to show $P(0)$. And we only need to provide a derivation for $P(m+1)$ *under the assumption* that we have a derivation for $P(m)$.[4] That is, it suffices to show that a proof for $P(m+1)$ can be constructed from a proof for $P(m)$. Informally, the reason this works is that if $m = 0$ then we have a proof for $P(m)$ since we have a proof for $P(0)$. Since we can construct a proof for $P(m+1)$ from a proof for $P(m)$ that in turn means that we have a proof for $P(1)$. The argument continues that having a proof for $P(1)$ and a way of constructing a proof for $P(2)$ from that of $P(1)$ gives us a proof for $P(2)$ and so on.

[4] Variations allow for assumptions that P is true for all numbers up to m; see Exercise 4.

7.5.1 Natural Induction

Formally, the inference rule for natural induction can be expressed as follows:

$$\frac{P(0) \qquad \forall k : \mathbb{N} \bullet P(k) \Rightarrow P(k+1)}{\forall m : \mathbb{N} \bullet P(m)}\ \mathbb{N}\text{-ind}$$

We will refer to a derivation for $P(0)$ as the *base case*, and a derivation for $\forall k : \mathbb{N} \bullet P(k) \Rightarrow P(k+1)$ as the *inductive case*. The inductive case will be proved by an implicit universal quantifier introduction: we assume that k is an arbitrary natural number, and try to derive $P(k) \Rightarrow P(k+1)$. We will then derive $P(k+1)$ under the assumption that $P(k)$ holds – this assumption will be known as the *induction hypothesis*. This step of the proof will often require transforming the predicate $P(k+1)$ into a predicate containing $P(k)$.[5]

Example 7.8. As an example we prove that $2^0 + 2^1 + 2^2 + ... + 2^n = 2^{n+1} - 1$; we have therefore let $P(n)$ be the property $2^0 + 2^1 + 2^2 + ... + 2^n = 2^{n+1} - 1$.
Base case: We show $P(0)$, that is $2^0 = 2^{0+1} - 1$; this follows trivially from arithmetic facts.
Inductive step: We assume $k \in \mathbb{N}$ and $P(k)$ holds. That is, our induction hypothesis is $2^0 + 2^1 + 2^2 + ... + 2^k = 2^{k+1} - 1$. We then show $P(k+1)$ holds, that is $2^0 + 2^1 + 2^2 + ... + 2^k + 2^{k+1} = 2^{(k+1)+1} - 1$.

$$\begin{array}{ll}
2^0 + 2^1 + 2^2 + ... + 2^k + 2^{k+1} & \\
= & \text{[substitution, Induction Hypothesis]} \\
(2^{k+1} - 1) + 2^{k+1} & \\
= & \text{[arithmetic]} \\
2 \times 2^{k+1} - 1 & \\
= & \text{[arithmetic]} \\
2^{k+2} - 1 & \\
= & \text{[arithmetic]} \\
2^{(k+1)+1} - 1 &
\end{array}$$

□

[5] In fact, if such a reduction is not necessary for the proof then the proof can be carried out with non-inductive strategies.

As with most other proof techniques some derivations will require several applications of the induction technique. For example, proving $2^m \times 2^n = 2^{m+n}$ requires induction on both m and n.

7.5.2 Structural Induction over Binary Trees

In the rest of this section we illustrate the use of structural induction over binary trees. Other forms of structural induction work similarly. Here is how we will proceed. First, we provide a "structural" definition of trees: this determines the rules that allow us to define all trees of interest. Next, we define what we mean by "size" of a tree. Then we will propose a small theorem about calculating the size; this is what we will then prove.

Definition of Binary Trees

Consider the recursive definition of trees from Section 6.8.

$$TREE ::= leaf \mid node\langle\langle TREE \times TREE \rangle\rangle$$

It says that a tree is either a *leaf* or it is made up of two subtrees glued together with a *node*. Some examples of the kind of structures that we can build up using this definition:

$$leaf$$
$$node(leaf, leaf)$$
$$node(node(leaf, leaf), leaf)$$
$$node(leaf, node(node(leaf, leaf), leaf))$$

An Induction Rule for Binary Trees

Our definition of the binary trees gives rise to the following inference rule:

$$\frac{P(leaf) \qquad \forall t_1, t_2 : TREE \bullet P(t_1) \land P(t_2) \Rightarrow P(node(t_1, t_2))}{\forall t : TREE \bullet P(t)} \text{ TREE-ind}$$

As with natural induction there are two cases: a base case for leaves, and an inductive case for composite trees. Since a composite tree includes two subtrees, the induction hypothesis will include two assumptions, one for each of the subtrees.

Definition of Size

There are many ways that we might define the size of a tree. Some definitions would count just the leaves, others just the nodes. Here we will count both.

Informally, we could say that the size of a single leaf is 1, while the size of a tree built out of two subtrees, say t_1 and t_2, is the sum of the sizes of the two subtrees plus 1 (for the joining node).

The basic idea of this definition is that we define the size of a tree inductively over the structure, saying how the size of a given tree is calculated from sizes its parts. We define the function axiomatically, by first declaring its type (in this case $size : TREE \rightarrow \mathbb{N}$), and then by saying how it is defined in each of the two cases.

$$
\begin{array}{|l}
size : TREE \rightarrow \mathbb{N} \\
\hline
\forall t_1, t_2 : TREE \bullet \\
\quad size(leaf) = 1 \land \\
\quad size(node(t_1, t_2)) = 1 + size(t_1) + size(t_2)
\end{array}
$$

In a similar way, we might make other definitions about trees. Here are two useful ones:

$$
\begin{array}{|l}
leaves : TREE \rightarrow \mathbb{N} \\
nodes : TREE \rightarrow \mathbb{N} \\
\hline
\forall t_1, t_2 : TREE \bullet \\
\quad leaves(leaf) = 1 \land \\
\quad leaves(node(t_1, t_2)) = leaves(t_1) + leaves(t_2) \land \\
\quad nodes(leaf) = 0 \land \\
\quad nodes(node(t_1, t_2)) = 1 + nodes(t_1) + nodes(t_2)
\end{array}
$$

Proving a Theorem by Structural Induction

Based on what we've said about our definitions, there is a pretty obvious connection between the three definitions given above. That is, we would expect the size of a tree to be the sum of the leaves and the nodes. We can make this precise as follows:

Theorem 1. $\forall t : TREE \bullet size(t) = leaves(t) + nodes(t)$.

We will prove this by structural induction.

Proof:
Base Case: Show the property holds for *leaf*, that is, $size(leaf) = leaves(leaf) + nodes(leaf)$.

$$
\begin{aligned}
& size(leaf) \\
= \quad & \text{[definition of } size\text{]} \\
& 1 \\
= \quad & \text{[arithmetic]} \\
& 1+0 \\
= \quad & \text{[definition } leaves\text{]} \\
& leaves(leaf)+0 \\
= \quad & \text{[definition } nodes\text{]} \\
& leaves(leaf)+nodes(leaf)
\end{aligned}
$$

Induction Case: Assume that the property holds for trees t_1 and t_2, that is, $size(t_1) = leaves(t_1)+nodes(t_1)$, and $size(t_2) = leaves(t_2)+nodes(t_2)$. Show that it holds for $node(t_1,t_2)$.

$$
\begin{aligned}
& size(node(t_1,t_2)) \\
= \quad & \text{[definition of } size\text{]} \\
& 1+size(t_1)+size(t_2) \\
= \quad & \text{[induction hypothesis]} \\
& 1+(leaves(t_1)+nodes(t_1))+(leaves(t_2)+nodes(t_2)) \\
= \quad & \text{[commutative and associative properties of } +\text{]} \\
& (leaves(t_1)+leaves(t_2))+(1+nodes(t_1)+nodes(t_2)) \\
= \quad & \text{[definition of } leaves\text{]} \\
& leaves(node(t_1,t_2))+(1+nodes(t_1)+nodes(t_2)) \\
= \quad & \text{[definition of } nodes\text{]} \\
& leaves(node(t_1,t_2))+nodes(node(t_1,t_2))
\end{aligned}
$$

7.6 Proof Strategies

[TBD]

Chapter Notes

[TBD]

Further Reading

[TBD]

Exercises

1. Prove in equational style the following laws for set union:

   (a) $S \cup T = T \cup S$

   (b) $S \cup \emptyset = S$

   (Hint: To prove $S = T$ show $x \in S \Leftrightarrow x \in T$. This can often be done in an equational style.)

2. Prove the following equivalences in equational style:

   (a) $\neg(p \wedge (q \vee r)) \;\Leftrightarrow\; (\neg p \vee \neg q) \wedge (\neg p \vee \neg r)$
   (b) $(p \vee \neg r) \wedge (r \vee \neg p) \;\Leftrightarrow\; (p \Leftrightarrow q) \wedge (q \Leftrightarrow r)$

   (Hint: Use properties from Figure 4.2 in Chapter 4.)

3. **Natural Induction**

   Prove the following claims by induction over the natural numbers:

   (a) The sum of the first n odd natural numbers is n^2.
   (Hint: the n^{th} odd integer $= 2n-1$.)
   (b) $2^m \times 2^n = 2^{m+n}$
   (Hint: the proof will require two applications of induction, one for m and one for n.)

4. **Natural Induction: Alternative Rule**
   The following is an alternative inference rule for natural induction:

$$\frac{P(0) \qquad \forall k : \mathbb{N} \bullet (\forall i : \mathbb{N} \bullet i \leq k \wedge P(k)) \;\Rightarrow\; P(k+1)}{\forall m : \mathbb{N} \bullet P(m)}\ \mathbb{N}\text{-ind-alt}$$

Consider the definitions below:

$$
\begin{array}{|l}
T : \mathbb{N} \to \mathbb{N} \\
Fib : \mathbb{N} \to \mathbb{N} \\
\hline
\forall n : \mathbb{N} \bullet \\
\quad T(0) = 1 \land \\
\quad T(1) = 1 \land \\
\quad 2 \leq n \Rightarrow T(n) = T(n-1) + T(n-2) + 1 \land \\
\\
\quad Fib(0) = 0 \land \\
\quad Fib(1) = 1 \land \\
\quad 2 \leq n \Rightarrow Fib(n) = Fib(n-1) + Fib(n-2)
\end{array}
$$

(a) Using $\mathbb{N}$-ind-alt prove that $\forall n : \mathbb{N} \bullet Fib(n+2) \leq T(n+1)$.
(b) Would the proof for $\forall n : \mathbb{N} \bullet Fib(n+1) \leq T(n)$ be different from that of (4a)? Why or why not?

5. **Structural Induction over Binary Trees**

For this exercise use the definitions for binary trees in Section 7.5.2.

(a) Show that $\forall t : TREE \bullet leaves(t) = nodes(t) + 1$.
(b) Define a *mirror* function that recursively swaps the branches of a tree.
(c) Show that $\forall t : TREE \bullet size(mirror(t)) = size(t)$.
(d) Show that $\forall t : TREE \bullet mirror(mirror(t)) = t$.

6. **Induction over Sequences**

Consider sequences over natural numbers defined as:

$$SEQ ::= \langle\rangle \mid cons\langle\langle \mathbb{N} \times SEQ \rangle\rangle$$

That is, a sequence is either empty or is formed by adding a natural number to the front of another sequence. To avoid clutter we will pretty-print *cons* as an infix operator "::". For example, $cons(n, s)$ will be written $(n :: s)$ for a number n and sequence s.

(a) List five legal sequences of different lengths.
(b) Formalize the structural induction rule for sequences.

(c) Define a function *rev* that reverses a sequence; for example, applied to sequence $(5 :: (3 :: (2 :: \langle\rangle)))$ the function will return $(2 :: (3 :: (5 :: \langle\rangle)))$.
(d) Prove that *rev* is *idempotent*, that is, $rev(rev(s)) = s$.
(e) Define an infix function $\frown$ that concatenates two sequences; for example, for two sequences $s = (4 :: (2 :: \langle\rangle))$ and $t = (6 :: (3 :: (5 :: \langle\rangle)))$, $s \frown t$ will return sequence $(6 :: (3 :: (5 :: (4 :: (2 :: \langle\rangle)))))$.
(f) Prove that $\langle\rangle$ is *unit* of $\frown$, that is, $\langle\rangle \frown s = s$ and $s \frown \langle\rangle = s$.
(g) Prove that $\frown$ is *associative*, that is, $(s \frown t) \frown r = s \frown (t \frown r)$.
(h) Prove that $n :: s = (n :: \langle\rangle) \frown s$.
(i) Prove that $rev(s \frown t) = rev(t) \frown rev(s)$.
(j) Prove that $rev(s \frown rev(t)) = t \frown rev(s)$.

Part II

State Machines

In Part II we will be studying general concepts associated with state machines. We introduce some basic ideas for defining state machines and their behaviors (Chapters 8 and 9), for reasoning about state machines (Chapter 10), and for relating two state machines to each other (Chapters 11 to **??**).

In Part II we will emphasize concepts, not notation. We will use standard mathematical notation (as we have seen in Part I) for defining terms. We will also introduce a little notation to make state machine descriptions easier to read. Later in Part III we will be covering a few *specific* notations that are useful for describing *specific* classes of state machines. For example, Z is useful for describing sequential state machines; CSP, concurrent ones.

At some times in Part II, we will be excessively pedantic and precise. At others we will be intentionally sloppy because the details are either unimportant or tedious. Of course the hard part is knowing when we may be sloppy and when we must be precise. Acquiring this sensibility takes time and practice.

There is no one accepted model of state machines that is used as a standard for all of computer science or software engineering. Rather, each domain, discipline, area, etc., defines its own variation that is appropriate for the class of problems at hand. Thus, we are making a noble attempt in the first chapter of Part II to present a single model that is simple and abstract enough that will then allow us to present many of the common variations in subsequent chapters.

Chapter 8

State Machines: Basics

In this chapter we cover some basic concepts for defining simple state machines. These basic concepts underlie many of the different kinds of state machine models used in computer science and software engineering. In the next chapter we look at some of those variations. In this chapter we will also see our use of concepts from Part I: in Section 8.5 we see how to use set notation and predicate logic to describe infinite state machines in a succinct and precise manner.

8.1 Why State Machines?

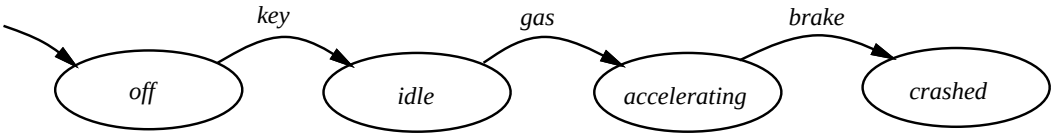
A state machine is a simple mathematical model. It is a fundamental and ubiquitous model in computer science. A computer is nothing but a state machine. It has registers and memory (state) which contain values that change over time as its operations are executed (state transitions). A programming language is a way to describe a class of state machines. A program written in that programming language is a description of a state machine. The execution of that program corresponds to an execution of the state machine it describes.

A software system is a very, very complicated state machine. One thing that makes it complicated is its size: there is usually an infinite number of states, an infinite number of state transitions, an infinite number of executions, and each execution may possibly be infinite (not terminate). If everything were finite and not very large, we could probably reason about the behavior of the software system entirely in our heads. But since the real world is not so small and manageable, we need to find ways to model, describe, and reason about these large things in terms of a finite and small(er) number of small things.

One purpose of this book is to describe some of these ways of managing complexity. There are three themes we will visit and revisit: *notation*, *abstraction*, and *modularization*. With the appropriate notation, using abstraction and modularization (composition and decomposition) techniques, we model and reason about complex software systems. But remember, as with any mathematical model, we discuss only those things that that model models. If a state machine does not let us model the cost of the system, then we cannot reason about how expensive or cheap it will be to build.

8.2 A Simple Example

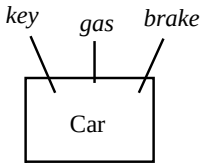
Let's start with a simple example of a car and model it as a state machine:



Car's State Transition Diagram

This Car has a very short lifetime. It starts out in the initial state where it is *off*. When we perform the action of turning the *key* to start the car, it moves into the *idle* state. After we apply some *gas*, it moves into the *accelerating* state. If we apply the *brake*, it dies, ending up in the *crashed* state.

We can think of the Car as a black box whose *interface* to the outside world is a set of observable states (ovals above) and a set of actions (arrows above). Sometimes we focus on just the actions and thus depict the Car's interface as follows:



Car's Interface

Imagine stuffing the Car's state transition diagram inside the box. In Section 8.6.1 we will discuss interfaces more.

Suppose in this Car example, we turn the key and we are unlucky: the car will not start. To model the possibility that the car goes from the *off* state to more than one possible next state (*off* and *idle*), we need to model *nondeterministic* behavior.

Nondeterminism comes up naturally in the real world when we cannot predict what the next state of a machine will be given some event or action. Perhaps it is due to an internal choice made by the machine. For example, choosing an element from a set is a nondeterministic action; we get some element of the set back, but we do not know which one.

8.3 State Machines: Definitions of Basic Concepts

Definition 1. *A* **state machine** M *is a quadruple,* (S, I, A, δ)*, where*

- S *is a finite or infinite set of* **states***,*
- $I \subseteq S$ *is a finite set of* **initial states***,*
- A *is a finite set of* **actions***, and*
- $\delta \subseteq S \times A \times S$ *is a* **state transition relation**.

If S is finite, then M is a *finite state machine*.

I is sometimes defined so that it can be an infinite subset of S (when of course S is infinite). A is sometimes called the *alphabet* of M. Elements in A are sometimes called *events* or *operations*. In other models A may be infinite. Sometimes δ is defined to be a function, $S \times A \pfun S$, rather than a relation.

However, by our having δ be a relation, we can more easily model nondeterminism. Recall that it is equivalent to viewing the type of δ as $S \times A \rel S$; thus, given a state and an action, there are more than one next states to which we can move.

(Aside: the action component, a, of a triple, (s, a, s'), in δ, is sometimes just viewed as the *label* for the state transition from s to s'. Thus, sometimes these kinds of state machines are called *labeled state transition systems*.)

Applying the above definition of a state machine, we have for the Car:

$$
\begin{array}{l}
\mathrm{Car} == (\\
\{off, idle, accelerating, crashed\}, \\
\{off\}, \\
\{key, gas, brake\}, \\
\{(off, key, idle), (idle, gas, accelerating), (accelerating, brake, crashed)\} \\
).
\end{array}
$$

8.3.1 Concepts

Let M be a state machine (S, I, A, δ).

Definition 2. *Each triple, (s, a, s'), in δ of M is a* **step** *of M.*

Definition 3. *An* **execution fragment** *is a finite or infinite sequence $\langle s_0, a_1, s_1, a_2, s_3, \ldots \rangle$ of alternating states and actions such that for all i (s_i, a_{i+1}, s_{i+1}) is a step of M.*

Definition 4. *An* **execution** *is an execution fragment starting with an initial state of M (and ending in a state if finite).*

Definition 5. *A state is* **reachable** *if it is a last state of a finite execution.*

There are two reasonable ways to define what the *behavior* of a state machine is. One way ("event-based" or "action-based") says what is observable are a machine's actions; the other ("state-based") says what is observable are a machine's states. Which way we might prefer is philosophical. Here are two alternative definitions of what a trace is:

Definition 6. *(Event-based) A* **trace** *is the sequence of actions of an execution.*

Definition 7. *(State-based) A* **trace** *is the sequence of states of an execution or is the sequence, $\langle s_i \rangle$, for each $s_i \in I$.*

Finally, we define what the behavior of a machine is.

Definition 8. *The* **behavior** *of a machine M (Beh(M)) is the set of all traces of M.*

Behaviors are *prefix-closed*, which means, for a given behavior, B: (1) The empty trace is in $Beh(M)$; and (2) if a trace is in B then any prefix of that trace is in B.

In other work on state machine models, behaviors do not have or are not assumed to have the prefix-closure property.

8.3.2 Revisiting the Car

Consider the Car machine.

- $(idle, gas, accelerating)$ is a step of Car; $(idle, brake, crashed)$ is not.
- $\langle idle, gas, accelerating, brake, crashed \rangle$ is an execution fragment of Car.
- $\langle off, key, idle \rangle$ and $\langle off, key, idle, gas, accelerating, brake, crashed \rangle$ are executions, but $\langle accelerating, brake, crashed \rangle$ and $\langle accelerating, gas, idle \rangle$ are not.

- All states in Car are reachable.
- In an event-based viewpoint $\langle key, gas \rangle$ and $\langle key, gas, brake \rangle$ are traces of Car but $\langle gas, brake \rangle$ and $\langle gas, key \rangle$ are not.
- In a state-based viewpoint $\langle off \rangle$ and $\langle off, idle, accelerating, crashed \rangle$ are traces of Car but $\langle idle, accelerating \rangle$ and $\langle accelerating, idle \rangle$ are not.

Finally, using the event-based definition of trace, we have

$$Beh(Car) = \{\langle\rangle, \langle key \rangle, \langle key, gas \rangle, \langle key, gas, brake \rangle\}.$$

Using the state-based definition, we have

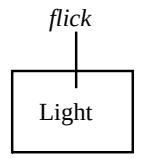
$$Beh(Car) =$$
$$\{\langle\rangle, \langle off \rangle, \langle off, idle \rangle, \langle off, idle, accelerating \rangle, \langle off, idle, accelerating, crashed \rangle\}.$$

In both cases the empty sequence is a member of the behavior.

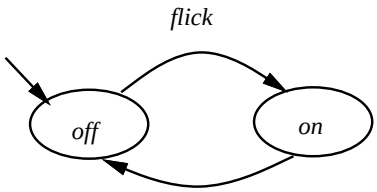
8.4 Infinite Executions and Infinite Behavior

The Car has a finite behavior of finite executions. In general, a state machine can have infinite executions and infinite behavior. An infinite execution is an infinite sequence of alternating states and actions. An infinite behavior is an infinite set of executions. Note that elements in an infinite behavior might all be finite.

Consider a simple light switch whose interface and state transition diagram are shown below:



Light's Interface



Light's State Transition Diagram

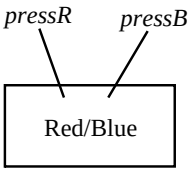
Light $== (\{off, on\}, \{off\}, \{flick\}, \{(off, flick, on), (on, flick, off)\})$. Some executions of Light are:

$\langle off, flick, on \rangle$
$\langle off, flick, on, flick, off \rangle$

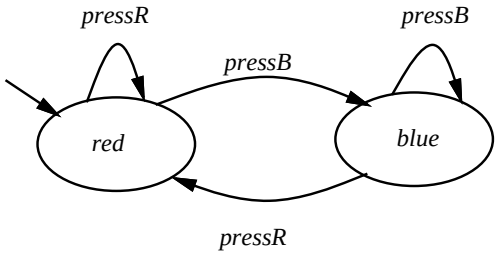
$\langle off, flick, on, flick, off, flick, on \rangle$
...
$\langle off, flick, on, flick, off, flick, on, flick, off, \ldots \rangle$

There are an infinite number of finite executions and the last execution listed above is infinite. Thus *Beh(Light)* is an infinite set of finite and infinite traces.

Here is an example of a state machine with more than one infinite trace:



Red/Blue Light's Interface



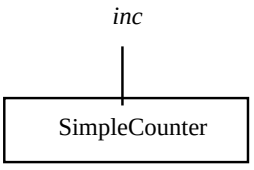
Red/Blue Light's State Transition Diagram

One of the infinite traces (event-based) is the infinite sequence of pressR actions. What are some others?

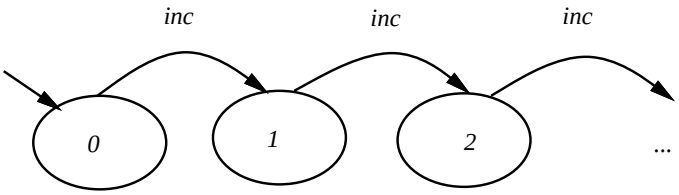
8.5 Infinite States and Infinite State Transitions

In all the examples so far, there has been only a finite number of states, and hence, a finite number of state transitions, i.e., S and δ were finite. We were able to draw state transition diagrams for these state machines in their entirety. For software systems in general we must deal with an infinite number of states and hence an infinite number of state transitions. Their state transition diagrams are impossible to draw out completely (at least in the way we have been drawing them).

The simplest example is an integer counter (which is like a ticking clock), initialized at 0.



SimpleCounter's Interface



SimpleCounter's State Transition Diagram

As soon as we need to model a system that deals with any domain with an infinite set of values, e.g., integers, then we admit the possibility of an infinite number of states. Now we see why most programs, let alone software systems, are infinite state machines.

The SimpleCounter example has just one action, *inc*. It has an infinite number of state transitions because there is an infinite number of states over which the state transition relation, δ, is defined — because there is an infinite number of values that the SimpleCounter can take.

Now, suppose we want to write a description of the state machine using the notation we have seen so far. We would write it something like this:

$$
\begin{array}{l}
\text{SimpleCounter} == (\\
\{0,1,2,\ldots\}, \\
\{0\}, \\
\{inc\}, \\
\{(0,inc,1),(1,inc,2),(2,inc,3),\ldots\} \\
).
\end{array}
$$

The problem is what about those two occurrences of ...? Here, the pattern is clear and we rely on sharing the same intuition as to what goes in those .... In general, we need a way to describe more complex sets. We would like to find a way to characterize an infinite set of things in terms of a finite string of symbols.

Fortunately, predicate logic provides just the notation we need. To characterize the set of all non-negative integers, we write:

$$\{x : \mathbb{Z} \mid x \geq 0\}$$

We can even characterize the set of initial states using a predicate:

$$\{x : \mathbb{Z} \mid x = 0\}$$

We have taken care of the first occurrence of ..., but what about the second? We define the state transition relation, δ, as a set of triples, (s,a,s'), for which the pair of states, s,s', satisfies a given predicate. We do this for each action in A and then take the union of these sets. For example, we define the set of triples, (s,a,s'), that *inc* contributes to δ as follows:

$$\delta_{inc} == \{(s,a,s') : S \times \{inc\} \times S \mid s' = s+1\}$$

where $S == \{x : \mathbb{Z} \mid x \geq 0\}$, as defined above. Since the SimpleCounter has only one action, $\delta = \delta_{inc}$. In general,

$$\delta == \bigcup_{a \in A} \delta_a$$

Notice the power of set notation and predicate logic. Using a few symbols we are able to write out the state transition relation which is defined on an infinite set of states. The critical thing is that because we have a *finite* set of actions, we define a finite number of δ_a, one for each action; in so doing, we are describing the state transition relation, δ, which is a possibly infinite set of state transitions (because we may have to define it over an infinite number of states).

To be pedantic, we put this all together and rewrite the description of SimpleCounter as follows:

$$\begin{array}{l}
\text{SimpleCounter} == (\\
\{x : \mathbb{Z} \mid x \geq 0\}, \\
\{x : \mathbb{Z} \mid x = 0\}, \\
\{inc\}, \\
\{(s, a, s') : \{x : \mathbb{Z} \mid x \geq 0\} \times \{inc\} \times \{x : \mathbb{Z} \mid x \geq 0\} \mid s' = s + 1\} \\
).
\end{array}$$

Technical Aside: The observant reader will notice that we use the state *name* for two purposes: to *name* the state and to give the *value* of the SimpleCounter in that state. In the next chapter we will refine the structure of states that will allow us to make a distinction between names and values.

8.6 Notes

8.6.1 Environment and Interfaces

A system does not live in isolation. It interacts with its *environment*. When we model a system as a state machine we are modeling the *interface* the system has with its environment. Later in the book when we discuss concurrency we will model the environment itself as a state machine and then discuss the interactions between a system and its environment as the behavior of the composition of two state machines. For now we focus our attention on modeling the system and understanding a system's behavior through its state machine model. For now, acting as the system's user, we are the system's environment.

Intuitively the behavior of a state machine captures what the environment *observes* of the system modeled by the machine. Many state machine models differ on what "observes" mean. (That is one reason why we introduced an event-based view and a state-based view.) When we identify what a system's sets of states and actions are we define what its *observable behavior* is. So, when we design a system, especially if it is supposed to be put together with some other system, it is critical that we identify its interface. A rule of thumb to use when we try to nail down a system's interface and we are unsure whether something belongs in the interface or not, is to ask, "Can I observe it?" If so, then "it" has to be modeled somehow; "it" is part of the system's interface.

Input and Output Actions

There are two kinds of interactions between an environment and a system: either the environment might do something to the system to cause a state change (or obtain information about its current state) or the system might do something (or produce something of interest) to the environment and cause a change in the environment. In the light switch example, we can *flick* the light. That is the only action we can do to the light. In the red and blue light example, there are two things we can do: *pressR* and *pressB*. On the other hand, consider a digital clock that displays the time in hours and minutes. Every time a minute passes, the clock's display changes; we can observe each of those state transitions. (Aside: The clock is an example of why one might prefer a state-based view of a trace; we could argue that what we really observe is the clock's state (the display), not the state transitions.)

One way to model this dichotomy of actions is to separate the set of actions into a set of *input actions* and a set of *output actions*. Intuitively, input actions correspond to those things the environment does to the system (like pressing buttons and flicking switches); output actions correspond to those things the system does to the environment (like an ATM handing out cash or a vending machine dispensing candy).

Abstraction

In the Light example we chose not to make a distinction between flicking the light switch up or flicking it down. In both cases we simply named the action of flicking the switch "flick." We made a choice to *abstract* from the direction of flicking the switch. (This abstraction gives the implementor of the Light model the freedom

to implement the light switch with a button that pops in and out rather than with a lever that moves up and down.) When we model a system we often are faced with such design decisions. When faced with the problem of whether something should be modeled at a particular level of abstraction, the question to ask is "Is this level of detail relevant to this level of abstraction?" or more precisely "Is this distinction observable by the environment?" If the answer is "no," i.e., the observer has no way of telling two things apart or we as the system designer do not want to provide the observer a way of telling two things apart, then we should abstract from the difference between the two things. For example, suppose we have an apple, an orange, and an eggplant. We might decide that we do not want an observer to tell the difference between the apple and the orange, but only the difference between fruits (apple or orange) and vegetables (eggplant).

As another example, think of the Car. By the way we chose to model it, the user (as the environment) does not get to see all its states or all its state transitions. For example, to go from the *idle* state to the *accelerating* state we may actually have shifted gears, say from first to second and second to third and so on, before getting to the *accelerating* state. It was our choice to *abstract* from some of its states (e.g., being in third gear) and state transitions (shifting from second to third). In making our choice of what to reveal to the observer, we hid those states and state transitions from the observer because they were irrelevant. The only information we have about the Car is what we reveal to the user. These are design decisions as a modeler that we made.

Some state machine models allow us to make a distinction between *external actions* and *internal actions*. External actions are part of the system's interface and are observable by the system's environment. Internal actions are hidden and not observable.

Actions Revisited

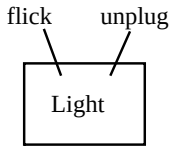
By combining the two points made in the previous two subsections we see that we can divide a set of actions into external and internal actions. We can further divide a set of external actions into input and output actions. Different models of state machines may or may not make these distinctions. (For example, the I/O automata model makes distinctions [**?**].)

8.6.2 A Subtle Point: Actions That Cannot Happen

There are two ways actions cannot happen. Either an action is simply not part of the system's interface; or it is, but no state transition occurs if we try to perform it. For example, when we get a pull-down menu in our browser, only those actions listed are part of the interface. It is simply impossible to try to do an action that is not listed. But also, any action that is "grayed out" indicates that nothing will happen if we select that action: no state transition occurs.

Consider the Light example. Since "unplug" is not in the set of actions, there is no way we can even think of doing such an action. The point is that the only actions that may possibly happen are those that are explicitly given in the state machine's set of actions, A.

However, suppose "unplug" were added to Light's set of actions and *we keep everything else the same*. In particular, its state transitions are still defined only for the action "flick." What happens if we try to unplug the light?



An Unpluggable Light?

There are four reasonable interpretations:

1. Nothing happens.
2. It is undefined. That is, using functional notation, $\delta(\mathit{off}, \mathit{unplug}) = \perp$ and $\delta(\mathit{on}, \mathit{unplug}) = \perp$. ($\perp$, read as "bottom," is the mathematical symbol for "undefined.")
3. It is an error. Chaos can occur (core dumped, machine crashes).
4. It cannot happen.

We will take the fourth interpretation because we can model the first three explicitly if that is the behavior we want. In the first case, we would define the state transition function such that for each state, $s \in \{\mathit{off}, \mathit{on}\}$, (s, unplug, s). In the state transition diagram, for each state, we would draw an arrow from it to itself and label the arrow with the action "unplug." In the second case, we would introduce a special state called $\perp$. In our state transition diagram we would simply draw an arrow from the *off* state to $\perp$ and an arrow from the *on* state to $\perp$. Both arrows would be labeled with the action "unplug." In the third case, we would introduce a state called "error" and draw arrows similar to those in the second case.

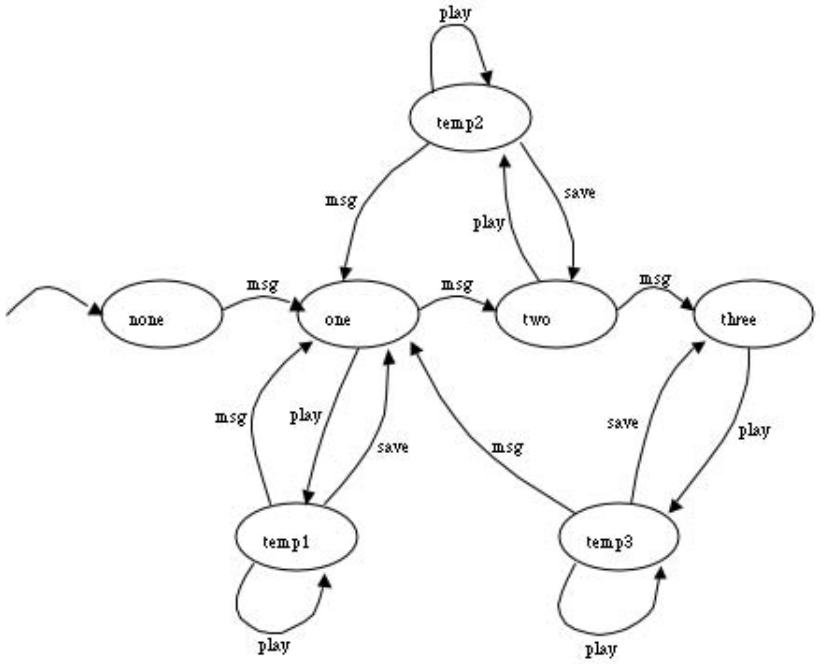
Notice that "bottom" means mathematically *undefined*, which is very different from "error." "Bottom" is like trying to divide by zero in mathematics or where we end up if we find ourselves in an infinite loop in computer science. "Error" is simply a way to denote an error has occurred. In computer science we make a distinction between being in an infinite loop (non-termination) and being in a (terminating) "bad" state (like core dumped) so it makes sense that we make this distinction in software engineering too.

Why we would include an action like "unplug" if we cannot do anything with it? One answer is that we are setting ourselves up so that we can put state machines together. Suppose we combine Unpluggable Light with a machine for which there is a state transition defined on the action "unplug"; then we may be able to make sense of doing the unplug action on the combined machine. We will return to this idea later, when we get to concurrency.

Further Reading

Exercises

1. A certain, simple, answering machine has two buttons, "play" and "save" and can, of course, receive messages. If someone plays the messages and doesn't save them, they are erased/overwritten when the next incoming message is received. The answering machine only holds a specified number of messages; when it reaches full capacity it refuses to accept new messages. The answering machine can be modeled by the state machine, **AnsMachine**, whose state transition diagram is attached.



(a) Give a 4-tuple description for this state machine.
(b) Give three execution fragments of **AnsMachine**, at least one of which is not an execution.
(c) Give both a finite and an infinite execution of **AnsMachine**.
(d) For each of the following, indicate whether or not it is an event-based trace of **AnsMachine**.
   i. $\langle play, save, msg, msg, play \rangle$
   ii. $\langle msg, msg, msg, play, save, msg, msg, msg \rangle$
   iii. $\langle msg, play, save, msg, msg, play, msg, msg, msg \rangle$
(e) Give two examples of state-based traces of **AnsMachine**.
(f) Give two sequences of states which are not state-based traces of **AnsMachine**.
(g) What are the reachable states of this machine?
(h) Is **AnsMachine**'s event-based behavior finite or infinite?
(i) Can a state machine $M = (S, I, A, \delta)$ with an infinite trace have finite behavior? Give an example or explain why not.

Chapter 9

State Machines: Variations

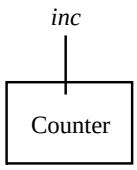
The state machine model presented in the previous chapter is not suitable, appropriate, or natural for modeling all systems. In this chapter we look at different variations of the basic model we have presented so far. (In this chapter, we do not spell out each of the components of the state machine to the same level of detail as in the last chapter. We also introduce some minimal notation for new concepts to make the examples concrete.)

Which model we choose to use depends on what we want to model. We want to choose one that allows us to state as precisely and concisely those things we care about. Some models may make distinctions that we do not care about; some may make assumptions that do not fit our problem. But sometimes which we choose is just a matter of taste. When we decide what model to use we should understand why we are choosing one over another. The choice should be deliberate, not arbitrary.

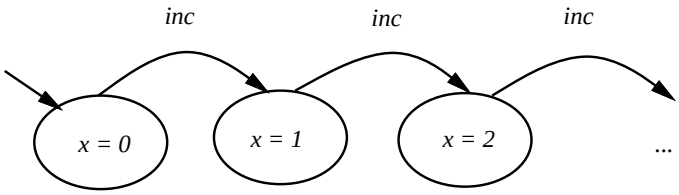
Given a state machine, $M == (S, I, A, \delta)$, in the first three sections, we refine some of M's components. First we give more structure to states in S (Section 9.1), then to actions in A (Section 9.2), and then generalize the functionality of δ (Section 9.3). In the fourth section we show how all these things can be used together. Finally, in the last two sections we discuss other refinements of state machine models that are often seen in practice.

9.1 States

Let's revisit the integer counter example.



Counter's Interface



Counter's State Transition Diagram

In the diagram above, we introduce the variable x to "hold" the value of the integer counter. The notion of a state as having variables that can have values of some type should be familiar to us from our programming experience.

With respect to state machine models, we are refining the notion of what a machine state is; we add some internal structure so that states are more than just named entities like *2*, *off*, or *crashed*. In general, each state in S of a state machine, M, is a record whose field names are variable "names" or "identifiers". Moreover, we assume variables and values are typed much like in a programming language: the values of a variable are drawn from the type associated with the corresponding field name. For example, the state space of our integer counter is defined as:

$$S == [x : \mathbb{Z}]$$

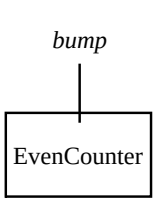
That is, S is the space of all records with field name x and whose values are drawn from the integers.

Since variables correspond to projection functions of records, $x(s)$ denotes the value of the variable x in state s.

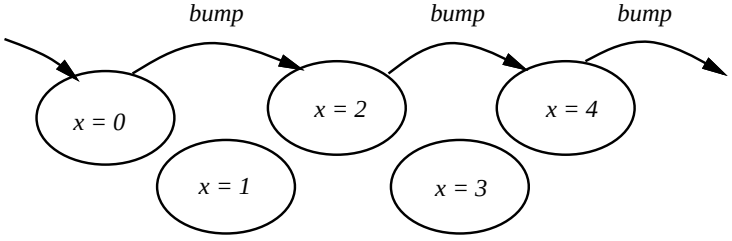
Suppose we want to write the state transition function for the Counter. Then, as defined earlier, let S (the set of states) be the set of records mapping the variable x to an integer value. Then similar to the SimpleCounter, we have

$$\delta_{inc} == \{(s, a, s') : (S \times \{inc\} \times S) \mid x(s') = x(s) + 1\}$$

Now suppose we have a counter that allows state transitions only from states whose value for its state variable, x, is an even number. EvenCounter starts in the initial state $x = 0$ and whenever we *bump* its state, we get to the next even number:



EvenCounter's Interface



Part of EvenCounter's State Transition Diagram

Let EvenCounter's set of states, S, be the same as for Counter. EvenCounter's transition function δ is:

$$\delta_{bump} == \{(s,a,s') : (S \times \{bump\} \times S) \mid even(x(s)) \wedge x(s') = x(s) + 2\}$$

where we assume the predicate *even* has been defined appropriately. (Notice that some states in EvenCounter are unreachable. Which ones?)

Unfortunately writing the state transition function as predicates over sets of pairs of states and writing $x(s)$ and/or $x(s')$ whenever we want to refer to the value of a state variable becomes pretty unwieldy quickly. By introducing two keywords, we write the state transition function for each action in a more readable notation. Here is what we write for Counter's *inc* action:

inc
pre *true*
post $x' = x + 1$

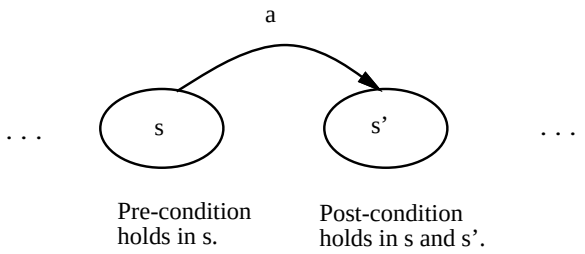
and for EvenCounter's *bump* action:

bump
pre $even(x)$
post $x' = x + 2$

The first line, which we call the *header*, gives the name of the action whose state transition behavior we describe in the subsequent two lines. The second line gives a *pre-condition*, which is just a predicate, and the third line gives a *post-condition*, which is just another predicate. The interpretation of the pre- and post-conditions is: In order for the state transition to occur from the state s to the state s' the pre-condition must hold in s; after the state transition occurs, then the post-condition

must hold in s'. The state transition cannot occur if the pre-condition is not met. Post-conditions in general need to talk about the values of state variables in both the state before the state transition occurs (the "pre-state") and the state after it occurs (the "post-state"). We use an unprimed variable to denote the value of the variable in the pre-state and a primed variable to denote its value in the post-state. So, x' really stands for $x(s')$; x, for $x(s)$.

Here is how to visualize what the pre- and post-conditions capture:



For action a to occur the pre-condition must hold in s. If a occurs, the post-condition must hold in s and s'.

In the Counter example, the *inc* action has the trivial pre-condition, "true." This means that the *inc* action can be performed in any state in S. EvenCounter's *bump* action has a non-trivial pre-condition. Another typical non-trivial pre-condition is requiring that a *pop* action not be performed on an empty stack. We'll see other examples of non-trivial pre-conditions later. *Inc*'s post-condition says that the value of the integer counter is increased by one from its previous value; *bump*'s post-condition says that the value is increased by two.

In general, for a given $M == (S, I, A, \delta)$, the template[1] we use to describe δ_{action} is

action
pre $\Phi(v)$
post $\Psi(v, v')$

where *action* is in A, and Φ and Ψ are (state) predicates over a vector, v, of state variables. The above template stands for the following part of the definition of the state transition function, δ:

$$\delta_{action} == \{(s, a, s') : S \times \{action\} \times S \mid \Phi[v := v(s)] \wedge \Psi[v' := v(s'), v := v(s)]\}$$

[1] We will be elaborating on this template throughout this chapter.

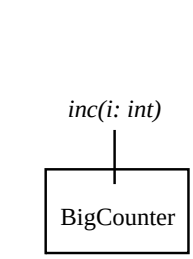
In English this says that the precondition (Φ) has to hold in the pre-state *and* the post-condition (Ψ) has to hold in the pre/post-states[2].

Other interpretations of pre/post-condition specifications are possible. We are just giving one reasonable one here.[3] For example, another common one is where the conjunction used above is replaced by an implication. Under this interpretation, which is used in Z and Larch, if the pre-condition does not hold and we try to do the action then anything can happen, i.e., "all bets are off." We could end up in an unexpected state, an error state, or an undefined state.

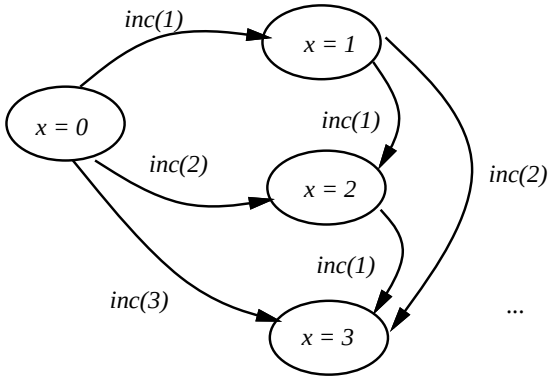
9.2 Actions

9.2.1 Actions with Arguments

Now suppose we want to let the integer counter's *inc* action take an integer argument. We see it is even more difficult to draw BigCounter's state transition diagram, only part of which is shown here:



BigCounter's Interface



Part of BigCounter's State Transition Diagram

It is much easier and more concise to write the state transition function for *inc* as follows:

$$
\begin{array}{l}
inc(i\colon \mathbb{Z}) \\
\quad \textbf{pre } true \\
\quad \textbf{post } x' = x + i
\end{array}
$$

[2]Recall that the post-condition is defined over two states.

[3]This one also is consistent with the discussion (in the Chapter 8) about actions that cannot occur.

We extend the header in the specification to include a list of input arguments (and their types). We intentionally choose syntax to look like programming language notation.

The technical term for what we do is *lambda abstraction*. Using a single template we define an infinite set of functions, one for each integer i. Instead of defining separate actions $inc_1, inc_2, \ldots$ we define a family of actions $inc(i)$.

According to the above specification, there is nothing preventing the input integer argument that we hand to *inc* from being negative. Suppose we want the counter to always increase in value, never decrease? we capture this requirement by strengthening the pre-condition:

inc(i: $\mathbb{Z}$*)*
pre $i > 0$
post $x' = x + i$

We call this new machine FatCounter. It is an example of a state machine with an action that has a non-trivial pre-condition.

9.2.2 Actions with Results

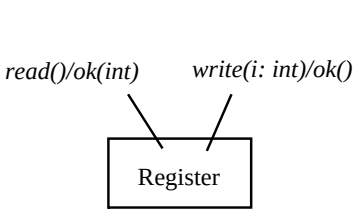
Sometimes actions produce results of interest to the external observer. When we query our checking account balance, we expect the result to be displayed on the ATM screen or printed on a piece of paper.

Here is a Register with *read* and *write* actions. *Read* returns the value of the register; *write* takes an argument and modifies the register's state. Its initial value is the integer value 0.

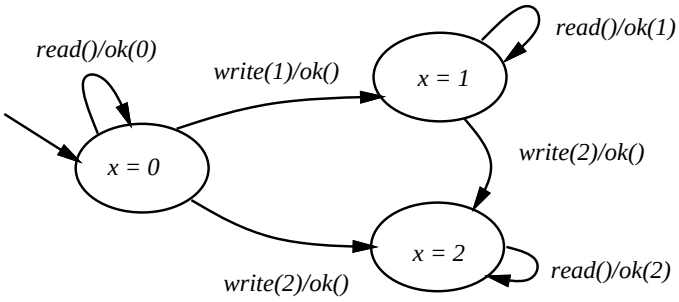
Here are the specifications of the actions:

read()/ok($\mathbb{Z}$*)*
pre *true*
post $result = x$

write(i: $\mathbb{Z}$*)/ok()*
pre *true*
post $x' = i$



Register's Interface



etc.

Part of the Register's State Transition Diagram

The first thing to notice is that we introduce the word "ok" in the header. We do this for two reasons. The first is that we want to set ourselves up so that we have a convenient way to distinguish normal termination from exceptional termination of a procedure, a feature supported by most advanced programming languages. More on this later. The second is a trivial point: For symmetry, we prefer writing *read()/ok(1)* instead of *read()/1*. Think of the instance of an action as a procedure call. Then the state transition labeled *read()/ok(1)* corresponds to calling the *read* procedure and getting the integer 1 back.

We are simply adding more structure to actions. In general, a state transition is an *action instance*, which is a pair of an *invocation event* and *response event*. An *invocation event* is the name of the action plus the values of its input arguments; a *response event* is the name of the termination condition (e.g., ok) plus the value of its result.

An execution of a state machine is a sequence of alternating states and action instances. Some executions of the Register machine are:

$$\langle x = 0, write(1)/ok(), x = 1, read()/ok(1), x = 1 \rangle$$
$$\langle x = 0, write(1)/ok(), x = 1, read()/ok(1), x = 1, read()/ok(1), x = 1, write(5)/ok(), x = 5, read()/ok(5), x = 5 \rangle$$
$$\langle x = 0, write(1)/ok(), x = 1, write(7)/ok(), x = 7, write(9001)/ok(), x = 9001 \rangle$$

For the above executions, we have the following (event-based) traces:

$$\langle write(1)/ok(), read()/ok(1) \rangle$$

$\langle write(1)/ok(), read()/ok(1), read()/ok(1), write(5)/ok(), read()/ok(5)\rangle$
$\langle write(1)/ok(), write(7)/ok(), write(9001)/ok()\rangle$

What would we have for a state-based definition of trace?

The second thing to notice in the above specification is that we give in parentheses the type of the result, if any, of each action. The *read* action returns an $\mathbb{Z}$ value; *write* does not return anything.

The third thing to notice is that in the post-condition we use a special reserved word *result* to stand for the return value. This trick works fine as long as an action produces only one result. It generalizes in the obvious way in case an action produces more than one result.

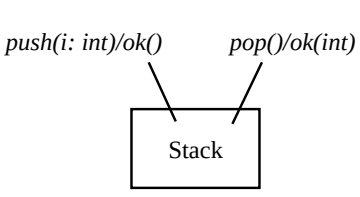
Technical Aside 1: There is a subtle difference between an action, which is a member of the finite set A, and an action instance, which is a member of the possibly infinite set of state transitions, as defined by δ. This difference is analogous in programming to the difference between the definition (declaration) of a procedure and a call (invocation) of it.

Technical Aside 2: There are state machine models that treat invocation events and response events as separate kinds of actions. Invocation events could be viewed as *input actions*; response events, as *output actions* (see Chapter 8). These models are mainly used for modeling classes of concurrent and distributed systems. For now, there is no compelling reason to treat them separately.

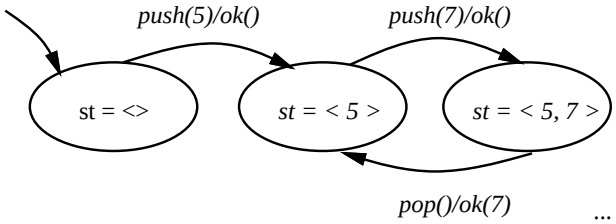
9.2.3 Actions that Terminate Exceptionally

Many advanced programming languages support exception handling and thus we should be able to specify the interface of a program that can raise exceptions.

Consider the following Stack machine:



Stack's Interface



Part of Stack's State Transition Diagram

with *push* and *pop* specified as follows:

push(i: $\mathbb{Z}$)/ok()
pre *true*
post $st' = st \frown \langle i \rangle$

pop()/ok($\mathbb{Z}$)
pre $st \neq \langle\rangle$
post $st = st' \frown \langle result \rangle$

Here is how we specify a more robust interface to Stack that allows *pop* to raise the exception *empty* if we try to perform the *pop* action on an empty stack (*push* stays the same):

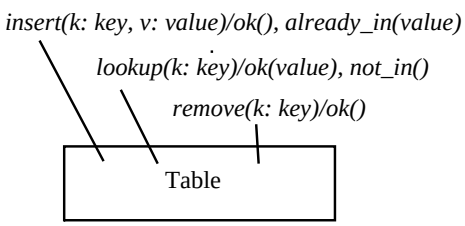
pop()/ok($\mathbb{Z}$), empty()
pre *true*
post $(st \neq \langle\rangle \Rightarrow (st = st' \frown \langle result \rangle \wedge terminates = ok)) \wedge$
$(st = \langle\rangle \Rightarrow (st = st' \wedge terminates = empty))$

The first thing to notice is the addition of the name of the exceptional termination condition, *empty*, in the header. For each termination condition (normal and exceptional), we allow some kind of result to be returned; here *empty* does not return any result.

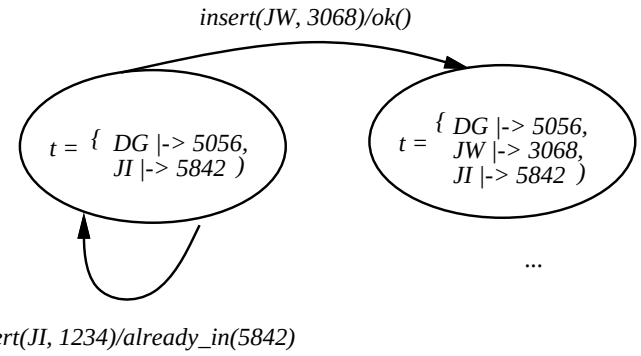
The second thing to notice is the special reserved word, *terminates*, which we introduce to hold the value of the termination condition ("ok" for normal termination and one of the exceptions listed in the header for an exceptional termination).

From a software engineering perspective, there is usually a close correlation between pre-conditions and exceptions. It is common to transform a "check" in the pre-condition to be a "check" in the post-condition. From our programming experience, this is the same as placing the responsibility on the callee rather than the caller of a procedure. With a pre-condition it is the caller's responsibility to check that the state of the system satisfies the pre-condition before calling the procedure; with an exception in lieu of the pre-condition, it is the callee's responsibility by performing a (run-time) check and raise an exception in case the state of the system violates the condition.

Here is an example where upon exceptional termination an interesting value is returned. Consider a Table machine, which stores keys and values. The state variable, *t*, stores the CMU telephone extensions of the 15-671 staff members:



Table's Interface



Part of Table's State Transition Diagram

We model the state of the Table as a function from keys to values [4]. The *insert* action returns an exception *already_in* if there already exists a value associated with the key for which we are trying to insert a particular key-value pair, (k, v), and if so, it returns the current value bound to that key, k:

insert(k: key, v: value)/ok(), already_in(value)
pre *true*
post $(k \notin dom\, t \Rightarrow (t' = t \cup \{k \mapsto v\} \wedge terminates = ok)) \wedge$
$(k \in dom\, t \Rightarrow (t' = t \wedge terminates = already\_in \wedge result = t(k)))$

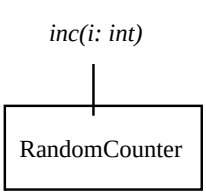
9.3 Nondeterminism

So far, δ has been a function, that is, for each state, s, and action, a, δ mapped us to at most one next state. Suppose we have a RandomCounter machine with an *inc* action that takes an integer argument:

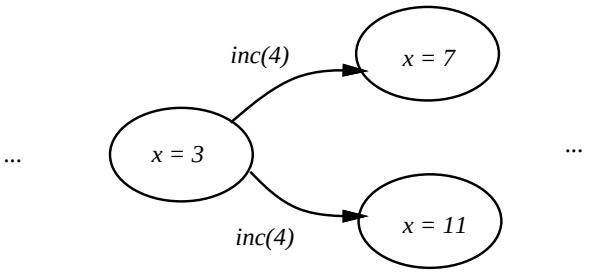
Here is the specification of *inc*:

inc(i: $\mathbb{Z}$)
pre $i > 0$
post $x' = x + i \vee x' = x + 2i$

[4] We leave it to the reader to formalize this state machine. We will see something similar when we get to the Birthday Book example in Z.



RandomCounter's Interface



Part of RandomCounter's State Transition Diagram

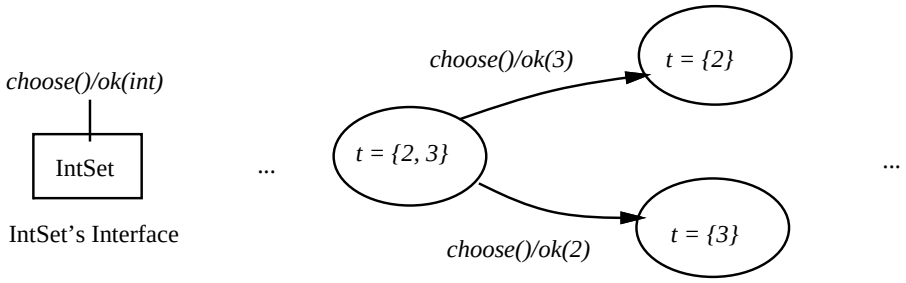
According to the specification, *inc* increments the counter's value either by the value of its argument *i* or by twice that value. Thus, there are two possible states that we might end up in after doing the *inc* action given some integer argument. Since there is more than one, δ needs to map to a set of states. (We view δ as a function from (state, action) pairs to a set of states, rather than as a relation on (state, action, state) triples.):

$$\delta([x = 3], inc(4)) = \{[x = 7], [x = 11]\}$$

As an observer, we do not know which state transition will occur when we feed *inc* the integer 4. We must be prepared to deal with either possibility. The choice of which post-state is taken is made by the machine itself. Notice that the nondeterminism shows up in the specification of *inc* in the use of logical disjunction in the post-condition.

9.4 Putting Everything So Far Together

Suppose we have an integer set, *t*, and in its interface is a *choose* action that does not take any arguments and removes and returns an element from *t*.



Part of IntSet's State Transition Diagram

where *choose* is specified as follows:

$$\begin{array}{l} \textit{choose()/ok}(\mathbb{Z}) \\ \quad \textbf{pre } t \neq \emptyset \\ \quad \textbf{post } \textit{result} \in t \wedge t' = t \setminus \{\textit{result}\} \end{array}$$

The nondeterminism shows up in the specification for *choose* in the use of the set membership operator ($\in$) in the post-condition. We do not know which element of t will be returned; we know only that some element will be returned.

Notice that the labels on the arcs in the state transition diagram above are different (by what is returned by *choose*); however, most people would still view the state transition function as nondeterministic because they would *abstract* from the actual value returned. In other words they would define δ something like this:

$$\delta([t = \{2,3\}], \textit{choose}()/\textit{ok}(\mathbb{Z})) = \{[t = \{2\}], [t = \{3\}]\}$$

Finally, in a programming language that supports exception handling we would probably export a more robust interface for the *choose* action:

$$\begin{array}{l} \textit{choose()/ok}(\mathbb{Z}), \textit{ empty()} \\ \quad \textbf{pre } \textit{true} \\ \quad \textbf{post } (t \neq \emptyset \Rightarrow (\textit{result} \in t \wedge t' = t \setminus \{\textit{result}\} \wedge \textit{terminates} = \textit{ok})) \wedge \\ \qquad (t = \emptyset \Rightarrow (t' = t \wedge \textit{terminates} = \textit{empty})) \end{array}$$

In general, the template we use for each *action* in A of $M = (S, I, A, \delta)$ is:

$$\begin{array}{l} \textit{action(inputs)/term}_1(\textit{output}_1), \ldots, \textit{term}_n(\textit{output}_n) \\ \quad \textbf{pre } \Phi(v) \\ \quad \textbf{post } \Psi(v, v') \end{array}$$

where *inputs* is a list of arguments and their types, $term_i$ is the name of one of i termination conditions (including "ok") and $output_i$ is the type of the result corresponding to $term_i$. Φ and Ψ are state predicates as defined earlier. The reserved identifiers we use are:

- *ok*, used in the header, to denote normal termination,
- *result*, used in the post-condition, to denote the value returned by an action, and
- *terminates*, used in the post-condition, to denote the value of termination condition. Its value can be any of the $term_i$, including "ok," listed in the header.

We are glossing over a number of technicalities here regarding state variables to store input arguments and the type of the result returned, depending on how an action terminates.

Finally, for simplicity, let's assume actions return only normally unless specified otherwise (by explicitly listing exceptional conditions in their headers). Under this default case, we avoid clutter in our specifications by not always having to write "*terminates* = *ok*" in the post-conditions of our actions.

9.5 Finite State Automata

9.5.1 Deterministic FSA

Readers who have taken an automata theory course in computer science have already come across state machines. A *deterministic finite state automata (FSA)* is defined as follows:

$$M == (S, I, F, A, \delta)$$

where S is a finite set of states; $I \subseteq S$, the set of initial states, is restricted to be a singleton set; $F \subseteq S$ is a finite set of *final* states; A is a finite set of actions, and $\delta : S \times A \nrightarrow S$ is a function.

FSA terminology	State Machine terminology
alphabet	actions
string	trace†
language L(M)	behavior Beh(M)

Here "trace†" means the trace of an execution ending in a final state. Sometimes the language of M is called *regular* or a *regular set* because it can be expressed as a regular expression using $*, \cup$, and concatenation. Just as with the behavior of a machine, the language of an FSA can be infinite, i.e., there might be an infinite number of strings accepted by M.

9.5.2 Nondeterministic FSA (NFSA)

The only interesting thing to say here is to refresh our knowledge about automata theory: It is always possible to turn an NFSA into a deterministic FSA; and of course, every deterministic FSA is an NFSA.

9.6 Finite Executions and Infinite Behavior

Recall the red and blue light example where the behavior of the light is infinite because there are an infinite number of traces associated with the light. Also, some of those traces are infinite (e.g., pressing red forever).

In some models of state machines, in particular, Communicating Sequential Processes (CSP), infinite traces are not included in the machine's behavior. This kind of model is sometimes called the *finite-trace model*. The behavior of the machine is defined to be a possibly *infinite* set of *finite* traces.

Why is this a reasonable model? It has a simple structure which has some nice mathematical properties. Using this model may help simplify our thinking about the system; we know every trace in the behavior is finite so we do not have to worry about infinite traces. With a model that has both finite and infinite traces, whenever we prove some property about the behavior it describes, we usually have to structure our proof into two parts, one to handle the finite trace case and one to handle the infinite trace case. But perhaps the most compelling intuitive argument for this model is that we can never *see* an infinite execution; if we cannot observe this behavior then why should we model it?

What are some of the disadvantages of this model? Because we cannot talk about infinite traces, the biggest disadvantage is that we cannot talk about certain properties like deadlock and fairness. To do so requires adding a lot of complicated structure to either traces or behaviors of a state machine.

Further Reading

Exercises

1. Consider a TV remote control that allows the user to select the channel to be viewed, add or remove a "parental block" to a channel that prevents the channel from being displayed (removal requires a password), and enter a password to allow a blocked channel currently selected to be displayed. If a blocked channel is selected, the channel is not initially displayed. The user may choose to select a different channel or may enter the password to display the channel. If the incorrect password is entered, the channel is not displayed. If the correct password is entered, the channel is displayed.

   Your task is to model the described functionality of the remote control. That is,

   (a) Specify the set of states
   (b) Specify the pre- and post-conditions for each action

   Your solution should satisfy the following requirements:

   i. Do *not* model any functionality other than that described above. In particular, assume that the correct password is fixed and cannot be changed.
   (a) Assume that the set of channels is $Channels == \{n : \mathbb{N} \mid 1 \leq n \leq 100\}$.
   ii. You may only use the following actions in your model:
      - *select*: Select a channel for viewing
      - *correctpw*: Enter correct password
      - *incorrectpw*: Enter incorrect password
      - *addblock*: Block selected channel
      - *removeblock*: Unblock selected channel
   iii. If (and only if) the requirements are ambiguous, state any assumptions that you made to resolve those ambiguities.

Chapter 10

Reasoning About State Machines

Given a state machine model of a system, we can do some formal reasoning about properties of the model. It is important to remember that we are proving some property about the *model of the system*, not the *system* itself. If the model is "incorrect" then we may not be able to prove anything useful about the system. Worse, if the model is "incorrect" then we may be able to prove something that has no correspondence to the real system. However, we hope that we have modeled our system properly so that whatever we prove about our model is true of the system being modeled.

But then, why not just reason about the system itself directly? One reason is that it is often impossible to reason about the system itself because it is too large, too complex, or too unwieldy. Another is that we may be interested in one aspect of the system and want to abstract from the irrelevant aspects. Another is that we may not actually have a real system; our model could simply be a high-level design of a system we might build and we want to do some reasoning about our design before spending the dollars building the real thing. Another is that it may be impossible to get our hands on the system (maybe it is proprietary). Another is that it may be impossible for us to run the system to check for the property because of its safety-critical side-effects (like setting off a bomb). So, in some cases, the best we can do is reason about a model of the system, and not the system itself.

In this chapter we discuss a few kinds of properties we might want to reason about a state machine model. The most important of these is *invariant properties*, properties that are true of every reachable state in the system.

10.1 Invariants

An *invariant* is a predicate that is true in all states. In the context of state machines, we usually care that an invariant is true in all *reachable* states. The statement of an invariant, θ, in full generality looks like:

$$\forall e : executions \bullet \forall s : S \bullet s \,\mathbf{in}\, e \Rightarrow \theta(s)$$

where θ is a predicate over variables in s and **in** is a predicate that says whether a state is in an execution. Normally the universal quantification over all executions and the condition that s be in e is omitted (it is understood):

$$\forall s : S \bullet \theta(s)$$

Sometimes we also omit the universal quantification over s as well because it is also understood:

$$\theta(s)$$

For example, here is an invariant for the Counter example given in Chapter 9:

$$x(s) \geq 0$$

which says that in all states, x's value is greater than or equal to 0. We know this is true because initially x's value is 0 and because the *inc* action always increases x's value by 1. Since *inc* is Counter's only action, there's no other way to change x.

10.1.1 Proving an Invariant

How do we show that a predicate is an invariant? There are lots of techniques. If the state machine is finite, we can do an exhaustive case analysis and show that it holds for every state. This technique is fine if there are a small number of states or if we have a tool called a *model checker* handy (see Chapter **??**).

If the state machine is infinite, we must resort to something else. We now sketch out three techniques. They can all also be used if the state machine is finite. Technique C is the one most often used in practice.

A. Use induction over states in executions.

When we have to reason about an infinite domain, the technique that should spring to mind is *induction*. Induction is especially appropriate when there is a natural, often recursive, structure to the domain. Since what we want to prove is that a property is true of every state of every execution of the state machine, then we induct over the states in the sequence of states of every execution. Recall that an execution looks like:

$$\langle s_0, a_1, s_1, a_2, \ldots, s_{i-1}, a_i, s_i, \ldots \rangle$$

Then to prove a property, θ, is invariant requires that for every execution we:

1. Base Case: Show it holds in the initial state s_0, and
2. Inductive Step: Assume it holds in state s_{i-1} and show that it holds in state s_i.

B. Show that the state space predicate implies the invariant.

If we are lucky the predicate that we use to define the state space is stronger than the invariant we are trying to prove. So regardless of whether a state is reachable or not, we can prove the invariant holds:

$$P \Rightarrow \theta$$

where P is the predicate describing the set of states. If it is true of every state, then certainly it is true of every reachable state.

For example, recall in the Counter example, the predicate P is simply $x(s) \geq 0$. Hence we can trivially show the invariant property holds:

$$x(s) \geq 0 \;\Rightarrow\; x(s) \geq 0$$

C. Use a proof rule using pre- post-condition specifications.

Technique A requires that we reason in terms of first principles — using structural induction (over states in executions) — which can sometimes be cumbersome. And, usually, of course, we are not so lucky as in Technique B. Technique C is an alternative inductive proof strategy that is usually more manageable than Technique A: we do a case analysis of all actions, which indirectly proves the same thing as in Technique A.

To argue that the Counter's invariant holds, we need to show that the invariant holds in each initial state and then for each action show that if the invariant holds in its pre-state, it holds in the post-state. (Here again, is another good reason to have only a *finite* set of actions.) In general, we have a proof rule that looks like:

$$\frac{\begin{array}{c}\forall s : I \bullet \theta(s) \\ \forall a : A \bullet \forall s, s' : S \bullet (s, a, s') \in \delta \wedge \Phi(s) \wedge \theta(s) \wedge \Psi(s, s') \Rightarrow \theta(s')\end{array}}{\forall s : S \bullet \theta(s)}$$

where I is the set of initial states and A is the set of actions. Φ and Ψ are the pre- and post-conditions of a, respectively. Or, said in English:

1. Show that θ is true for each initial state.
2. For each action,
   - assume
     - the pre-condition Φ holds in the pre-state,
     - the invariant θ holds in the pre-state, and
     - the post-condition Ψ holds in the pre- and post-states, then
   - show
     - θ holds in the post-state.
3. Conclude that θ is an invariant.

The two main proof steps are sometimes called (1) *establishing the invariant* (true in initial states) and (2) *preserving the invariant* (assuming it is true in a pre-state and showing that each action's post-state preserves it).

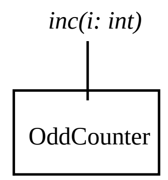
What is the rationale for this proof rule? First, notice we care only about reachable states (that is why the δ appears above). Second, consider any execution of a state machine:

$$\langle s_0, a_1, s_1, a_2, \ldots, s_{i-1}, a_i, s_i, \ldots \rangle$$

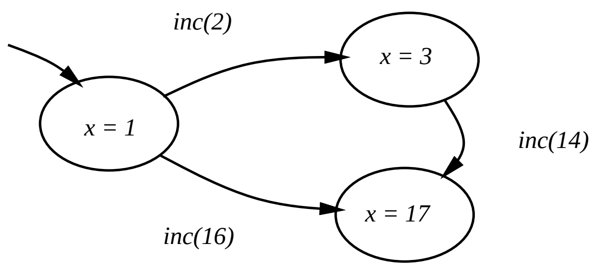
As in Technique A we need to make sure that θ holds in s_0 (establishing the invariant). Also, for any pair of successive states, (s_{i-1}, s_i), in the execution, if we assume θ holds in s_{i-1} then we need to show it holds in s_i. Since the only way we can get from any s_{i-1} to the next state s_i is by one of the actions a in A, then if we show the invariant is preserved for each a, we have shown for each reachable state the invariant holds.

10.1.2 OddCounter

Let OddCounter be the state machine:



OddCounter's Interface



Part of OddCounter's State Transition Diagram

More precisely,

OddCounter $==$ (
$[x : \mathbb{Z}]$,
$\{[x = 1]\}$,
$\{inc(i : \mathbb{Z})\}$,
δ $==$

inc(i: $\mathbb{Z}$*)*
pre i is even
post $x' = x + i$

).

The invariant we want to prove is that OddCounter's state always holds an odd integer:

$$\theta \;==\; x \text{ is odd.}$$

Intuitively we see this is true because

1. It is true of the initial state (1 is an odd integer).
2. For the action *inc*:
   - We assume:
     - i is even (i.e., the pre-condition holds in the pre-state),
     - x is odd (i.e., the invariant holds in the pre-state), and

  - $x' = x + i$ (i.e., the post-condition holds in the pre- and post-states)
- We need to show that x' is odd.
  - This is true since from facts about numbers, an odd number (x) plus an even number (i) is always odd.

10.1.3 Fat Sets

This example has two purposes. One is to give another example of an invariant. The other is to hint at how state machines are appropriate models of abstract data types, as we might implement in our favorite object-oriented language.

Here is a description of a FatSet abstract data type modeled as a state machine:

- $S == [t : \mathbb{P}\,\mathbb{Z}]$. The state variable, t, is a set of integers.
- $I == \{s : S \mid \#t(s) = 1\}$. The set of initial states is the set of states in which t is a singleton set. Notice that there are an infinite number of initial states.
- $A == \{union(u : \mathbb{F}\,\mathbb{Z})/ok(), card()/ok(\mathbb{Z})\}$ (Recall that $\mathbb{F}\,\mathbb{Z}$ is the set of finite subsets of $\mathbb{Z}$.)
- $\delta ==$

$$
\begin{array}{l}
union(u : \mathbb{F}\,\mathbb{Z})/ok() \\
\quad \textbf{pre } u \neq \emptyset \\
\quad \textbf{post } t' = t \cup u
\end{array}
$$

$$
\begin{array}{l}
card()/ok(\mathbb{Z}) \\
\quad \textbf{pre } true \\
\quad \textbf{post } result = \#t \wedge t' = t
\end{array}
$$

Suppose we want to show the property that the size of the FatSet t is always greater than or equal to 1:

$$\theta \;==\; \#t(s) \geq 1$$

Here's an informal proof:

1. It is true of all initial states since the size of all singleton sets is 1.
2. We need to show the invariant is preserved for each of *union* and *card*.
   (a) (*union*). Assume

- u is nonempty (i.e., the pre-condition holds in the pre-state),
- t's size is ≥ 1 (i.e., the invariant holds in the pre-state), and
- $t' = t \cup u$ (i.e., the post-condition holds in the pre- and post-states)

Then we need to show that the size of t' is ≥ 1. This is true since taking the union of two non-empty sets (t and u) is a non-empty set, whose size is ≥ 1.

(b) (*card*). Assume

- t's size is ≥ 1 (i.e., the invariant holds in the pre-state), and
- $result = \#t \wedge t' = t$ (i.e., the post-condition holds in the pre- and post-states)

Then we need to show that the size of t' is ≥ 1. This is true since $t' = t$ and t's size is ≥ 1.

Notice how awful it would be if we had to write out these proof steps in gory detail!

Two points about this example:

- Notice that the only interesting part of the proof above was for *union*, the only action that changes the value of any state variable. An action that changes the state of some state variable is called a *mutator*. We did not have anything to prove for *card* because the action does not change the state of any state variable. We call such an action a *non-mutator*. In practice, we need to show only for mutators that the invariant is preserved because non-mutators by definition cannot change state. So if the invariant holds before the non-mutator is called (pre-state); then it holds afterwards (post-state).
- Second, by design, the value space for t includes "unreachable values." In particular, t can never be the empty set because it starts out non-empty and always remains non-empty. But, remember, we need to show the invariant holds of only reachable states.

Now, suppose we add a *delete* action to the FatSet example. Let *delete* have the following behavior:

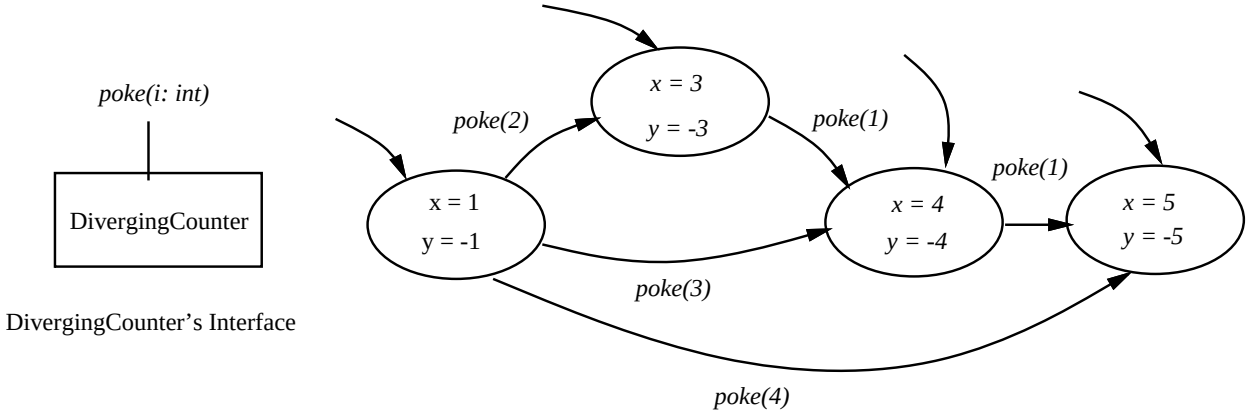
delete(i:$\mathbb{Z}$)/ok()
pre $t \neq \emptyset$
post $t' = t \setminus \{i\}$

Then the invariant would no longer hold because if *delete* were called in any state where $t = \{i\}$ (where i is the argument to *delete*) then the size of t' would be 0.

10.1.4 Diverging Counter

This example has two purposes. One is to give another example of an invariant. The other is to give an example of a state machine with more than one state variable. The invariant property captures an invariant relationship that we want to maintain between these two state variables.

Here's a two-integer counter with state variables, x and y, whose values get further and further apart as we *poke* the machine.



Part of DivergingCounter's State Transition Diagram

Here's a description of the DivergingCounter:

DivergingCounter = (
$[x, y : \mathbb{Z}]$,
$\{s : [x, y : \mathbb{Z}] \mid x(s) = -y(s)\}$,
$\{poke(i : \mathbb{Z})\}$,
$\delta ==$

$poke(i : \mathbb{Z})$
pre $i > 0$
post $x' = x + i \wedge y' = y - i$

).

Notice that all the states drawn above are legitimate initial states. So is the state where x and y are both initialized to 0.

The invariant maintained by DivergingCounter is:

$$\theta \quad == \quad x(s) + y(s) = 0$$

To the reader: Can you prove it?

10.1.5 Comment on Notation

In invariants it is awkward writing $x(s)$ and $y(s)$ all the time since s is universally quantified. So more typically we see invariants expressed directly in terms of the state variables. For the DivergingCounter, we see its invariant in this more readable format:

$$\theta \quad == \quad x + y = 0$$

We use this syntactic sugar in the pre- and post-conditions already.

10.2 Constraints

An invariant is supposed to be true of every *state* of every execution of a state machine. We might ask is there a corresponding notion for *state transitions*? The answer is "yes," though because it is not as commonly discussed in the literature, there is no common term for such a property. Some people might simply call it another kind of invariant, one over state transitions rather than states. To avoid confusion with state invariants, we call it a *constraint*. This word is not standard; also, others may use "constraint" to mean something different.

Consider any execution of a state machine:

$$\langle s_0, a_1, s_1, a_2, \ldots, s_{i-1}, a_i, s_i, a_{i+1}, s_{i+1}, \ldots, s_{j-1}, a_j, s_j, \ldots \rangle$$

A *constraint* is a predicate that is true in all pairs of states, s_i and s_j, in every execution, where s_j follows s_i, but need not immediately follow it (s_j does not have to be s_{i+1}).

The statement of a constraint, χ, in full generality looks like:

$$\forall e : \mathit{executions} \bullet \forall s_i, s_j : S \bullet (s_i \,\mathbf{in}\, e \wedge s_j \,\mathbf{in}\, e \wedge i < j) \Rightarrow \chi(s_i, s_j)$$

As for statements of invariants, we omit (because it is understood) the universal quantification over executions and states; the condition about s_i and s_j both being states in the execution; and the condition that s_i precedes s_j in the execution.

A constraint that holds for the Counter example is that its value always strictly increases:

$$x(s') > x(s)$$

Does this constraint (appropriately restated) hold for SimpleCounter? BigCounter (careful!)? FatCounter? RandomCounter?

10.2.1 Proving Constraints

If the constraint we are interested in proving is a "transitive" property (if it holds for s_i and s_{i+1} and for s_{i+1} and s_{i+2} then it holds for s_i and s_{i+2}) then we can use the following proof rule to show the constraint holds of any execution of the state machine:

$$\frac{\forall a : A \bullet \forall s, s' : S \bullet (s, a, s') \in \delta \wedge \Phi(s) \wedge \Psi(s, s') \;\Rightarrow\; \chi(s, s')}{\chi(s_i, s_j)}$$

where A is the set of actions and s_i and s_j are quantified and qualified as described above. Or, in English:

1. For each action $a \in A$,
   - assume
     - Φ holds in the pre-state,
     - Ψ holds in the pre- post-states, and
   - show
     - χ holds in the pre- post-states.
2. Conclude that χ holds of all pairs of states in all executions.

What is the rationale for this proof rule? First, again we care only about reachable states. Second, consider any execution of my state machine:

$$\langle s_0, a_1, s_1, a_2, \ldots, s_{i-1}, a_i, s_i, a_{i+1}, s_{i+1}, \ldots, s_{j-1}, a_j, s_j, \ldots \rangle$$

If we show that χ holds over any pair of successive states, (s_i, s_{i+1}), i.e., every single state transition, then surely it holds over any pair of states, (s_i, s_j), where $i < j$. To show that it holds in any pair of successive states, we need only consider every possible action, which is the only way we can get from s_i to s_{i+1}. For each action, we need to make sure that the conjunction of its pre- and post-condition predicates imply the constraint.

10.2.2 Fat Sets Again

Here is a constraint for the FatSet example:

$$\chi \;==\; \forall x : \mathbb{Z} \bullet x \in t(s_i) \Rightarrow x \in t(s_j)$$

where s_i and s_j are implicitly defined and qualified as usual. This says that once an integer gets added to my set t it never disappears. We know this constraint holds because there is no way to delete elements from the set.

Notice that saying that the cardinality of t always strictly increases:

$$\text{WRONG!} \qquad \chi \;==\; \#t(s_i) > \#t(s_j)$$

is not a constraint for FatSet. It does not hold since taking the union of two sets may not necessarily increase the size of either.

10.2.3 MaxCounter

Constraints are useful for stating succinctly when things do not change in value. Consider the following MaxCounter machine whose state variable x can never exceed the value of the other state variable *max*. *max* is initialized to 15 and its value never changes.

MaxCounter = (
$[x, max : \mathbb{Z}]$,
$\{[x = 0;\ max = 15]\}$,
$\{inc(i : \mathbb{Z})\}$,
δ ==

$inc(i : \mathbb{Z})$
pre $x + i \leq max$
post $x' = x + i \wedge max' = max$

).

It is trivial to show the following constraint:

$$\chi \;==\; max(s_i) = max(s_j)$$

This kind of example may look simplistic but it generalizes to any system where we want to ensure that some state variable never changes. When we have a huge state space (as is typical of software systems), very often we are careful to state how some state variable changes but forget to say what state variables do not change. Constraints are a nice way to describe those properties.

10.3 Other Properties of State Machines

The kinds of properties we have seen so far are sometimes called *safety* properties, properties that say that "nothing bad happens." Another class of properties that people often discuss are called *liveness* properties, properties that say that "something good eventually happens." For a sequential system, computing the correct answer is an example of a safety property and termination of a program is an example of a liveness property (this program "eventually" terminates). For a concurrent system, deadlock freedom and mutual exclusion are examples of safety properties; for a distributed system, the property that a message sent is eventually received is an example of a liveness property.

For concurrent and distributed systems, there are many interesting liveness properties so we will defer discussing them until later in this book.

Chapter Notes

[TBD]

Further Reading

[TBD]

Exercises

[TBD]

Chapter 11

Relating State Machines: Equivalence

So far we have seen what a state machine is and how it can be used to model concepts like system behavior, input actions (or events) with arguments, output actions (or events) with results, and nondeterminism. We have even hinted at how a state machine would be an appropriate model of an abstract data type, and hence one of the bases for object-oriented programming.

We have also seen that given a state machine we can reason about some of its properties, most importantly, invariants.

In this and the next three chapters we consider the following question: "Given two state machines, how are they related?" This chapter discusses the relationship of *equivalence* between two state machines. Chapter 12 discusses the problem of whether one machine *satisfies* (in some sense) another. Chapter 13 takes a break from definitions and gives two examples of how to show one state machine satisfies another. Finally, in Chapter **??** we generalize these relations to show when one state machine simulates another.

We do not cover in this book the many different notions of equivalence or the many different ways of showing two state machines equivalent. These chapters are meant only to introduce the reader to these concepts and to provide enough detail for the reader to see what the fundamental questions are. Hence this chapter is short and to be read for edification.

11.1 Why Care About Equivalence?

Given two machines, M_1 and M_2, why would we care to know whether they are "equivalent"? The most compelling answer is that if we know M_2 is equivalent to M_1 then we know that we can *substitute* M_2 for M_1 without changing the overall system in which we do the substitution. This substitution principle is extremely important. Many areas in computer science rely on this principle. The most obvious example is in using a compiler. A compiler transforms a program into (we hope) a more efficient one. The source and target programs had better compute the same answer given the same inputs; in this sense the two programs are equivalent, and the target program is an efficient substitute for the source program. Similarly, software engineering relies on the notion of modularity where we can replace one component for another without affecting the rest of the system; functional programming, equational reasoning, and rewrite rule theory all rely on the principle of substituting equals for equals, sometimes called *referential transparency*. Even caching protocols in a multiprocessor or distributed system aim at keeping replicas equivalent (the "cache consistency" problem) so that the existence of multiple versions is hidden from the user.

Efficiency is one reason we might want to substitute one machine for another. M_2 may have fewer states or fewer state variables or fewer state transitions than M_1. In the real world, there are other good reasons: cost, user-friendliness, maintainability, portability, or just esthetics.

11.2 What Does Equivalence Mean?

The hard question is "When are two machines equivalent?" The answer is dependent on the context in which we intend to do the substitution. Intuitively, we strive for some notion of *observational equivalence* such that we as an external observer (the context) "cannot tell the difference" between whether M_1 is being used or M_2. If we cannot perceive a difference, then for all intents and purposes they are equivalent.

Answering this question is a subject of much debate and decades of research, mostly in the theoretical community on models of concurrent systems. Let's see what people might debate about.

At first, we might think it is easy to simply define equivalence in terms of input-output behavior. If we feed the two machines the same input do we get the same output? We ask this question for all possible inputs. If we always get

the same output for the same input, they are equivalent; if not, they are different. This notion of equivalence takes a pretty narrow view of what a state machine is. It essentially says that a state machine represents a function and determining equivalence between two state machines boils down to the problem of determining whether they compute the same function.

However, a software system is not nearly as simple. For example, software systems, and hence their state machine models, often have intermediate observable effects on the environment (their context). It does not suffice to just consider the final states of the machines. Thus, many prefer to take the broader view that two machines are equivalent if their behaviors (i.e., the sets of traces) are the same. To determine whether two sets of traces are the same we need a way to determine whether two traces are the same. To determine whether two traces are the same we need a way to determine whether two actions (or states) are the same. To determine whether two actions (or states) are the same we may need to ignore some actions or states variables (e.g., internal ones) and not others. So, it is not so easy to decide whether two state machines are equivalent; each substructure of a state machine introduces another place for differing opinions.

For example, depending on whether we take an event-based or state-based view of what a trace is we could come up with different answers given two machines. Consider the Light example with *off* and *on* states and the *flick* action. A different Light example with three states, e.g., red, amber, and green, but with the same action set $\{flick\}$ (think of a lever with three positions rather than two) will have the same behavior if we take an event-based view of traces, but different behavior if we take a state-based view.

Even for the same view of traces, there are different notions of equivalence. Suppose in determining whether two behaviors are the same, we need to determine whether two state-based traces are the same. Would we view a trace with "stuttering" states as equivalent to one without? Consider the Register example that has *read* and *write* actions where after doing two *read*'s in a row, we remain in the same state:

$$\langle x = 0, write(1)/ok(), x = 1, read()/ok(1), x = 1, read()/ok(1), x = 1, write(5)/ok(), x = 5\rangle$$

A state-based trace of this execution is:

$$\langle x = 0, x = 1, x = 1, x = 1, x = 5\rangle$$

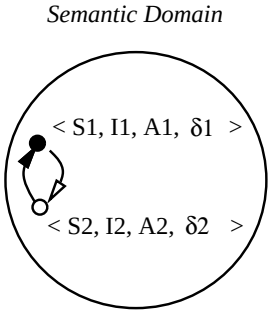
Is it equivalent to the following?

$\langle x = 0, x = 1, x = 5 \rangle$

It may be even harder to answer this question if our model includes infinite traces where a trace might end with an infinite number of stuttering states.

11.3 Showing Equivalence

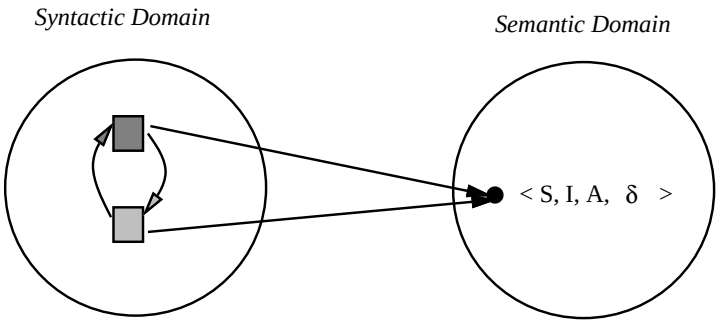
How do we show whether two machines are equivalent or not? There are two general approaches to take. First, we could work within just the semantic domain



Show one quadruple (semantic element)
is the same as the other.

and show that the two semantic entities, in this case quadruples, are equivalent. If our notion of equivalence is not defined directly in terms of quadruples, but rather behavior sets then our semantic domain would be behavior sets and we would have to show equivalence between two behavior sets, which might require showing equivalence between traces, etc..

Or, second, we could work within the syntactic domain



and show that the two syntactic descriptions, e.g., pre/post-condition specifications, Z specifications, or CSP programs, are equivalent in some sense. For example, for two different pre/post-condition specifications or two different Z specifications we might show that each of the predicates of one implies the corresponding predicate of the other and vice versa. For CSP programs, we might use properties of process algebras to show equivalence. Since the two syntactic descriptions are "the same," then they denote the same semantic entity, in this a case a quadruple representing a state machine; thus, they must describe the same state machine.

Another way to categorize how we might show the equivalence between two machines is as follows:

- Show equivalence from first principles using mathematical logic, theories of sets and sequences, induction, etc. Since equality on sets, sequences, integers, booleans, and other primitive types is mathematically well-defined, if we beat everything down to these primitive types, then we have a way of showing equivalence. We might use this technique if we want to show the equivalence of two quadruples.

- Show one "simulates" the other and vice versa. Anything we can do in one machine has some equivalent action or sequence of actions in the other. For example, if we want to show that one behavior set is the same as the other, then we might use this approach. We return to this idea in Chapter **??**.

- Transform one into the other. For example, if we want to show that one state machine description is "the same" as the other, we might use this approach. Showing that compiled code is "correct" amounts to showing that the transformed code is "the same" as the original code.

Further Reading

Exercises

Chapter 12

Relating State Machines: Satisfies

This chapter discusses the problem of whether one machine *satisfies* (in some sense) another. It should be read simultaneously (if that's possible!) with Chapter 13, which gives two examples.

Just as there is not one standard notion of equivalence, there is not one standard notion of "satisfies." In this chapter we give a definition that is reasonable, representative, and used in practice.

This chapter presents three key ideas:

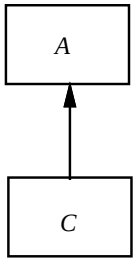
- A definition of *satisfies* in terms of a subset relationship between behavior sets of two machines.
- The notion of a mapping, called an *abstraction function*, that relates states of one machine to states of another.
- A proof technique that uses the abstraction function in a *commuting diagram* (see Section 12.3.1) to relate state transitions of one machine to state transitions of another.

Although our notion of *satisfies* and the proof technique that we present here may seem specific to the model of state machines described so far, what is the most important idea to learn from this chapter is the notion of an abstraction function. We will see it again, in a different guise, when we cover refinement in Z.

12.1 Why Care About "Satisfies"?

Suppose our favorite programming language does not support the notion of a set (i.e., *set* is not a built-in type) and we were asked to implement a set type in terms of the built-in types of the language, e.g., sequences (or arrays, more likely). This problem is a standard exercise in programming with data abstraction. For example, we might (1) represent sets in terms of sequences and then (2) implement each set operation in terms of operations on sequences and other built-in types. After solve this implementation problem, suppose we wanted *to prove* that our representation using sequences satisfies the abstract set type. How do we do it?

What we really care about is the relationship between a *concrete* state machine, *C*, and an *abstract* state machine, *A*:



where we usually read the arrow in one of many ways:

C satisfies A.
C implements A.
C is a refinement of A.
The program *C is correct with respect to* the specification *A*.

C and *A* can be interpreted in many ways:

C and *A* are both state machines.
C is a program and *A* is a specification.
C is an implementation and *A* is a specification.
C is a state machine and *A* is a predicate.
C and *A* are both programs.
C and *A* are both specifications.
C is an implementation and *A* is an interface specification.
C is a concrete data type and *A* is an abstract data type.
C is a C function definition and *A* is a C function declaration.
C is a C++ class definition (implementation) and *A* is a class declaration (interface).

C is a high-level design and A is a set of customer's requirements.
C is a low-level design and A is a high-level design.
C is an implementation and A is a low-level design.

12.2 What Does *Satisfies* Mean?

12.2.1 Binary Relations

Consider a simpler problem of determining when one binary relation *satisfies* another. Suppose we have a specification for a *square_root* procedure:

$$\begin{array}{l} \mathit{square\_root(x:\ int)/ok(int)} \\ \quad \textbf{pre}\ \exists i : \mathit{int} \bullet x = i * i \\ \quad \textbf{post}\ x = \mathit{result} * \mathit{result} \end{array}$$

that denotes the binary relation:

$$R_A = \{(4,2),(4,-2),(9,3),(9,-3),(16,4),(16,-4)\ldots\}$$

We could choose to implement this procedure such that we always return the positive square root of an integer. Its relation is:

$$R_C = \{(4,2),(9,3),(16,4)\ldots\}$$

Informally, the implementation *satisfies* the specification because R_C just narrows the possible choice of the integer returned allowed by R_A and the implementation defines some value for each input integer that is defined for the specification.

More formally, we have, given an abstract relation, R_A, and a concrete relation, R_C:

Definition 9. R_C **satisfies** R_A *iff:*

- $R_C \subseteq R_A$ *and*
- $dom R_C = dom R_A$.

The first property says that any pair in the concrete relation is also an element of the abstract relation. The second property says that for each input that is related by R_A, we want R_C to be defined. Without the second property, R_C could be empty and the *satisfies* relation would hold[1].

[1] A very "unsatisfying" relation!

12.2.2 State Machines

Consider the two state machines,

$$A = (S_A, I_A, A_A, \delta_A)$$
$$C = (S_C, I_C, A_C, \delta_C)$$

each denoting a behavior set, $Beh(C)$ and $Beh(A)$, respectively. We take an event-based view of trace: Each trace is a sequence of (invocation, response) pairs and each pair represents a single execution of one of the actions provided by the machine. We assume for simplicity that there is a one-to-one correspondence between the action names in the concrete machine to those in the abstract machine and that we use a renaming function, α, to define the relationship:

$$\alpha : A_C \rightarrow A_A$$

Using α we relate each state transition involving a C action to a state transition involving an A action.[2]

We define the *satisfies* relation as follows between two state machines as follows:

Definition 10. *C* **satisfies** *A iff Beh(C)* $\subseteq$ *Beh(A).*

Since $Beh(C)$ is a set of traces and $Beh(A)$ is a set of traces, the *satisfies* relation is satisfied if every trace in $Beh(C)$ is in $Beh(A)$. This means that A's set can be larger; C's set reduces the choices of possible acceptable behaviors.

Why does this definition of *satisfies* make sense? Viewing C as an implementation of a design specification A, this definition says that an implementor makes decisions that restrict the scope of the freedom *allowed by* a design. In other words, the specification is saying what *may* or *is permitted* to occur at the implementation level, not what *must* occur. The implementation narrows down the choice of what is allowed to happen. For example, an implementation might reduce the amount of nondeterminism allowed by the specification.

Thus, certainly having the behavior sets equal ($Beh(C) = Beh(A)$) is too strong. Having the subset relation go the other way ($Beh(A) \subseteq Beh(C)$) cannot be right either; otherwise there may be executions of the concrete machine that are not permitted by the abstract one.

[2] In all the examples, the renaming function will be obvious.

According to this definition of correctness the concrete machine with the empty behavior would be a perfectly acceptable implementation of an abstract machine. Since the empty behavior is not very satisfying, we normally assume that the set of initial states and the state transition function for the concrete machine are both non-empty; thus its behavior is non-empty.

The empty behavior case is the extreme case where the concrete machine does not do anything bad (a safety property)[3] since it does not do anything at all; however, the machine also does not do anything good (a liveness property). Our definition of *satisfies* does not require that our machine do anything; the definition requires only that the machine does only allowed things.

12.3 Showing One Machine Satisfies Another

How do we show the *satisfies* relationship holds between two state machines? Given our two state machines, $A = (S_A, I_A, A_A, \delta_A)$ and $C = (S_C, I_C, A_C, \delta_C)$, in general it is not so straightforward to determine if C *satisfies* A because their state sets, S_A and S_C, may be different. The proof technique we present uses an *abstraction function* to relate these state sets[4].

12.3.1 A Proof Technique

There are two major steps in the proof technique for showing one machine satisfies another.

1. The Creative (Intellectually Hard) Step.

   - Define an *abstraction function*

     $$AF : S_C \rightarrow S_A$$

     to relate concrete states with abstract states.

   - Define a *representation* (sometimes called *concrete*) invariant,

     $$RI : S_C \rightarrow bool$$

     that characterizes the domain of AF. This predicate prunes down the set of concrete states, S_C, to only those of interest (only those that

[3] See end of Handout on Reasoning About State Machines.

[4] Much like we use the renaming function α to relate different actions sets.

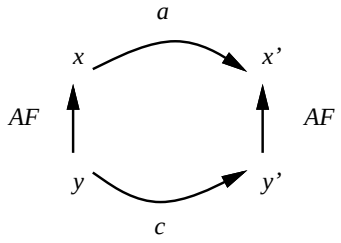
represent some abstract state in S_A). After we define this predicate, we must prove that it indeed is an invariant. We use the technique presented in Chapter 10 to show this.

2. The Checklist (Tedious) Step.

   - For each $i_C \in I_C$ (for all initial states in I_C) show:

     $$AF(i_C) \in I_A$$

     This requires showing that each initial state of the concrete machine is some initial state of the abstract machine.
   - For each state transition $(y, c, y') \in \delta_C$ of the concrete machine C where y satisfies RI, there must exist a state transition $(AF(y), \alpha(c), AF(y')) \in \delta_A$ of the abstract machine A. To show this, it suffices to show the following commutative relationship holds:

     where $x = AF(y), x' = AF(y')$ and $a = \alpha(c)$. Here's how to read the diagram: Put both index fingers at the concrete state y; move your right one to the right (performing the concrete action c) and then up (applying AF to the new concrete state y'); move your left one up (applying AF to y) and then to the right (performing the corresponding abstract action $a = \alpha(c)$; you should end up in the same place: the abstract state, x'.

To the reader: Please brand this *commuting diagram* on your brain.

12.3.2 Rationale

Why does this proof technique make sense? The technique should smell familiar. It is inductive in nature. There is a base case ("for each initial state ...") and an inductive step, defined in terms of all possible action instances ("for each state transitions ..."). As before, because the action sets are finite, we do a big case analysis, one action per case, in the inductive step.

- In the base step, notice there is no requirement that all initial states of the abstract machine be covered. So, there may be some abstract executions that have no corresponding concrete execution since we could not even get started. This is okay since we only need to show a subset relationship between behavior sets.

- The intuition behind the inductive step is that if a state transition can occur in the concrete machine then it must be *allowed to* occur in the abstract machine. If we show the inductive step for all state transitions in the concrete machine, then we will have shown it for all of its possible executions. (Notice that in showing the commuting diagram for each action of the concrete machine, we are really showing it for each state transition that involves that action.)

After showing the base case and inductive step, we will have shown that each trace in the behavior set of the concrete machine has a corresponding (modulo the abstraction function) trace in the behavior set of the abstract machine. Hence, the behavior set of the concrete machine is a subset of the behavior set of the abstract machine (modulo the abstraction function), which is what we needed to prove.

Aside: In our proof technique, because of our simplistic way of associating actions in one machine to another (through the one-to-one function α) we are associating a single state transition in C to a single state transition in A. More generally, a state transition in C might map to a *sequence* of transitions in the abstract machine A. This generalization is especially needed for proving concurrent state machines correct and/or dealing with state machines with internal as well as external actions. We will return to this generalization in Chapter **??**.

12.3.3 Abstraction Functions and Representation Invariants

An abstraction function maps a state of the concrete machine to a state of the abstract machine. It explains how to interpret each state of the concrete machine as a state of the abstract machine. It solves the problem of the concrete and abstract machines having different sets of states.

Since the abstraction function is a function, we rely on the substitution property: If AF is a function and we know $x = y$ then we know $AF(x) = AF(y)$. If AF were a more general kind of relation, this property would not necessarily hold.

We might think that the abstraction function should map the other way, explaining how to represent each state of the abstract machine. This is usually not a

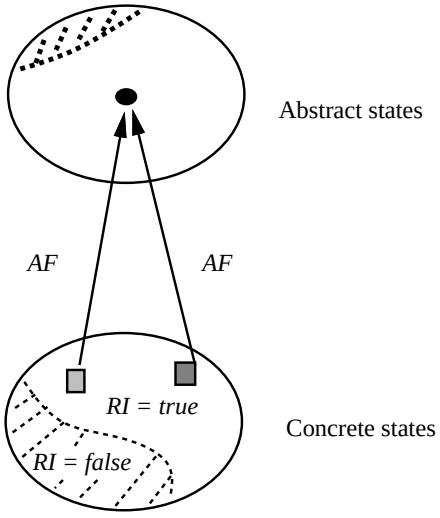
good idea, in large part because there is often more than one way of representing each state of the abstract machine. For example, suppose we represent a set by a sequence. Then many *different* sequences, e.g., $\langle 3,5,-1\rangle$, $\langle 5,3,-1\rangle$, $\langle -1,5,3\rangle$, $\langle 3,5,5,-1\rangle$, could (given the appropriate definition of *AF* and *RI*) all represent the *same* set $\{3,5,-1\}$.

In other words, *AF*, in general, is many-to-one.

AF may be partial. Not all states of the concrete machine may represent a "legal" abstract state. For example, in the integer-modulo7 example, integers not within 0..6 are not "legal" representations of days of the week. The representation invariant serves to restrict the domain of the abstraction function. We may assume that for any concrete state for which the representation invariant does not hold, *AF* is undefined.

Finally, *AF* is not necessarily onto. There may be states of the abstract machine that are not represented by any state of the concrete machine. This can be true of initial states as well. In the context of showing that one machine adequately implements another, this may sound strange; we say more on *adequacy* in Section 12.3.4.

Putting everything together, we have the final diagram:



where the bottom *unshaded* region represents the domain of *AF* and the top *unshaded* region represents its range.

12.3.4 Variations on a Theme

Adequacy

In this handout we explicitly stated that AF need not be surjective (onto). Were we to require AF to be surjective, then we would require that every abstract state have some concrete representation, i.e., that AF is *adequate*. Requiring AF to be adequate makes very good sense since we might like to know that every abstract state we have modeled is implemented by some concrete state. Some refinement methods like VDM require that AF be adequate. And, in proving the correctnesss of an abstract data type, we usually require AF to be adequate for the concrete representation type.

As mentioned, we also do not require adequacy in the sense of having every state transition of A be implemented in terms of one in C. Rather, we only require that every state transition in C relate to some state transition of A. (C cannot do anything not permitted by A.) This laxity is in contrast to how we defined whether one binary relation, R_C, *satisfies* another, R_A; we required that for each input related by R_A, R_C should be defined. This requirement gets at the *adequacy* of R_C viewed as an implementation of R_A.

Taking this last point to an extreme, we do not even require that every action in A actually be "implemented" by some action (or sequence of actions) in C. In other words there may be state transitions, all associated with a particular action of A, that have no correspondence are not adequately represented) in C.

We will see later in Part III when we cover Z and CSP that other state machine-based models impose different kinds of adequacy restrictions in defining a *refinement/correctness* relation between two machines.

Abstraction Relations

Some people prefer to use abstraction *relations* or abstraction *mappings* more generally than functions. There are examples where it is easier, more convenient, or more natural to map a concrete state to a set of abstract states. As mentioned, however, we would lose the substitution property of abstraction functions.

Auxiliary Variables

Sometimes it is not so straightforward to prove a concrete machine satisfies an abstract machine in terms of just the state variables of the concrete machine. In this case, we need to introduce *auxiliary variables* (sometimes called *dummy variables* or *history variables* in the literature). The need for auxiliary variables in such proofs is especially common for reasoning about concurrent programs.

Further Reading

Exercises

Chapter 13

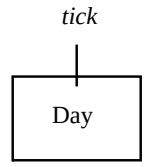
Relating State Machines: Two Examples

13.1 Days

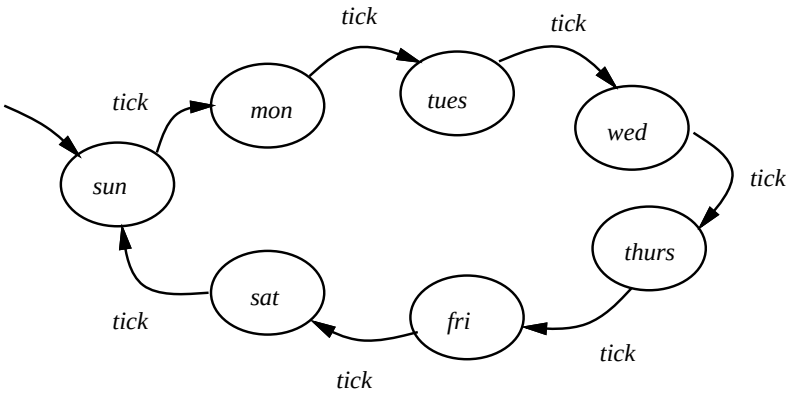
Here is a simple example to show how an "integer mod-7" counter state machine satisfies a "days of the week" machine. Both the abstract and concrete machines have a finite number of states, a finite number of transitions, and infinite traces. The proof of correctness uses an abstraction function that is one-to-one.

13.1.1 The Abstract Machine

The Day abstract machine, has one *tick* action.



Day's Interface



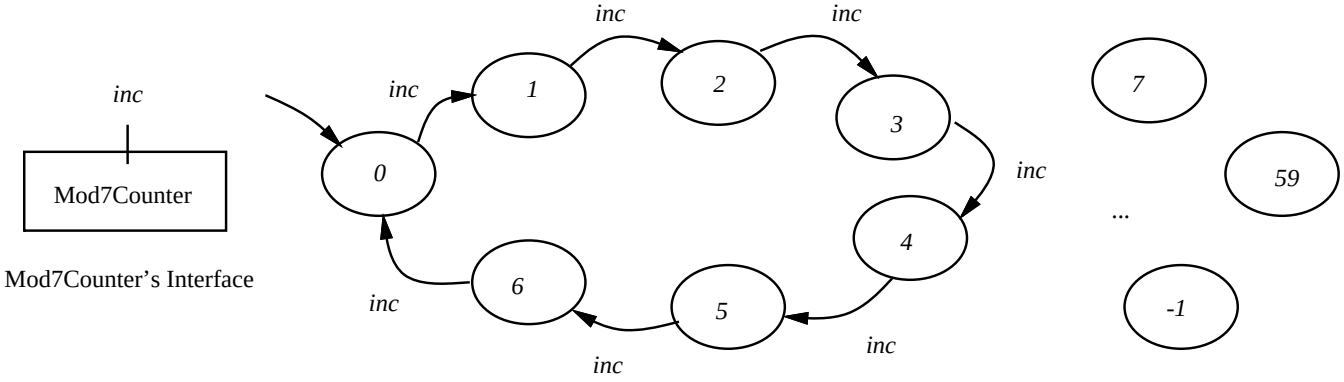
Day's State Transition Diagram

Day is like an enumerated type in Pascal; its set of states is just the days of the week:

$$
\begin{array}{l}
\text{Day} = (\\
\{sun, mon, tues, wed, thurs, fri, sat\}, \\
\{sun\}, \\
\{tick\}, \\
\delta_A = \{(sun, tick, mon), (mon, tick, tues), (tues, tick, wed), (wed, tick, thurs), \\
\ (thurs, tick, fri), (fri, tick, sat), (sat, tick, sun)\} \\
).
\end{array}
$$

13.1.2 The Concrete Machine

Most programming languages do not support something like Day as a built-in type. But fortunately, they usually support integers. So let's represent the Day machine in terms of integers modulo-7.



Mod7Counter's State Transition Diagram

$$
\begin{array}{l}
\text{Mod7Counter} = (\\
\{x : int\}, \\
\{0\}, \\
\{inc\}, \\
\delta_C = \{(0, inc, 1), (1, inc, 2), (2, inc, 3), (3, inc, 4), (4, inc, 5), \\
\ (5, inc, 6), (6, inc, 0)\} \\
).
\end{array}
$$

The Mod7Counter concrete machine has a single action, *inc*, which is defined in the obvious way. Let's intentionally defined Mod7Counter's set of states to be

the set of integers rather than just the integers between 0 and 6, inclusive. (What are the reachable states of Mod7Counter?)

13.1.3 Proof of Correctness

It should be pretty obvious that Mod7Counter machine satisfies the Day machine. But, let's go through the steps.

1. Step 1: Define an abstraction function and representation invariant.

   - First, we define an abstraction function.

     $$AF : int \nrightarrow \{sun, mon, tues, wed, thurs, fri, sat\}$$
     $$AF(0) = sun$$
     $$AF(1) = mon$$
     $$\ldots$$
     $$AF(6) = sat$$

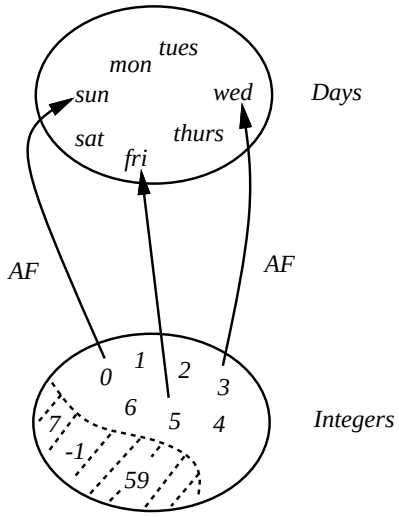
     Notice that it is one-to-one.

   - Next, we define the representation invariant. We do not need all integers to represent the days of the week. We only need seven. Moreover, the transition function, δ_C is defined for only those seven.

     $$RI : int \rightarrow bool$$
     $$RI(i) = 0 \leq i \leq 6$$

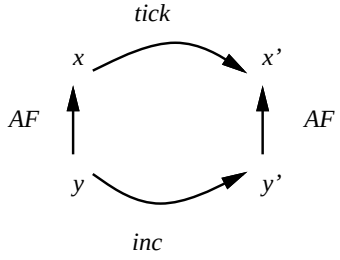
     The representation invariant just says that the only integer values we have to worry about are between 0 and 6 inclusive. It characterizes the domain of AF.

   The picture relating concrete to abstract states looks like:



2. Step 2: Initial conditions and commuting diagram for each action.

   - Initial conditions: We need to show that each initial state of Mod7Counter is an initial state of Day (under the abstraction function).
     - 0 is the initial state for the Mod7Counter. Thus, $AF(0)$ had better be some initial state of Day. Indeed, $AF(0) = sun$, the initial state of Day.
   - Commuting diagram: We need to show it holds for each action of Mod7Counter.



$$\delta_A(AF(y), tick) = AF(\delta_C(y, inc))$$

In other words, we need to show[1] that $\delta_A(AF(y), tick) = AF(\delta_C(y, inc))$ for all y that satisfy the representation invariant, i.e., for all $y \in \{i : int \mid$

[1]Note that the use of functional notation for the state transition relations δ_A and δ_C; more importantly, because they are both functions, it is sound to use equational reasoning in the proof.

$0 \leq i \leq 6\}$. We need to show the commuting diagram for only those states that satisfy *RI*. The simplest proof is to do an exhaustive case analysis. y can take on only seven values so there are seven cases. Let's do the most "interesting" case ($y = 6$).

- Case: $y = 6$

$$\begin{array}{ll} \delta_A(AF(6), tick) & \\ = \delta_A(sat, tick) & \text{def'n of } AF \\ = sun & \text{def'n of } \delta_A \\ = AF(0) & \text{def'n of } AF \\ = AF(\delta_C(6, inc)) & \text{def'n of } \delta_C \end{array}$$

In the last part of the proof above, we did not really do it as shown, but rather we reduced both sides of the equation at the same time yielding $sun = sun$. we can do that because equality is bi-directional:

$$\begin{array}{lll} \delta_A(AF(6), tick) = AF(\delta_C(6, inc)) & & \\ \delta_A(sat, tick) & = AF(0) & \text{def'ns of } AF \text{ and } \delta_C \\ sun & = sun & \text{def'ns of } \delta_A \text{ and } AF \end{array}$$

The above proof is more readable and it is perfectly acceptable. Just remember that we need to give our justification next to each proof step if it is not obvious or clear from context.

13.2 Sets

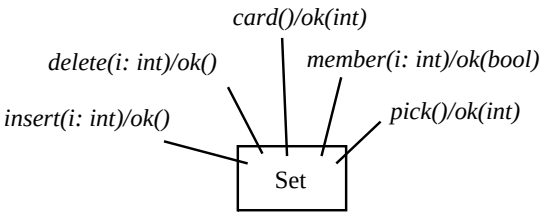
The motivation for this example is show that when we do "object-oriented" programming, we are really identifying certain abstract objects (better known an *data abstractions* or *abstract data types*) like sets, stacks, queues, symbol tables, etc.. We eventually have to realize (i.e., implement) these objects in a real programming language in terms of either other abstract objects or the language's built-in data objects like sequences, arrays, records, linked lists, etc. After we write our (concrete) implementation we are then faced with proving it correct with respect to the (abstract) specification.

Not surprisingly, these data objects (abstract or built-in) can themselves be viewed as little state machines. So, to show the correctness of an implementation of an abstract object is very much like showing that one state machine (the concrete one) satisfies another (the abstract one).

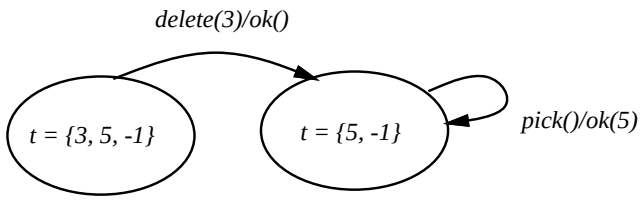
There are other proof techniques used to prove the correctness of the implementations of abstract data types.

13.2.1 Abstract Machine: Set

The Set abstract machine has the following interface:



Set's Interface



Part of Set's State Transition Diagram

Set = (
$\{s : \{t\} \rightarrow set[int]\}$,
$\{s : \{t\} \rightarrow set[int] \mid s(t) = \emptyset\}$,
$\{insert(i : int)/ok(), \ldots$ see above $\ldots, pick()/ok(int)\}$,
$\delta_A = \ldots$ see next page $\ldots$
).

Here are specifications of the actions, *insert, delete, card, member, and pick*.

insert(i: int)/ok()
pre *true*
post $t' = t \cup \{i\}$

delete(i: int)/ok()
pre *true*
post $t' = t \setminus \{i\}$

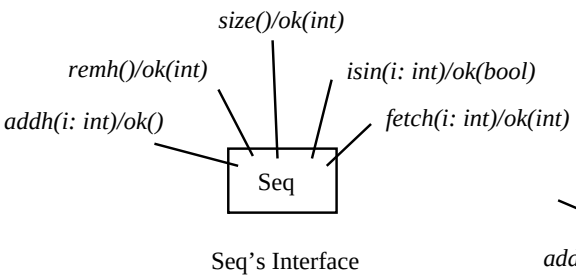
card()/ok(int)
pre *true*
post $t' = t \wedge result = \#t$

member(i: int)/ok(bool)
pre *true*
post $t' = t \wedge result = (i \in t)$

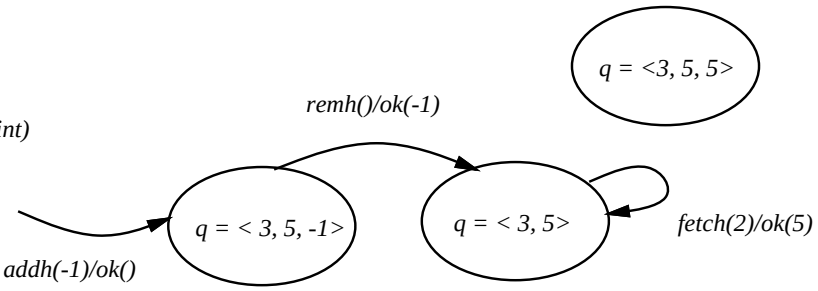
pick()/ok(int)
pre $t \neq \emptyset$
post $t' = t \wedge result \in t$

13.2.2 Concrete Machine: Seq

Suppose we decide to implement the Set machine in terms of a Seq machine with the following interface:



Seq's Interface



Part of Seq's State Transition Diagram

We define the actions such that the state where $q = \langle 3,5,5 \rangle$ is unreachable. We will see why soon.

Seq = (

$\{s : \{q\} \rightarrow seq[int]\}$,
$\{s : \{q\} \rightarrow seq[int] \mid s(q) = \langle\rangle\}$,
$\{addh(i : int)/ok(), \ldots$ see above $\ldots, fetch(i : int)/ok(int)\}$,
$\delta_C = \ldots$ see next page $\ldots$
).

Seq's actions have the following specification:

addh(i: int)/ok()
pre $i \notin ran\, q$
post $q' = q \frown \langle i \rangle$

remh()/ok(int)
pre $q \neq \langle\rangle$
post $q = q' \frown \langle result \rangle$

size()/ok(int)
pre *true*
post $q' = q \wedge result = \#q$

isin(i: int)/ok(bool)
pre *true*
post $q' = q \wedge result = (i \in ran\, q)$

fetch(i: int)/ok(int)
pre $q \neq \langle\rangle \wedge i \in dom\, q$
post $q' = q \wedge result = q\, i$

13.2.3 Proof of Correctness

Formally, *RI* and *AF* are defined over Seq's set of states, but it is going to be notationally convenient (and more understandable) if we denote each of these states by the sequence value to which Seq's state variable, q, maps. In other words, we should write something like:

$$RI(s(q) = \langle 3, 5, -1 \rangle) = \ldots$$

but instead we write:

$$RI(\langle 3,5,-1\rangle) = \ldots$$

Step 1

We first need to define an abstraction function and a representation invariant.

1. Abstraction Function.

Informally AF takes an (ordered) sequence of elements and turns it into an (unordered) set of the sequence's elements. Formally,

$$AF : seq[int] \nrightarrow set[int]$$
$$AF(\langle\rangle) = \emptyset$$
$$AF(q \frown \langle e\rangle) = AF(q) \cup \{e\}$$

It is common for abstraction functions to be defined recursively like this.

Notice that this AF is many-to-one. There are many sequence values that map to the same set value because we do not care what the order of elements is in a set. In fact, the orderedness property of sequences is exactly the "irrelevant" property from which we abstract. For example,

$$AF(\langle 3,5,-1\rangle) = \{3,5,-1\}$$
$$AF(\langle 5,3,-1\rangle) = \{3,5,-1\}$$
$$AF(\langle -1,5,3\rangle) = \{3,5,-1\}$$

These three different sequence values map to the same set value.

2. Representation Invariant.

Notice that the *addh* action has a pre-condition that checks whether the element to be inserted is already in the sequence. Thus, only sequence values that have no duplicate elements serve to represent set values. We have the following representation invariant, which characterizes the domain of AF:

$$RI : seq[int] \rightarrow bool$$
$$RI(q) = \forall\, 1 \leq i,j \leq \#q \bullet i \neq j \Rightarrow q\,i \neq q\,j$$

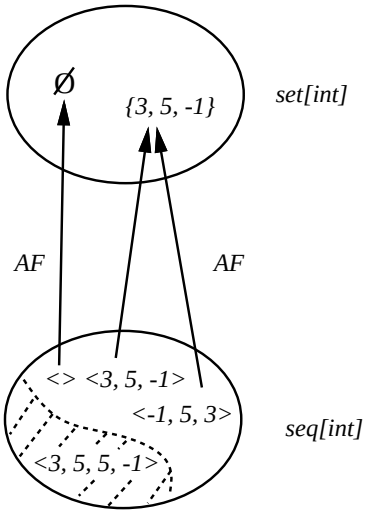
Informally we call this representation invariant, NoDups. Thus we have,

$RI(\langle 3,5,5,-1\rangle)$	$= \mathit{false}$	5 appears twice.
$RI(\langle 3,5,-1\rangle)$	$= \mathit{true}$	NoDups
$RI(\langle\rangle)$	$= \mathit{true}$	The empty sequence is ok.

All those sequence values that *RI* maps to *true* are legal representations of set values.

IMPORTANT: Remember that there is a side proof that we need to do here. We need to show that the representation invariant is indeed an invariant. That is, along the lines of the proof technique described in Chapter 10, we need to show that the invariant is established in the initial states and preserved by each action. We leave this part of the proof as an exercise to the reader.

Here is a picture illustrating that *AF* is partial and many-to-one:



Step 2

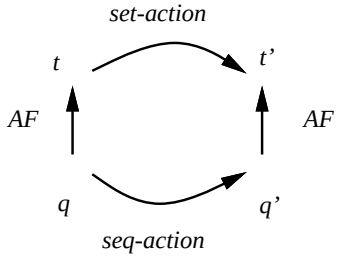
Armed with *RI* and *AF*, we now show the concrete machine satisfies the abstract one.

1. Initial condition.

We need to show that each initial state of Seq maps to some initial state of Set. More formally, we need to show that $AF(\langle\rangle) = \emptyset$. This is obviously true by the definition of *AF*.

2. Commuting diagram for each Seq action.

We need to show this diagram:



δ_A (AF(q), set-action) = AF(δ_C (q, seq-action))

There are five cases, one for each Seq action. Let's just do three actions, *addh*, *remh*, and *size*.

Case: *addh* satisfies *insert*.

We first consider the case where i is not in the sequence and then the case for when it is.

Case 1: $i \notin ran\, q$

According to the post-condition for Set's *insert* action we need to show that the set value for t' obtained after doing an *insert* with argument i is $t \cup \{i\}$. Seq's *addh* action has the effect of adding to the high end of the sequence only if its argument i is not already stored in the sequence. Thus if q is the value of the sequence before, then $q \frown \langle i \rangle$ is the value after. In other words, we have:

$$
\begin{array}{lll}
 & AF(\delta_C(q, addh(i)/ok())) & \\
. & = AF(q) \frown \langle i \rangle) & \text{post-condition of } addh \\
 & = AF(q) \cup \{i\} & \text{def'n of AF} \\
 & = t \cup \{i\} & \text{since } t = AF(q) \\
 & = t' & \text{post-condition of } insert
\end{array}
$$

Case 2: $i \in ran\, q$

If i is already in the sequence then no state transition occurs and q stays the same.

Case: *remh* satisfies *delete*.

According to the post-condition of Set's *delete* action the set value for t' obtained after doing the *delete* is the set with i removed. Seq's *remh* action has the effect of removing and returning the high end of the original sequence q. Informally speaking, it is this element *result*, which we would "pass to" *delete* as an argument. In other words, given a particular state transition involving *remh*, we

choose a particular state transition involving *delete*—the one for which *result* is passed as an argument. We have:

$$
\begin{array}{ll}
\delta_A(AF(q), delete(result)/ok()) & \\
= \delta_A(AF(q' \frown \langle result \rangle), delete(result)/ok()) & \text{post-condition of } remh \\
= \delta_A(AF(q') \cup \{result\}, delete(result)/ok()) & \text{def'n of } AF \\
= (AF(q') \cup \{result\}) \setminus \{result\} & \text{post-condition of } delete \text{ (and def'n of } \delta \\
= AF(q') & \text{properties about set union and set differ} \\
= t' & t' = AF(q')
\end{array}
$$

Case: *size* satisfies *card*

Looking at the post-condition for Set's *card* action, there are two things to show.

First, we need to show that the value returned by the Seq's *size* action is the size of the corresponding set (under AF). Because of NoDups (the RI), we know that the size of the sequence representing a set is the size of the set it represents. More formally, we would need to prove a lemma like:

Lemma 2. $\forall q : seq[int] \bullet (\#q = \#AF(q))$

Second, we need to show that *size* does not change the abstract value of the set that q represents. More formally,

$$
\begin{array}{ll}
q' = q & \\
\Rightarrow & \text{post-condition of } size \\
AF(q') = AF(q) & \\
\Rightarrow & AF \text{ is a function.} \\
t' = t &
\end{array}
$$

IMPORTANT: Notice that we rely on the abstraction function AF on being a function here. In the second step above, we apply AF to two equal things; since AF is a function (and not a relation), we know the result of applying AF to two equal things will result in two equal things.

For the homework exercises, it is fine to give informal proofs like the ones given here.

Further Reading

Exercises

Bibliography

[1] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

[2] B. Cohen, W. T. Harwood, and M. I. Jackson. *The Specification of Complex Systems*. Addison-Wesley, 1986.

[3] Gerhard Gentzen. Untersuchungen über das logische Schliessen. *Mathematische Zeitschrift*, 39(1):176–210; 405–431, 1935.

[4] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, i. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[5] J. K. Rowling. *Harry Potter and the Sorcerer's Stone*. Scholastic Press, 1998.

[6] Jim Woodcock and Martin Loomes. *Software Engineering Mathematics*. The SEI Series in Software Engineering. Addison-Wesley, 1998.