Formal Reasoning About Programs

Adam Chlipala

MIT, Cambridge, MA, USA
Email address: adamc@csail.mit.edu

Abstract. *Briefly*, this book is about an approach to bringing software engineering up to speed with more traditional engineering disciplines, providing a mathematical foundation for rigorous analysis of realistic computer systems. As civil engineers apply their mathematical canon to reach high certainty that bridges will not fall down, the software engineer should apply a different canon to argue that programs behave properly. As other engineering disciplines have their computer-aided-design tools, computer science has proof assistants, IDEs for logical arguments. We will learn how to apply these tools to certify that programs behave as expected.

More specifically: Introductions to two intertangled subjects: the Coq proof assistant, a tool for machine-checked mathematical theorem proving; and formal logical reasoning about the correctness of programs.

For more information, see the book's home page:
http://adam.chlipala.net/frap/

Contents

CHAPTER 1

Why Prove the Correctness of Programs?

The classic engineering disciplines all have their standard mathematical techniques that are applied to the design of any artifact, before it is deployed, to gain confidence about its safety, suitability for some purpose, and so on. The engineers in a discipline more or less agree on what are "the rules" to be followed in vetting a design. Those rules are specified with a high degree of rigor, so that it isn't a matter of opinion whether a design is safe. Why doesn't software engineering have a corresponding agreed-upon standard, whereby programmers convince themselves that their systems are safe, secure, and correct? The concepts and tools may not quite be ready yet for broad adoption, but they have been under development for decades. This book introduces one particular tool and a body of ideas for how to apply it to different tasks in program proof.

As this document is in a very early draft stage, no more will be said here, in favor of jumping right into the technical material. Eventually, there will no doubt be some sort of historical overview here, as part of a general placing-in-context of the particular approach that will come next. There will also be plenty of scholarly citations (here and throughout the book). In this early version, you get to take the author's word for it that we are about to learn a promising approach!

However, one overarching element of our strategy is important enough to deserve to be called out here. We will study a variety of different approaches for formalizing what a program should do and for proving that a program does what it should. At every step, we will pay close attention to the *common foundation* that underlies everything. For one thing, we will be proving all of our theorems with the Coq proof assistant, a powerful framework for writing and machine-checking proofs. Coq itself is based on a relatively small set of core features, much like a well-designed programming language, and in both we build up increasingly sophisticated abstractions as libraries. Those features can be thought of as the core of all mathematical reasoning.

We will also apply a recipe specific to program proof. When we encounter a new challenge, to prove a new kind of property about a new kind of program, we will generally be considering four broad elements that appear in nearly all techniques.

- **Encoding.** Every programming language has both *syntax*, which defines what programs look like, and *semantics*, which defines how programs behave when run. Even when these elements seem obvious intuitively, we often find that there are surprisingly subtle choices to be made in defining syntax and semantics at the highest level of rigor. Seemingly minor decisions can have big impacts on how smoothly our proofs go.
- **Invariants.** Nearly every theorem about a program is stated in terms of a *transition system*, with some set of states and a relation for stepping

from one state to the next, moving forward in time. Nearly every program proof also works by finding an *invariant* of a transition system, or a property that always holds of every state reachable from some starting state. The concept of invariant is very close to being a direct reinterpretation of mathematical induction, that glue of every serious mathematical development, known and loved by all.

- **Abstraction.** Often a transition system is too complex to analyze directly. Instead, we *abstract* it with another transition system that is somehow more tractable, proving that the new system preserves all relevant properties of the original.
- **Modularity.** Similarly, when a transition system is too complex, we often break it into separate *modules* and use some well-behaved composition operators to reassemble them into the whole. Often abstraction and modularity go together, as we decompose a system both *horizontally* (i.e., with modularity), splitting it into more manageable parts, and *vertically* (i.e., with abstraction), simplifying parts in ways that preserve key properties. We can even alternate between strategies, breaking a system into parts, abstracting one as a simpler part, further decomposing that part into pieces, and so on.

In the course of the book, we will never quite define any of these meta-techniques in complete formality. Instead, we'll meet many examples of each, called out by eye-catching margin notes. Generalizing from the examples should help the reader start developing an intuition for when to use each element and for the common design patterns that apply.

The core subject matter of the book is often grouped under traditional disciplinary headers like *semantics*, *programming-languages theory*, *formal methods*, and *verification*. Often these different traditions have their own competing terminology for shared concepts. We'll follow one particular set of unified terminology and notation, cherry-picked from the conventions of different communities. There really is a huge amount of commonality across everything that we'll study, so we don't want to distract by constantly translating between notations. It is quite important to be literate in the standard notational conventions, which are almost always implemented with LaTeX, and we stick entirely to that kind of notation in this book. However, we follow another, much less usual convention: while we give theorem and lemma statements, we rarely give their proofs. The reason is that the author and many other researchers today feel that proofs on paper have outlived their usefulness. Instead, the proofs are all found in the parallel world of the accompanying Coq source code.

That is, each chapter of this book has a corresponding Coq source file, distributed with the general book source code. The Coq sources are heavily commented and may even, in many cases, be feasible to read without also reading the book chapters. More importantly, the Coq sources aren't just meant to be *read*. They are meant to be *executed*. We suggest stepping through them interactively, seeing intermediate states of proofs as appropriate. The book proper can be read without the Coq sources, to learn the standard background material of program proof; and the Coq sources can be read without the book proper, to learn a particular concrete realization of those ideas. However, they go better together.

CHAPTER 2

Formalizing Program Syntax

2.1. Concrete Syntax

The definition of a program starts with the definition of a programming language, and the definition of a programming language starts with its *syntax*, which covers which sorts of phrases are basically well-formed. In the next chapter, we turn to *semantics*, which, in the course of saying what programs *mean*, may impose further validity conditions. Turning to examples, let's start with *concrete syntax*, which decrees which sequences of characters are acceptable. For a simple language of arithmetic expressions, we might accept the following strings as valid.

$$3$$
$$x$$
$$3 + x$$
$$y * (3 + x)$$

Plenty of other strings might be invalid, like these.

$$1 + + 2$$
$$x\ y\ z$$

Rather than appeal to our intuition about grade-school arithmetic, we prefer to formalize concrete syntax with a *grammar*, following a style known as *Backus-Naur Form (BNF)*. We have a set of *nonterminals* (e.g., e below), standing for sets of allowable strings. Some are defined by appeal to existing sets, as below, when we define constants n in terms of the well-known set $\mathbb{N}$ of natural numbers (nonnegative integers).

Encoding

$$\begin{array}{rrcl} \text{Constants} & n & \in & \mathbb{N} \\ \text{Variables} & x & \in & \mathsf{Strings} \\ \text{Expressions} & e & ::= & n \mid x \mid e + e \mid e \times e \end{array}$$

To interpret the grammar in plain English: we assume sets of constants and variables, based on well-known sets of natural numbers and strings, respectively. We then define expressions to include constants, variables, addition, and multiplication. Crucially, the last two cases are specified *recursively*: we show how to build bigger expressions out of smaller ones.

Incidentally, we're already seeing how many different formal notations creep into the discussion of formal program proofs. All of this content is typeset in LaTeX, and it may be helpful to consult the book sources, to see how it's all done.

Throughout the subject, one of our most crucial tools will be *inductive definitions*, explaining how to build up bigger sets from smaller ones. The recursive nature of the grammar above is implicitly giving an inductive definition. A more general notation for inductive definitions provides a series of *inference rules* that

define a set. Formally, the set is defined to be *the smallest one that satisfies all the rules*. Each rule has *premises* and a *conclusion*. We illustrate with four rules that together are equivalent to the BNF grammar above, for defining a set Exp of expressions.

Encoding

$$\frac{n \in \mathbb{N}}{n \in \mathsf{Exp}} \quad \frac{x \in \mathsf{Strings}}{x \in \mathsf{Exp}} \quad \frac{e_1 \in \mathsf{Exp} \quad e_2 \in \mathsf{Exp}}{e_1 + e_2 \in \mathsf{Exp}} \quad \frac{e_1 \in \mathsf{Exp} \quad e_2 \in \mathsf{Exp}}{e_1 \times e_2 \in \mathsf{Exp}}$$

The general reading of an inference rule is: **if** all the facts above the horizontal line are true, **then** the fact below the line is true, too. The rule implicitly needs to hold for *all* values of the *metavariables* (like n and e_1) that appear within it; we can model them more explicitly with a sort of top-level universal quantification. Newcomers to semantics often react negatively to seeing this style of definition, but very quickly it becomes apparent as a remarkably compact notation for expressing many concepts. Think of it as a domain-specific programming language for mathematical definitions, an analogy that becomes quite concrete in the associated Coq code!

2.2. Abstract Syntax

After that brief interlude with concrete syntax, we now drop all formal treatment of it, for the rest of the book! Instead, we concern ourselves with *abstract syntax*, the real heart of language definitions. Now programs are *abstract syntax trees* (*ASTs*), corresponding to inductive type definitions in Coq or algebraic datatype definitions in Haskell. Such types can be defined by enumerating their *constructor* functions with types.

Encoding

$$\begin{aligned} \mathsf{Const} &: \mathbb{N} \to \mathsf{Exp} \\ \mathsf{Var} &: \mathsf{Strings} \to \mathsf{Exp} \\ \mathsf{Plus} &: \mathsf{Exp} \times \mathsf{Exp} \to \mathsf{Exp} \\ \mathsf{Times} &: \mathsf{Exp} \times \mathsf{Exp} \to \mathsf{Exp} \end{aligned}$$

Note that the "×" here is not the multiplication operator of concrete syntax, but rather the Cartesian-product operator of set theory, to indicate a type of pairs!

Such a list of constructors defines the set Exp to contain exactly those terms that can be built up with the constructors. In inference-rule notation:

Encoding

$$\frac{n \in \mathbb{N}}{\mathsf{Const}(n) \in \mathsf{Exp}} \quad \frac{x \in \mathsf{Strings}}{\mathsf{Var}(x) \in \mathsf{Exp}} \quad \frac{e_1 \in \mathsf{Exp} \quad e_2 \in \mathsf{Exp}}{\mathsf{Plus}(e_1, e_2) \in \mathsf{Exp}} \quad \frac{e_1 \in \mathsf{Exp} \quad e_2 \in \mathsf{Exp}}{\mathsf{Times}(e_1, e_2) \in \mathsf{Exp}}$$

Actually, semanticists get tired of writing such verbose descriptions, so proofs on paper tend to use exactly the sort of notation that we associated with concrete syntax. The trick is mental desugaring of the concrete-syntax notation into abstract syntax! We will generally not dwell on the particularities of that process. Instead, we repeatedly illustrate it by example, using Coq code that starts with abstract syntax, accompanied by LaTeX-based "code" in this book that applies concrete syntax freely.

Abstract syntax is handy for writing *recursive definitions* of functions. Here is one in the clausal style of Haskell.

$$\begin{aligned}
\mathsf{size}(\mathsf{Const}(n)) &= 1 \\
\mathsf{size}(\mathsf{Var}(x)) &= 1 \\
\mathsf{size}(\mathsf{Plus}(e_1, e_2)) &= 1 + \mathsf{size}(e_1) + \mathsf{size}(e_2) \\
\mathsf{size}(\mathsf{Times}(e_1, e_2)) &= 1 + \mathsf{size}(e_1) + \mathsf{size}(e_2)
\end{aligned}$$

It is important that we include *one clause per constructor of the inductive type*. Otherwise, the function would not be *total*. We also need to be careful to ensure *termination*, by making recursive calls only on the arguments of the constructors. This termination criterion, adopted by Coq, is called *primitive recursion*.

It is also common to associate a recursive definition with a new notation. For example, we might prefer to write $|e|$ for $\mathsf{size}(e)$, as follows.

$$\begin{aligned}
|\mathsf{Const}(n)| &= 1 \\
|\mathsf{Var}(x)| &= 1 \\
|\mathsf{Plus}(e_1, e_2)| &= 1 + |e_1| + |e_2| \\
|\mathsf{Times}(e_1, e_2)| &= 1 + |e_1| + |e_2|
\end{aligned}$$

Let's continue to exercise our creative license and write $\lceil e \rceil$ for the *depth* of e, that is, the length of the longest downward path from the syntax-tree root to any leaf.

$$\begin{aligned}
\lceil \mathsf{Const}(n) \rceil &= 1 \\
\lceil \mathsf{Var}(x) \rceil &= 1 \\
\lceil \mathsf{Plus}(e_1, e_2) \rceil &= 1 + \max(\lceil e_1 \rceil, \lceil e_2 \rceil) \\
\lceil \mathsf{Times}(e_1, e_2) \rceil &= 1 + \max(\lceil e_1 \rceil, \lceil e_2 \rceil)
\end{aligned}$$

2.3. Structural Induction Principles

The main reason to prefer abstract syntax is that, while strings of text *seem* natural and simple to our human brains, they are really a lot of trouble to treat in complete formality. Inductive trees are much nicer to manipulate. Considering the name, it's probably not surprising that the main thing we want to do on them is *induction*, an activity most familiar in the form of *mathematical induction* over the natural numbers. In this book, we will not dwell on many proofs about natural numbers, instead presenting the more general and powerful idea of *structural induction* that subsumes mathematical induction in a formal sense, based on viewing the natural numbers as one simple inductively defined set.

There is a general recipe to go from an inductive definition to its associated induction principle. When we define set S inductively, we gain an induction principle for proving that some predicate P holds for all elements of S. To make this conclusion, we must discharge one proof obligation per rule of the inductive definition. Recall our last rule-based definition above, for the abstract syntax of Exp. To derive an Exp structural induction principle, we produce a new set of rules, cloning each rule with two key modifications:

(1) Replace each conclusion, of the form $E \in S$, with a conclusion $P(E)$. That is, the obligations involve *showing* that P holds of certain terms.

(2) For each premise $E \in S$, add a companion premise $P(E)$. That is, the obligation allows *assuming* that P holds of certain terms. Each such assumption is called an *inductive hypothesis* (*IH*).

That mechanical procedure derives the following four proof obligations, associated with an inductive proof that $\forall x \in \mathsf{Exp}.\ P(x)$.

$$\frac{n \in \mathbb{N}}{P(\mathsf{Const}(n))} \qquad \frac{x \in \mathsf{Strings}}{P(\mathsf{Var}(x))}$$

$$\frac{e_1 \in \mathsf{Exp} \quad P(e_1) \quad e_2 \in \mathsf{Exp} \quad P(e_2)}{P(\mathsf{Plus}(e_1, e_2))} \qquad \frac{e_1 \in \mathsf{Exp} \quad P(e_1) \quad e_2 \in \mathsf{Exp} \quad P(e_2)}{P(\mathsf{Times}(e_1, e_2))}$$

In other words, to establish $\forall x \in \mathsf{Exp}.\ P(x)$, we need to prove that each of these inference rules is valid.

To see induction in action, we prove a theorem giving a sanity check on our two recursive definitions from earlier: depth can never exceed size.

THEOREM 2.1. *For all* $e \in \mathsf{Exp}$, $\lceil e \rceil \leqslant |e|$.

PROOF. By induction on the structure of e. □

That sort of minimalist proof often surprises and frustrates newcomers. Our position here is that proof checking is an activity fit for machines, not people, so we will leave out gory details, which are to be found in the accompanying Coq code, for this theorem and many others associated with this chapter. Actually, even published proofs on paper tend to use "proofs" as brief as the one above, relying on the reader's experience to "fill in the blanks"! Unsurprisingly, fairly often there are logical errors in such arguments, leading to acceptance of bogus theorems. For that reason, we stick to machine-checked proofs here, using the book chapters to introduce concepts, reasoning principles, and statements of key theorems and lemmas.

2.4. Decidable Theories

We do, however, need to get all the proof details filled in somehow. One of the most convenient cases is when a proof goal fits into some *decidable theory*. We follow the sense from computability theory, where we consider some *decision problem*, as a (usually infinite) set F of formulas and some subset $T \subseteq F$ of *true* formulas, possibly considering only those provable using some limited set of inference rules. The decision problem is *decidable* if and only if there exists some always-terminating program that, when passed some $f \in F$ as input, returns "true" if and only if $f \in T$. Decidability of theories is handy because, whenever our goal belongs to the F set of a decidable theory, we can discharge the goal automatically by running the deciding program that must exist.

One common decidable theory is *linear arithmetic*, whose F set is generated by the following grammar as ϕ.

Constants	n	$\in$	$\mathbb{Z}$
Variables	x	$\in$	$\mathsf{Strings}$
Terms	e	$::=$	$x \mid n \mid e + e \mid e - e$
Propositions	ϕ	$::=$	$e = e \mid e < e \mid \neg\phi \mid \phi \wedge \phi$

The arithmetic terms used here are *linear* in the same sense as *linear algebra*: we never multiply together two terms containing variables. Actually, multiplication

is prohibited outright, but we allow multiplication by a constant as an abbreviation (logically speaking) for repeated addition. Propositions are formed out of equality and less-than tests on terms, and we also have the Boolean negation ("not") operator $\neg$ and conjunction ("and") operator $\wedge$. This set of propositional operators is enough to encode the other usual inequality and propositional operators, so we allow them, too, as convenient shorthands.

Using decidable theories in a proof assistant like Coq, it is important to understand how a theory may apply to formulas that don't actually satisfy its grammar literally. For instance, we may want to prove $f(x) - f(x) = 0$, for some fancy function f well outside the grammar above. However, we only need to introduce a new variable y, defined with the equation $y = f(x)$, to arrive at a new goal $y - y = 0$. A linear-arithmetic procedure makes short work of this goal, and we may then derive the original goal by substituting back in for y. Coq's tactics based on decidable theories do all that hard work for us.

Another important decidable theory is of *equality with uninterpreted functions*.

$$\begin{array}{rrcl} \text{Variables} & x & \in & \mathsf{Strings} \\ \text{Functions} & f & \in & \mathsf{Strings} \\ \text{Terms} & e & ::= & x \mid f(e, \ldots, e) \\ \text{Propositions} & \phi & ::= & e = e \mid \neg\phi \mid \phi \wedge \phi \end{array}$$

In this theory, we know nothing about the detailed properties of the variables or functions that we use. Instead, we must reason solely from the basic properties of equality:

$$\frac{}{e = e}\ \mathsf{Reflexivity} \quad \frac{e_2 = e_1}{e_1 = e_2}\ \mathsf{Symmetry} \quad \frac{e_1 = e_3 \quad e_3 = e_2}{e_1 = e_2}\ \mathsf{Transitivity}$$

$$\frac{f = f' \quad e_1 = e_1' \quad \ldots \quad e_n = e_n'}{f(e_1, \ldots, e_n) = f'(e_1', \ldots, e_n')}\ \mathsf{Congruence}$$

As one more example of a decidable theory, consider the algebraic structure of *semirings*, which may profitably be remembered as "types that act like natural numbers." A semiring is any set containing two elements notated 0 and 1, closed under two binary operators notated $+$ and $\times$. The notations are suggestive, but in fact we have free reign in choosing the set, elements, and operators, so long as the

following axioms[1] are satisfied:

$$\begin{array}{rcl}
(a+b)+c & = & a+(b+c) \\
0+a & = & a \\
a+0 & = & a \\
a+b & = & b+a \\
(a\times b)\times c & = & a\times(b\times c) \\
1\times a & = & a \\
a\times 1 & = & a \\
a\times(b+c) & = & (a\times b)+(a\times c) \\
(a+b)\times c & = & (a\times c)+(b\times c) \\
0\times a & = & 0 \\
a\times 0 & = & 0
\end{array}$$

The formal theory is then as follows, where we consider as "true" only those equalities that follow from the axioms.

$$\begin{array}{rcll}
\text{Variables} & x & \in & \mathsf{Strings} \\
\text{Terms} & e & ::= & x \mid e+e \mid e\times e \\
\text{Propositions} & \phi & ::= & e=e
\end{array}$$

Note how the applicability of the semiring theory is incomparable to the applicability of the linear-arithmetic theory. That is, while some goals are provable via either, some are provable only via the semiring theory and some provable only by linear arithmetic. For instance, by the semiring theory, we can prove $x(y+z) = xy+xz$, while linear arithmetic can prove $x - x = 0$.

2.5. Simplification and Rewriting

While we leave most proof details to the accompanying Coq code, it does seem important to introduce two key principles that are often implicit in proofs on paper.

The first is *algebraic simplification*, where we apply the defining equations of a recursive definition to simplify a goal. For example, recall that our definition of expression size included this clause.

$$|\mathsf{Plus}(e_1, e_2)| \quad = \quad 1 + |e_1| + |e_2|$$

Now imagine that we are trying to prove this formula.

$$|\mathsf{Plus}(e, \mathsf{Const}(7))| = 2 + |e|$$

We may apply the defining equation to rewrite into a different formula, where we have essentially pushed the definition of $|\cdot|$ through the Plus.

$$1 + |e| + |\mathsf{Const}(7)| = 2 + |e|$$

Another application of a different defining equation, this time for Const, takes us to here.

$$1 + |e| + 1 = 2 + |e|$$

From here, the goal follows by linear arithmetic.

[1]The equations are taken almost literally from `https://en.wikipedia.org/wiki/Semiring`.

Such a proof establishes a theorem $\forall e \in \mathsf{Exp}.\ |\mathsf{Plus}(e, \mathsf{Const}(7))| = 2 + |e|$. We may use already-proved theorems via a more general *rewriting* mechanism, applying whenever we know some quantified equality. Within a new goal we are proving, we find some subterm that matches the lefthand side of that equality, after we choose the proper values of the quantified variables. The process of finding those values automatically is called *unification*. Rewriting enables us to take the subterm we found and replace it with the righthand side of the equation.

As an example, assume that, for some P, we know $P(2 + |\mathsf{Var}(x)|)$ and are trying to prove $P(|\mathsf{Plus}(\mathsf{Var}(x), \mathsf{Const}(7))|)$. We may use our earlier fact to rewrite the argument of P in what we are trying to show, so that it now matches the argument from what we already know, at which point the proof is trivial to finish. Here, unification found the assignment $e = \mathsf{Var}(x)$.

Encoding

We close the chapter with an important note on terminology. A formula like $P(|\mathsf{Plus}(\mathsf{Var}(x), \mathsf{Const}(7))|)$ combines several levels of notation. We consider that we are doing our mathematical reasoning in some *metalanguage*, which is often applicable to a wide variety of proof tasks. We also happen to be applying it here to reason about some *object language*, a programming language whose syntax is defined formally, here the language of arithmetic expressions. We have x as a variable of the metalanguage, while $\mathsf{Var}(x)$ is a variable expression of the object language. It is difficult to use English to explain the distinction between the two in complete formality, but be on the lookout for places where formulas mix concepts of the metalanguage and object language! The general patterns should soon become clear, as they are somehow already familiar to us from natural-language sentences like:

> The wise man said, "it is time to prove some theorems."

The quoted remark could just as well be in Spanish instead of English, in which case we have two languages nested in a nontrivial way.

CHAPTER 3

Data Abstraction

All of the fully formal proofs in this book are worked out only in associated Coq code. Therefore, before proceeding to more topics in program semantics and proof, it is important to develop some basic Coq competence. Several heavily commented examples files are associated with this crucial point in the book. We won't discuss details of Coq proving in this document, outside Appendix A. However, one of the possibilities shown off in the Coq code is worth drawing attention to, as a celebrated semantics idea in its own right, though we don't yet connect it to formalized syntax of programming languages. That idea is *data abstraction*, one of the most central ideas in program structuring. Let's consider the mathematical meaning of *encapsulation* in data structures.

3.1. Algebraic Interfaces for Abstract Data Types

Consider the humble queue, a classic data structure that allows us to enqueue data elements and then dequeue them in the order received. Perhaps surprisingly, there is already some complexity in efficient queue implementation. So-called *client code* that relies on queues shouldn't need to know about that complexity, though. We should be able to formulate "queue" as an *abstract data type*, hiding implementation details. In the setting of pure functional programming, as in Coq, here is our first cut at such a data type, as a set of types and operations, somewhat reminiscent of e.g. interfaces in Java. Type $\mathsf{t}(\alpha)$ stands for queues holding data values in some type α.

$$\begin{array}{rcl} \mathsf{t}(\alpha) & : & \mathsf{Set} \\ \mathsf{empty} & : & \mathsf{t}(\alpha) \\ \mathsf{enqueue} & : & \mathsf{t}(\alpha) \times \alpha \to \mathsf{t}(\alpha) \\ \mathsf{dequeue} & : & \mathsf{t}(\alpha) \rightharpoonup \mathsf{t}(\alpha) \times \alpha \end{array}$$

A few notational conventions of note: We declare that $\mathsf{t}(\alpha)$ is a type by assigning it the type Set, which itself contains all the normal types of programming. An empty queue exists for any α, and enqueue and dequeue operations are also available for any α. The type of $\mathsf{dequeue}$ indicates function partiality by the arrow $\rightharpoonup$: dequeuing yields no answer for an empty queue. For partial function $f : A \rightharpoonup B$, we indicate lack of a mapping for $x \in A$ by writing $f(x) = \cdot$.

In normal programming, we stop at this level of detail in defining an abstract data type. However, when we're after formal correctness proofs, we must enrich data types with *specifications* or "specs." One prominent spec style is *algebraic*: write out a set of *laws*, quantified equalities that use the operations of the data type. For queues, here are two reasonable laws.

$$\begin{array}{l}\mathsf{dequeue}(\mathsf{empty}) = \cdot \\ \forall q.\ \mathsf{dequeue}(q) = \cdot \Rightarrow q = \mathsf{empty}\end{array}$$

Actually, the inference-rule notation from last chapter also makes algebraic laws more readable, so here is a restatement.

$$\frac{}{\mathsf{dequeue}(\mathsf{empty}) = \cdot} \quad \frac{\mathsf{dequeue}(q) = \cdot}{q = \mathsf{empty}}$$

One more rule suffices to give a complete characterization of behavior, with the familiar math notation for piecewise functions.

$$\mathsf{dequeue}(\mathsf{enqueue}(q, x)) = \begin{cases}(\mathsf{empty}, x), & \mathsf{dequeue}(q) = \cdot \\ (\mathsf{enqueue}(q', x), y), & \mathsf{dequeue}(q) = (q', y)\end{cases}$$

Now several implementations of this functionality are possible. Here's one of the two "obvious" ones, where we enqueue to list fronts and dequeue from list backs. We write $\mathsf{list}(\alpha)$ for the type of lists with data elements from α, with $\ell_1 \bowtie \ell_2$ for concatenation of lists ℓ_1 and ℓ_2, and with comma-separated lists inside square brackets for list literals.

$$\begin{array}{rcl}\mathsf{t}(\alpha) & = & \mathsf{list}(\alpha) \\ \mathsf{empty} & = & [] \\ \mathsf{enqueue}(q, x) & = & [x] \bowtie q \\ \mathsf{dequeue}([]) & = & \cdot \\ \mathsf{dequeue}([x] \bowtie q) & = & ([], x), \text{ when } \mathsf{dequeue}(q) = \cdot. \\ \mathsf{dequeue}([x] \bowtie q) & = & ([x] \bowtie q', y), \text{ when } \mathsf{dequeue}(q) = (q', y).\end{array}$$

There is also a dual implementation where we enqueue to list backs and dequeue from list fronts.

$$\begin{array}{rcl}\mathsf{t}(\alpha) & = & \mathsf{list}(\alpha) \\ \mathsf{empty} & = & [] \\ \mathsf{enqueue}(q, x) & = & q \bowtie [x] \\ \mathsf{dequeue}([]) & = & \cdot \\ \mathsf{dequeue}([x] \bowtie q) & = & (q, x)\end{array}$$

Proofs of the algebraic laws, for both implementations, appear in the associated Coq code. Both versions actually take quadratic time in practice, assuming concatenation takes time linear in the length of its first argument. There is a famous, more clever implementation that achieves amortized constant time (linear time to run a whole sequence of operations), but we will need to expand our algebraic style to accommodate it.

3.2. Algebraic Interfaces with Custom Equivalence Relations

We find it useful to extend the base interface of queues with a new, mathematical "operation":

$$\begin{array}{rcl} \mathsf{t}(\alpha) & : & \mathsf{Set} \\ \mathsf{empty} & : & \mathsf{t}(\alpha) \\ \mathsf{enqueue} & : & \mathsf{t}(\alpha) \times \alpha \to \mathsf{t}(\alpha) \\ \mathsf{dequeue} & : & \mathsf{t}(\alpha) \rightharpoonup \mathsf{t}(\alpha) \times \alpha \\ \approx & : & \mathcal{P}(\mathsf{t}(\alpha) \times \mathsf{t}(\alpha)) \end{array}$$

We use the "powerset" operation $\mathcal{P}$ to indicate that $\approx$ is a *binary relation* over queues (of the same type). Our intention is that $\approx$ be an *equivalence relation*, as formalized by the following laws that we add.

$$\frac{}{a \approx a}\ \mathsf{Reflexivity} \qquad \frac{b \approx a}{a \approx b}\ \mathsf{Symmetry} \qquad \frac{a \approx b \quad b \approx c}{a \approx c}\ \mathsf{Transitivity}$$

Now we rewrite the original laws to use $\approx$ instead of equality. We implicitly lift $\approx$ to apply to results of the partial function dequeue: nonexistent results $\cdot$ are related, and existent results (q_1, x_1) and (q_2, x_2) are related iff $q_1 \approx q_2$ and $x_1 = x_2$.

$$\frac{}{\mathsf{dequeue}(\mathsf{empty}) = \cdot} \qquad \frac{\mathsf{dequeue}(q) = \cdot}{q \approx \mathsf{empty}}$$

$$\mathsf{dequeue}(\mathsf{enqueue}(q, x)) \approx \begin{cases} (\mathsf{empty}, x), & \mathsf{dequeue}(q) = \cdot \\ (\mathsf{enqueue}(q', x), y), & \mathsf{dequeue}(q) = (q', y) \end{cases}$$

What's the payoff from this reformulation? Well, first, it passes the sanity check that the two queue implementations from the last section comply, with $\approx$ instantiated as simple equality. However, we may now also handle the classic *two-stack queue*. Here is its implementation, relying on list-reversal function rev (which takes linear time).

$$\begin{array}{rcl} \mathsf{t}(\alpha) & = & \mathsf{list}(\alpha) \times \mathsf{list}(\alpha) \\ \mathsf{empty} & = & ([], []) \\ \mathsf{enqueue}((\ell_1, \ell_2), x) & = & ([x] \bowtie \ell_1, \ell_2) \\ \mathsf{dequeue}(([], [])) & = & \cdot \\ \mathsf{dequeue}((\ell_1, [x] \bowtie \ell_2)) & = & ((\ell_1, \ell_2), x) \\ \mathsf{dequeue}((\ell_1, [])) & = & (([], q_1'), x), \text{ when } \mathsf{rev}(\ell_1) = [x] \bowtie q_1'. \end{array}$$

The basic trick is to encode a queue as a pair of lists (ℓ_1, ℓ_2). We try to enqueue into ℓ_1 by adding elements to its front in constant time, and we try to dequeue from ℓ_2 by removing elements from its front in constant time. However, sometimes we run out of elements in ℓ_2 and need to *reverse* ℓ_1 and transfer the result into ℓ_2. The suitable equivalence relation formalizes this plan.

$$\begin{array}{rcl} \mathsf{rep}((\ell_1, \ell_2)) & = & \ell_1 \bowtie \mathsf{rev}(\ell_2) \\ q_1 \approx q_2 & = & \mathsf{rep}(q_1) = \mathsf{rep}(q_2) \end{array}$$

We can prove both that this $\approx$ is an equivalence relation and that the other queue laws are satisfied. As a result, client code (and its correctness proofs) can use

this fancy code, effectively viewing it as a simple queue, with the two-stack nature hidden.

Why did we need to go through the trouble of introducing custom equivalence relations? Consider the following two queues. Are they equal? (We write π_1 for the function that projects out the first element of a pair.)

$$\mathsf{enqueue}(\mathsf{empty}, 2) \stackrel{?}{=} \pi_1(\mathsf{dequeue}(\mathsf{enqueue}(\mathsf{enqueue}(\mathsf{empty}, 1), 2)))$$

No, they aren't equal! The first expression reduces to $([2], [])$, while the second reduces to $([], [2])$. This data structure is *noncanonical*, in the sense that the same logical value may have multiple physical representations. The equivalence relation lets us indicate which physical representations are equivalent.

3.3. Representation Functions

That last choice of equivalence relations suggests another specification style, based on *representation functions*. We can force every queue to include a function to convert to a standard, canonical representation. Real executable programs shouldn't generally call that function; it's most useful to us in phrasing the algebraic laws. Perhaps surprisingly, the mere existence of any compatible function is enough to show correctness of a queue implementation, and the approach generalizes to essentially all other data structures cast as abstract data types.

Here is how we revise our type signature for queues.

$$\begin{aligned} \mathsf{t}(\alpha) &: \mathsf{Set} \\ \mathsf{empty} &: \mathsf{t}(\alpha) \\ \mathsf{enqueue} &: \mathsf{t}(\alpha) \times \alpha \to \mathsf{t}(\alpha) \\ \mathsf{dequeue} &: \mathsf{t}(\alpha) \rightharpoonup \mathsf{t}(\alpha) \times \alpha \\ \mathsf{rep} &: \mathsf{t}(\alpha) \to \mathsf{list}(\alpha) \end{aligned}$$

And here are the revised axioms.

$$\frac{}{\mathsf{rep}(\mathsf{empty}) = []} \qquad \frac{}{\mathsf{rep}(\mathsf{enqueue}(q, x)) = [x] \bowtie \mathsf{rep}(q)}$$

$$\frac{\mathsf{rep}(q) = []}{\mathsf{dequeue}(q) = \cdot} \qquad \frac{\mathsf{rep}(q) = \ell \bowtie [x]}{\exists q'.\ \mathsf{dequeue}(q) = (q', x) \wedge \mathsf{rep}(q') = \ell}$$

Notice that this specification style can also be viewed as *giving a reference implementation of the data type*, where rep shows how to convert back to the reference implementation at any point.

3.4. Fixing Parameter Types for Abstract Data Types

Here's another classic abstract data type: finite sets, where we write $\mathbb{B}$ for the set of Booleans.

$$\begin{aligned} \mathsf{t}(\alpha) &: \mathsf{Set} \\ \mathsf{empty} &: \mathsf{t}(\alpha) \\ \mathsf{add} &: \mathsf{t}(\alpha) \times \alpha \to \mathsf{t}(\alpha) \\ \mathsf{member} &: \mathsf{t}(\alpha) \times \alpha \to \mathbb{B} \end{aligned}$$

A few laws characterize expected behavior, with $\top$ and $\bot$ the respective elements "true" and "false" of $\mathbb{B}$.

$$\frac{}{\mathsf{member}(\mathsf{empty}, k) = \bot} \quad \frac{}{\mathsf{member}(\mathsf{add}(s, k), k) = \top} \quad \frac{k_1 \neq k_2}{\mathsf{member}(\mathsf{add}(s, k_1), k_2) = \mathsf{member}(s, k_2)}$$

There is a simple generic implementation of this data type with unsorted lists.

$$\begin{aligned}
\mathsf{t} &= \mathsf{list} \\
\mathsf{empty} &= [] \\
\mathsf{add}(s, k) &= [k] \bowtie s \\
\mathsf{member}([], k) &= \bot \\
\mathsf{member}([k'] \bowtie s, k) &= k = k' \vee \mathsf{member}(s, k)
\end{aligned}$$

However, we can build specialized finite sets for particular element types and usage patterns. For instance, assume we are working with sets of natural numbers, where we know that most sets contain consecutive numbers. In those cases, it suffices to store just the lowest and highest elements of sets, and all the set operations run in constant time. Assume a fallback implementation of finite sets, with type t_0 and operations empty_0, add_0, and member_0. We implement our optimized set type like so, assuming an operation $\mathsf{fromRange} : \mathbb{N} \times \mathbb{N} \to \mathsf{t}_0$ to turn a range into an ad-hoc set.

$$\begin{aligned}
\mathsf{t} &= \mathsf{Empty} \mid \mathsf{Range}(\mathbb{N} \times \mathbb{N}) \mid \mathsf{AdHoc}(\mathsf{t}_0) \\
\mathsf{empty} &= \mathsf{Empty} \\
\mathsf{add}(\mathsf{Empty}, k) &= \mathsf{Range}(k, k) \\
\mathsf{add}(\mathsf{Range}(n_1, n_2), k) &= \mathsf{Range}(n_1, n_2), \text{ when } n_1 \leqslant k \leqslant n_2 \\
\mathsf{add}(\mathsf{Range}(n_1, n_2), n_1 - 1) &= \mathsf{Range}(n_1 - 1, n_2), \text{ when } n_1 \leqslant n_2 \\
\mathsf{add}(\mathsf{Range}(n_1, n_2), n_2 + 1) &= \mathsf{Range}(n_1, n_2 + 1), \text{ when } n_1 \leqslant n_2 \\
\mathsf{add}(\mathsf{Range}(n_1, n_2), k) &= \mathsf{AdHoc}(\mathsf{add}_0(\mathsf{fromRange}(n_1, n_2), k)), \text{ otherwise} \\
\mathsf{add}(\mathsf{AdHoc}(s), k) &= \mathsf{AdHoc}(\mathsf{add}_0(s, k)) \\
\mathsf{member}(\mathsf{Empty}, k) &= \bot \\
\mathsf{member}(\mathsf{Range}(n_1, n_2), k) &= n_1 \leqslant k \leqslant n_2 \\
\mathsf{member}(\mathsf{AdHoc}(s), k) &= \mathsf{member}_0(s, k)
\end{aligned}$$

This implementation can be proven to satisfy the finite-set spec, assuming that the baseline ad-hoc implementation does, too. For workloads that only build sets of consecutive numbers, this implementation can be much faster than the generic list-based implementation, converting quadratic-time algorithms into linear-time.

CHAPTER 4

Semantics via Interpreters

That's enough about what programs *look like*. Let's shift our attention to what programs *mean*.

4.1. Semantics for Arithmetic Expressions via Finite Maps

To explain the meaning of one of Chapter 2's arithmetic expressions, we need a way to indicate the value of each variable. A theory of *finite maps* is helpful here. We apply the following notations throughout the book:

Encoding

$\bullet$	empty map, with $\varnothing$ as its domain
$m(k)$	mapping of key k in map m
$m[k \mapsto v]$	extension of map m to also map key k to value v

As the name advertises, finite maps are functions with finite domains, where the domain may be expanded by each extension operation. Two axioms explain the essential interactions of the basic operators.

$$\frac{}{m[k \mapsto v](k) = v} \qquad \frac{k_1 \neq k_2}{m[k_1 \mapsto v](k_2) = m(k_2)}$$

With these operators in hand, we can write a semantics for arithmetic expressions. This is a recursive function that *maps variable valuations to numbers*. We write $[\![e]\!]$ for the meaning of e; this notation is often referred to as *Oxford brackets*. Recall that we allow notations like this as syntactic sugar for arbitrary functions, even when giving the equations that define those functions. We write v for a valuation (finite map).

Encoding

$$\begin{aligned}
[\![n]\!]v &= n \\
[\![x]\!]v &= v(x) \\
[\![e_1 + e_2]\!]v &= [\![e_1]\!]v + [\![e_2]\!]v \\
[\![e_1 \times e_2]\!]v &= [\![e_1]\!]v \times [\![e_2]\!]v
\end{aligned}$$

Note how parts of the definition feel a little bit like cheating, as we just "push notations inside the brackets." It's important to remember that plus *inside* the brackets is syntax, while plus *outside* the brackets is the normal addition of math!

To test our semantics, we define a *variable substitution* function. A substitution $[e'/x]e$ stands for the result of running through the syntax of e, replacing every occurrence of variable x with expression e'.

$$\begin{array}{rcl} [e/x]n & = & n \\ [e/x]x & = & e \\ [e/x]y & = & y, \text{ when } y \neq x \\ [e/x](e_1 + e_2) & = & [e/x]e_1 + [e/x]e_2 \\ [e/x](e_1 \times e_2) & = & [e/x]e_1 \times [e/x]e_2 \end{array}$$

We can prove a key compatibility property of these two recursive functions.

THEOREM 4.1. *For all e, e', x, and v, $[\![[e'/x]e]\!]v = [\![e]\!](v[x \mapsto [\![e']\!]v])$.*

That is, in some sense, the operations of interpretation and substitution *commute* with each other. That intuition gives rise to the common notion of a *commuting diagram*, like the one below for this particular example.

$$\begin{array}{ccc} (e, v) & \xrightarrow{[e'/x]\ldots} & ([e'/x]e, v) \\ \Big\downarrow{\scriptstyle \ldots[x \mapsto [\![e']\!]v]} & & \Big\downarrow{\scriptstyle [\![\ldots]\!]} \\ (e, v[x \mapsto [\![e']\!]v]) & \xrightarrow{[\![\ldots]\!]} & [\![[e'/x]e]\!]v \end{array}$$

We start at the top left, with a given expression e and valuation v. The diagram shows the equivalence of *two different paths* to the bottom right. Each individual arrow is labeled with some description of the transformation it performs, to get from the term at its source to the term at its destination. The right-then-down path is based on substituting and then interpreting, while the down-then-right path is based on extending the valuation and then interpreting. Since both paths wind up at the same spot, the diagram indicates an equality between the corresponding terms.

It's a matter of taste whether the theorem statement or the diagram expresses the property more clearly!

4.2. A Stack Machine

As an example of a very different language, consider a *stack machine*, similar at some level to, for instance, the Forth programming language, or to various postfix calculators.

Encoding

$$\begin{array}{rrcl} \text{Instructions} & i & ::= & \mathsf{PushConst}(n) \mid \mathsf{PushVar}(x) \mid \mathsf{Add} \mid \mathsf{Multiply} \\ \text{Programs} & \overline{i} & ::= & \cdot \mid i; \overline{i} \end{array}$$

Though here we defined an explicit grammar for programs, which are just sequences of instructions, in general we'll use the notation $\overline{X}$ to stand for sequences of X's, and the associated concrete syntax won't be so important. We also freely use single instructions to stand for programs, writing just i in place of $i; \cdot$.

Each instruction of this language transforms a *stack*, a last-in-first-out list of numbers. Rather than spend more words on it, here is an interpreter that makes everything precise. Here and elsewhere, we overload the Oxford brackets $[\![\ldots]\!]$ shamelessly, where context makes clear which language or interpreter we are dealing with. We write s for stacks, and we write $n \rhd s$ for pushing number n onto the top of stack s.

Encoding

$$\begin{aligned}
[\![\mathsf{PushConst}(n)]\!](v, s) &= n \rhd s \\
[\![\mathsf{PushVar}(x)]\!](v, s) &= v(x) \rhd s \\
[\![\mathsf{Add}]\!](v, n_2 \rhd n_1 \rhd s) &= (n_1 + n_2) \rhd s \\
[\![\mathsf{Multiply}]\!](v, n_2 \rhd n_1 \rhd s) &= (n_1 \times n_2) \rhd s
\end{aligned}$$

The last two cases require the stack have at least a certain height. Here we'll ignore what happens when the stack is too short, though it suffices, for our purposes, to add pretty much any default behavior for the missing cases. We overload $[\![\overline{i}]\!]$ to refer to the *composition* of the interpretations of the different instructions within $\overline{i}$, in order.

Next, we give our first example of what might be called a *compiler*, or a translation from one language to another. Let's compile arithmetic expressions into stack programs, which then become easy to map onto the instructions of common assembly languages. In that sense, with this translation, we make progress toward efficient implementation on commodity hardware.

Throughout this book, we will use notation $\lfloor \ldots \rfloor$ for compilation, where the floor-based notation suggests *moving downward* to a lower abstraction level. Here is the compiler that concerns us now, where we write $\overline{i_1} \bowtie \overline{i_2}$ for concatenation of two instruction sequences $\overline{i_1}$ and $\overline{i_2}$.

Encoding

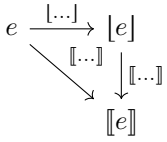
$$\begin{aligned}
\lfloor n \rfloor &= \mathsf{PushConst}(n) \\
\lfloor x \rfloor &= \mathsf{PushVar}(x) \\
\lfloor e_1 + e_2 \rfloor &= \lfloor e_1 \rfloor \bowtie \lfloor e_2 \rfloor \bowtie \mathsf{Add} \\
\lfloor e_1 \times e_2 \rfloor &= \lfloor e_1 \rfloor \bowtie \lfloor e_2 \rfloor \bowtie \mathsf{Multiply}
\end{aligned}$$

The first two cases are straightforward: their compilations just push the obvious values onto the stack. The binary operators are just slightly more tricky. Each first evaluates its operands in order, where each operand leaves its final result on the stack. With both of them in place, we run the instruction to pop them, combine them, and push the result back onto the stack.

The correctness theorem for compilation must refer to both of our interpreters. From here on, we consider that all unaccounted-for variables in a theorem statement are quantified universally.

THEOREM 4.2. $[\![\lfloor e \rfloor]\!](v, \cdot) = [\![e]\!]v$.

Here's a restatement as a commuting diagram.



As usual, we leave proof details for the associated Coq code, but the key insight of the proof is to strengthen the induction hypothesis via a lemma.

LEMMA 4.3. $[\![\lfloor e \rfloor \bowtie \overline{i}]\!](v, s) = [\![\overline{i}]\!](v, [\![e]\!]v \rhd s)$.

We strengthen the statement by considering both an arbitrary initial stack s and a sequence of extra instructions $\overline{i}$ to be run after e.

4.3. A Simple Higher-Level Imperative Language

The interpreter approach to semantics is usually the most convenient one, when it applies. Coq requires that all programs terminate, and that requirement is effectively also present in informal math, though it is seldom called out with the same terms. Instead, with math, we worry about whether recursive systems of equations are well-founded, in appropriate senses. From either perspective, extra encoding tricks are required to write a well-formed interpreter for a Turing-complete language. We will dodge those complexities for now by defining a simple imperative language with bounded loops, where termination is easy to prove. We take the arithmetic expression language as a base.

Encoding

$$\text{Command} \quad c \quad ::= \quad \mathsf{skip} \mid x \leftarrow e \mid c; c \mid \mathsf{repeat}\ e\ \mathsf{do}\ c\ \mathsf{done}$$

Now the implicit state, read and written by a command, is a variable valuation, as we used in the interpreter for expressions. A skip command does nothing, while $x \leftarrow e$ extends the valuation to map x to the value of expression e. We have simple command sequencing $c_1; c_2$, in addition to the bounded loop repeat e do c done, which executes c a number of times equal to the value of e.

To give the semantics, we need a few commonplace notations that are worth reviewing. We write id for the identity function, where $\mathsf{id}(x) = x$; and we write $f \circ g$ for composition of functions f and g, where $(f \circ g)(x) = f(g(x))$. We also have iterated self-composition, written like *exponentiation* of functions f^n, defined as follows.

$$\begin{aligned} f^0 &= \mathsf{id} \\ f^{n+1} &= f^n \circ f \end{aligned}$$

Encoding

From here, $[\![\ldots]\!]$ is easy to define yet again, as a transformer over variable valuations.

$$\begin{aligned} [\![\mathsf{skip}]\!]v &= v \\ [\![x \leftarrow e]\!]v &= v[x \mapsto [\![e]\!]v] \\ [\![c_1; c_2]\!]v &= [\![c_2]\!]([\![c_1]\!]v) \\ [\![\mathsf{repeat}\ e\ \mathsf{do}\ c\ \mathsf{done}]\!]v &= [\![c]\!]^{[\![e]\!]v}(v) \end{aligned}$$

To put this semantics through a workout, let's consider a simple *optimization*, a transformation whose input and output programs are in the same language. There's an additional, fuzzier criterion for an optimization, which is that it should improve the program somehow, usually in terms of running time, memory usage, etc. The optimization we choose here may be a bit dubious in that respect, though it is related to an optimization found in every serious C compiler.

In particular, let's tackle *loop unrolling*. When the iteration count of a loop is a constant n, we can replace the loop with n sequenced copies of its body. C compilers need to work harder to find the iteration count of a loop, but luckily our language includes loops with very explicit iteration counts! To define the transformation, we'll want a recursive function and notation for sequencing of n copies of a command c, written ${}^n c$.

$$\begin{aligned} {}^0 c &= \mathsf{skip} \\ {}^{n+1} c &= c; {}^n c \end{aligned}$$

Now the optimization itself is easy to define. We'll write $|\ldots|$ for this and other optimizations, which move neither down nor up a tower of program abstraction levels.

Encoding

$$\begin{array}{rcl} |\mathsf{skip}| & = & \mathsf{skip} \\ |x \leftarrow e| & = & x \leftarrow e \\ |c_1; c_2| & = & |c_1|; |c_2| \\ |\mathsf{repeat}\ n\ \mathsf{do}\ c\ \mathsf{done}| & = & {}^n|c| \\ |\mathsf{repeat}\ e\ \mathsf{do}\ c\ \mathsf{done}| & = & \mathsf{repeat}\ e\ \mathsf{do}\ |c|\ \mathsf{done} \end{array}$$

Note that, when multiple defining equations apply to some function input, by convention we apply the *earliest* equation that matches.

Let's prove that this optimization preserves program behavior; that is, we prove that it is *semantics preserving*.

Theorem 4.4. $[\![|c|]\!]v = [\![c]\!]v$.

It all looks so straightforward from that statement, doesn't it? Indeed, there actually isn't so much work to do to prove this theorem. We can also present it as a commuting diagram much like the prior one.

$$\begin{array}{ccc} c & \xrightarrow{|\ldots|} & |c| \\ & \searrow^{[\![\ldots]\!]} & \downarrow{\scriptstyle [\![\ldots]\!]} \\ & & [\![c]\!] \end{array}$$

The statement of Theorem 4.4 happens to already be in the right form to do induction directly, but we need a helper lemma, capturing the interaction of ${}^n c$ and the semantics.

Lemma 4.5. $[\![{}^n c]\!] = [\![c]\!]^n$.

Let us end the chapter with the commuting-diagram version of the lemma statement.

$$\begin{array}{ccc} c & \xrightarrow{{}^n\ldots} & {}^n c \\ \downarrow{\scriptstyle [\![\ldots]\!]} & & \downarrow{\scriptstyle [\![\ldots]\!]} \\ [\![c]\!] & \xrightarrow{\ldots^n} & [\![c]\!]^n \end{array}$$

CHAPTER 5

Transition Systems and Invariants

For simple programming languages where programs always terminate, it is often most convenient to formalize them using interpreters, as in the last chapter. However, many important languages don't fall into that category, and for them we need different techniques. Nontermination isn't always a bug; for instance, we expect a network server to run indefinitely. We still need to be able to talk about the correct behavior of programs that run forever, by design. For that reason, in this chapter and in most of the rest of the book, we model programs using relations, in much the same way that may be familiar from automata theory. An important difference, though, is that, while undergraduate automata-theory classes generally study *finite-state machines*, for general program reasoning we want to allow infinite sets of states, otherwise referred to as *infinite-state systems*.

Let's start with an example that almost seems too mundane to be associated with such terms.

5.1. Factorial as a State Machine

We're familiar with the factorial operation, implemented as an imperative program with a loop.

```
factorial(n) {
  a = 1;
  while (n > 0) {
    a = a * n;
    n = n - 1;
  }
  return a;
}
```

In the analysis to follow, consider some value $n_0 \in \mathbb{N}$ fixed, as the input passed to this operation. A state machine is lurking within the surface syntax of the program. In fact, we have a variety of choices in modeling it as a state machine. Here is the set of states that we choose to use here:

Encoding

$$\begin{array}{rrcl} \text{Natural numbers} & n & \in & \mathbb{N} \\ \text{States} & s & ::= & \mathsf{AnswerIs}(n) \mid \mathsf{WithAccumulator}(n, n) \end{array}$$

There are two types of states. An $\mathsf{AnswerIs}(a)$ state corresponds to the `return` statement. It records the final result a of the factorial operation. A $\mathsf{WithAccumulator}(n, a)$ records an intermediate state, giving the values of the two local variables, just before a loop iteration begins.

Following the more familiar parts of automata theory, let's define a set of *initial states* for this machine.

$$\frac{}{\mathsf{WithAccumulator}(n_0, 1) \in \mathcal{F}_0}$$

For consistency with the notation we will be using later, we define the set $\mathcal{F}_0$ using an inference rule. Equivalently, we could just write $\mathcal{F}_0 = \{\mathsf{WithAccumulator}(n_0, 1)\}$, essentially reading off the initial variable values from the first lines of the code above.

Similarly, we also define a set of *final states*.

$$\frac{}{\mathsf{AnswerIs}(a) \in \mathcal{F}_\omega}$$

Equivalently: $\mathcal{F}_\omega = \{\mathsf{AnswerIs}(a) \mid a \in \mathbb{N}\}$. Note that this definition only captures when the program is *done*, not when it *returns the right answer*. It follows from the last line of the code.

The last and most important ingredient of our state machine is its *transition relation*, where we write $s \to s'$ to indicate that state s advances to state s' in one step, following the semantics of the program. Here inference rules are more obviously a good fit.

$$\frac{}{\mathsf{WithAccumulator}(0, a) \to \mathsf{AnswerIs}(a)}$$

$$\frac{}{\mathsf{WithAccumulator}(n+1, a) \to \mathsf{WithAccumulator}(n, a \times (n+1))}$$

The first rule corresponds to the case where the program ends, because the loop test has failed and we now know the final answer. The second rule corresponds to going once around the loop, following directly from the code in the loop body.

We can fit these ingredients into the general concept of a *transition system*, the term we will use throughout this book for this sort of state machine. Actually, the words "state machine" suggest to many people that the state set must be finite, hence our preference for "transition system," which is also used fairly frequently in semantics.

DEFINITION 5.1. A *transition system* is a triple $\langle S, S_0, \to \rangle$, with S a set of states, $S_0 \subseteq S$ a set of initial states, and $\to \subseteq S \times S$ a transition relation.

For an arbitrary transition relation $\to$, not just the one defined above for factorial, we define its *transitive-reflexive closure* $\to^*$ with two inference rules:

$$\frac{}{s \to^* s} \qquad \frac{s \to s' \quad s' \to^* s''}{s \to^* s''}$$

That is, a formal claim $s \to^* s'$ corresponds exactly to the informal claim that "starting from state s, we can reach state s'."

DEFINITION 5.2. For transition system $\langle S, S_0, \to \rangle$, we say that a state s is *reachable* if and only if there exists $s_0 \in S_0$ such that $s_0 \to^* s$.

Building on these notations, here is one way to state the correctness of our factorial program, which, defining S according to the state grammar above, we model as $\mathcal{F} = \langle S, \mathcal{F}_0, \to \rangle$.

THEOREM 5.3. *For any state s reachable in $\mathcal{F}$, if $s \in \mathcal{F}_\omega$, then $s = \mathsf{AnswerIs}(n_0!)$.*

That is, whenever the program finishes, it returns the right answer. (Recall that n_0 is the initial value of the input variable.)

We could prove this theorem now in a relatively ad-hoc way. Instead, let's develop the general machinery of *invariants*.

5.2. Invariants

The concept of "invariant" may be familiar from such relatively informal notions as "loop invariant" in introductory programming classes. Intuitively, an invariant is a property of program state that *starts true and stays true*, but let's make that idea a bit more formal, as applied to our transition-system formalism.

Invariants

DEFINITION 5.4. An *invariant* of a transition system is a property that is always true, in all of the system's reachable states. That is, for transition system $\langle S, S_0, \rightarrow \rangle$, where R is the set of all its reachable states, some $I \subseteq S$ is an invariant iff $R \subseteq I$. (Note that here we adopt the mathematical convention that "properties" of states and "sets" of states are synonymous, so that in each case we can use what terminology seems most natural. The "property" holds of exactly those states that belong to the "set.")

At first look, the definition may appear a bit silly. Why not always just take the reachable states R as the invariant, instead of scrambling to invent something new? The reason is the same as for strengthening induction hypotheses to make proofs easier. Often it is easier to characterize an invariant that isn't fully precise, admitting some states that the system can never actually reach. Additionally, it can be easier to prove existence of an approximate invariant by induction, by the method that the next key theorem formalizes.

THEOREM 5.5. *Consider a transition system $\langle S, S_0, \rightarrow \rangle$ and its candidate invariant I. The candidate is truly an invariant if (1) $S_0 \subseteq I$ and (2) for every $s \in I$ where $s \rightarrow s'$, we also have $s' \in I$.*

That's enough generalities for now. Let's define a suitable invariant for factorial.

Invariants

$$\begin{aligned} I(\mathsf{AnswerIs}(a)) &= n_0! = a \\ I(\mathsf{WithAccumulator}(n, a)) &= n_0! = n! \times a \end{aligned}$$

It is an almost-routine exercise to prove that I really is an invariant, using Theorem 5.5. The key new ingredient we need is *inversion*, a principle for deducing which inference rules may have been used to prove a fact.

For instance, at one point in the proof, we need to draw a conclusion from a premise $s \in \mathcal{F}_0$, meaning that s is an initial state. By inversion, because set $\mathcal{F}_0$ is defined by a single inference rule, that rule must have been used to conclude the premise, so it must be that $s = \mathsf{WithAccumulator}(n_0, 1)$.

Similarly, at another point in the proof, we must reason from a premise $s \rightarrow s'$. The relation $\rightarrow$ is defined by two inference rules, so inversion leads us to two cases to consider. In the first case, corresponding to the first rule, $s = \mathsf{WithAccumulator}(0, a)$ and $s' = \mathsf{AnswerIs}(a)$. In the second case, corresponding to the second rule, $s = \mathsf{WithAccumulator}(n + 1, a)$ and $s' = \mathsf{WithAccumulator}(n, a \times (n + 1))$. It's worth checking that these values of s and s' are read off directly from the rules.

Though a completely formal and exhaustive treatment of inversion is beyond the scope of this text, generally it follows standard intuitions about "reverse-engineering" a set of rules that could have been used to derive some premise.

Another important property of invariants formalizes the connection with weakening an induction hypothesis.

THEOREM 5.6. *If I is an invariant of a transition system, then $I' \supseteq I$ (a superset of the original) is also an invariant of the same system.*

Note that the larger I' above may not be suitable to use in an inductive proof by Theorem 5.5! For instance, for factorial, we might define $I' = \{\mathsf{Answerls}(n_0!)\} \cup \{\mathsf{WithAccumulator}(n, a) \mid n, a \in \mathbb{N}\}$, clearly a superset of I. However, by forgetting everything that we know about intermediate WithAccumulator states, we will get stuck on the inductive step of the proof. Thus, what we call invariants here needn't also be *inductive invariants*, and there may be slight terminology mismatches with other sources.

Combining Theorems 5.5 and 5.6, it is now easy to prove Theorem 5.3, establishing the correctness of our particular factorial system $\mathcal{F}$. First, we use Theorem 5.5 to deduce that I is an invariant of $\mathcal{F}$. Then, we choose the very same I' that we warned above is not an inductive invariant, but which is fairly easily shown to be a superset of I. Therefore, by Theorem 5.6, I' is also an invariant of $\mathcal{F}$, and Theorem 5.3 follows quite directly from that fact, as I' is essentially a restatement of Theorem 5.3.

5.3. Rule Induction

Another crucial reasoning technique was hidden within the elided proof of Theorem 5.5. That technique is *rule induction*, which generalizes inversion just as normal structural induction generalizes case analysis. As an example, consider again the definition of transitive-reflexive closure by inference rules.

$$\frac{}{s \to^* s} \qquad \frac{s \to s' \quad s' \to^* s''}{s \to^* s''}$$

The relation $\to^*$ is a subset of $S \times S$. Imagine that we want to prove that some relation P holds of all pairs of states, where the first can reach the second. That is, we want to prove $\forall s, s'.\ (s \to^* s') \Rightarrow P(s, s')$, where $\Rightarrow$ is logical implication. We can actually derive a suitable induction principle, in the same way that we produced structural induction principles from definitions of inductive datatypes. We modify each defining rule of $\to^*$, replacing its conclusion with a use of P and adding a P induction hypothesis for each recursive premise.

$$\frac{}{P(s, s)} \qquad \frac{s \to s' \quad s' \to^* s'' \quad P(s', s'')}{P(s, s'')}$$

As before, where the defining rules of $\to^*$ show us how to *conclude* facts, the two new rules here are *proof obligations*. To apply rule induction and establish P for all reachability pairs, we must prove that each new rule is correct, as a kind of quantified implication.

As a simpler example than the invariant-induction theorem, consider transitivity for reachability.

THEOREM 5.7. *If $s \to^* s'$ and $s' \to^* s''$, then $s \to^* s''$.*

PROOF. By rule induction on the derivation of $s \to^* s'$, taking $P(s_1, s_2)$ to be that, if $s_2 = s'$, then $s_1 \to^* s''$. We consider variables s' and s'' fixed throughout the induction, along with their associated premise $s' \to^* s''$.

Base case: We must show $P(s, s)$ for an arbitrary s. Given that (based on the definition of P) we may assume $s = s'$, our premise $s' \to^* s''$ precisely matches the desired conclusion $s \to^* s''$.

Induction step: Assume $s \to s_1$, $s_1 \to^* s'$, and $P(s_1, s')$. We may apply the second rule defining $\to^*$, whose two premises become $s \to s_1$ and $s_1 \to^* s''$. The first is one of the available premises of the induction step. The second follows by the induction hypothesis about P. □

This sort of proof really is easier to follow in Coq code, so we especially encourage the reader to consult the mechanized version here!

In general, any inductive definition of a predicate, via a set of inference rules, implies a rule-induction principle. We will meet many such definitions throughout the book, and we will apply rule induction to most of them. It is valuable to understand basically how the rule-induction principle of a definition is read off from its original rules, but it is also true that Coq comes up with these principles automatically.

5.4. An Example with a Concurrent Program

Imagine that we want to verify a multithreaded, shared-memory program where multiple threads run this code at once.

```
f() {
  lock();
  local = global;
  global = local + 1;
  unlock();
}
```

Consider `global` as a variable shared across all threads, while each thread has its own version of variable `local`. The meaning of `lock()` and `unlock()` is as usual, where at most one thread can hold the lock at once, claiming it via `lock()` and relinquishing it via `unlock()`. When variable `global` is initialized to 0 and n threads run this code at once and all terminate, we expect that `global` finishes with value n. Of course, bugs in this program, like forgetting to include the locking, could lead to all sorts of wrong answers, with any value between 1 and n possible with the right demonic thread interleaving.

Encoding

To prove that we got the program right, let's formalize it as a transition system. First, our state set:

$$\text{States} \quad P \quad ::= \quad \mathsf{Lock} \mid \mathsf{Read} \mid \mathsf{Write}(n) \mid \mathsf{Unlock} \mid \mathsf{Done}$$

Compared to the last example, here we see more clearly that kinds of states correspond to *program counters* in the imperative code. The first four state kinds respectively mean that the program counter is right before the matching line in the program's code. The last state kind means the program counter is past the end of the function. Only Write states carry extra information, in this case the value of variable `local`. At every other program counter, we can prove that the value of variable `local` has no effect on further transitions, so we don't bother to store it. We will account for the value of variable `global` separately, in a way to be described shortly.

In particular, we will define a transition system for a single thread as $\mathcal{L} = \langle (\mathbb{N} \times \mathbb{B}) \times P, \mathcal{L}_0, \to_{\mathcal{L}} \rangle$. We define the state to include not only the thread-local state P but also the value of `global` (in $\mathbb{N}$) and whether the lock is currently taken (in $\mathbb{B}$, the Booleans, with values $\top$ [true] and $\bot$ [false]). There is one designated initial state.

$$\frac{}{((0, \bot), \mathsf{Lock}) \in \mathcal{L}_0}$$

Four inference rules explain the four transitions between program counters that a single thread can make, reading and writing shared state as needed.

$$\frac{}{((g, \bot), \mathsf{Lock}) \to_{\mathcal{L}} ((g, \top), \mathsf{Read})} \quad \frac{}{((g, \ell), \mathsf{Read}) \to_{\mathcal{L}} ((g, \ell), \mathsf{Write}(g))}$$

$$\frac{}{((g, \ell), \mathsf{Write}(n)) \to_{\mathcal{L}} ((n+1, \ell), \mathsf{Unlock})} \quad \frac{}{((g, \ell), \mathsf{Unlock}) \to_{\mathcal{L}} ((g, \bot), \mathsf{Done})}$$

Note that these rules will allow a thread to read and write the shared state even without holding the lock. The rules also allow any thread to unlock the lock, with no consideration for whether that thread must be the current lock holder. We must use an invariant-based proof to show that there are, in fact, no lurking violations of the lock-based concurrency discipline.

Of course, with just a single thread running, there aren't any interesting violations! However, we have been careful to describe system $\mathcal{L}$ in a generic way, with its state a pair of shared and private components. We can define a generic notion of a multithreaded system, with two systems that share some state and maintain their own private state.

Encoding

DEFINITION 5.8. Let $T^1 = \langle S \times P^1, S_0 \times P_0^1, \to^1 \rangle$ and $T^2 = \langle S \times P^2, S_0 \times P_0^2, \to^2 \rangle$ be two transition systems, with a shared-state type S in common between their state sets, also agreeing on the initial values S_0 for that shared state. We define the *parallel composition* $T^1 \mid T^2$ as $\langle S \times (P^1 \times P^2), S_0 \times (P_0^1 \times P_0^2), \to \rangle$, defining new transition relation $\to$ with the following inference rules, which capture the usual notion of thread interleaving.

$$\frac{(s, p_1) \to^1 (s', p_1')}{(s, (p_1, p_2)) \to (s', (p_1', p_2))} \quad \frac{(s, p_2) \to^2 (s', p_2')}{(s, (p_1, p_2)) \to (s', (p_1, p_2'))}$$

Note that the operator $\mid$ is carefully defined so that its output is suitable as input to a further instance of itself. As a result, while $\mathcal{L} \mid \mathcal{L}$ is a transition system modeling two threads running the code from above, we also have $\mathcal{L} \mid (\mathcal{L} \mid \mathcal{L})$ as a three-thread system based on that code, $(\mathcal{L} \mid \mathcal{L}) \mid (\mathcal{L} \mid \mathcal{L})$ as a four-thread system based on that code, etc.

Also note that $\mid$ constructs transition systems with our first examples of *nondeterminism* in transition relations. That is, given a particular starting state, there are multiple different places it may wind up after a given number of execution steps. In general, with thread-interleaving concurrency, the set of possible final states grows exponentially in the number of steps, a fact that torments concurrent-software testers to no end! Rather than consider all possible runs of the program, we will use an invariant to tame the complexity.

First, we should be clear on what we mean to prove about this program. Let's also restrict our attention to the two-thread case for the rest of this section; the n-thread case is left as an exercise for the reader!

THEOREM 5.9. *For any reachable state* $((g, \ell), (p^1, p^2))$ *of* $\mathcal{L} \mid \mathcal{L}$*, if* $p^1 = p^2 = \mathsf{Done}$*, then* $g = 2$*.*

That is, when both threads terminate, `global` equals 2.

As a first step toward an invariant, define function $\mathcal{C}$ from private states to numbers, capturing the *contribution of* a thread with that state, summarizing how much that thread has added to `globals`.

$$\mathcal{C}(p) \quad = \quad \begin{cases} 1 & p \in \{\mathsf{Unlock}, \mathsf{Done}\} \\ 0 & \text{otherwise} \end{cases}$$

Next, we define a function that, given a thread's private state, determines whether that thread *holds the lock*.

$$\mathcal{H}(p) \quad = \quad \begin{cases} \bot & p \in \{\mathsf{Lock}, \mathsf{Done}\} \\ \top & \text{otherwise} \end{cases}$$

Now, the main insight: we can reconstruct the shared state uniquely from the two private states! Function $\mathcal{S}$ does exactly that.

$$\mathcal{S}(p^1, p^2) \quad = \quad (\mathcal{H}(p^1) \vee \mathcal{H}(p^2), \mathcal{C}(p^1) + \mathcal{C}(p^2))$$

One last ingredient will help us write the invariant: a predicate $\mathcal{O}(p, p')$ capturing when, given the state p of one thread, the state p' is compatible with all of the implications of p's state, primarily in terms of mutual exclusion for the lock.

$$\mathcal{O}(p, p') \quad = \quad \begin{cases} \top & p \in \{\mathsf{Lock}, \mathsf{Done}\} \\ \neg\mathcal{H}(p') & p \in \{\mathsf{Read}, \mathsf{Unlock}\} \\ \neg\mathcal{H}(p') \wedge n = \mathcal{C}(p') & p = \mathsf{Write}(n) \end{cases}$$

Invariants

Finally, we can write the invariant.

$$I(s, (p^1, p^2)) \quad = \quad \mathcal{O}(p^1, p^2) \wedge \mathcal{O}(p^2, p^1) \wedge s = \mathcal{S}(p^1, p^2)$$

As is often the case, defining the invariant is the hard part of the proof, and the rest follows by the standard methodology that we used for factorial. To recap that method, first we use Theorem 5.5 to show that I really is an invariant of $\mathcal{L} \mid \mathcal{L}$. Next, we use Theorem 5.6 to show that I implies the original property of interest, that finished program states have value 2 for `global`. Most of the action is in the first step, where we must work through fussy details of all the different steps that could happen from a state within the invariant, using arithmetic reasoning in each case to either derive a contradiction (that step couldn't happen from this starting state) or show that a specific new state also belongs to the invariant. We leave those details to the Coq code, as usual.

The reader may be worried at this point that coming up with invariants can be rather tedious! In the next chapter, we meet a technique for finding invariants automatically, in some limited but important circumstances.

CHAPTER 6

Model Checking

Our analyses so far have been tedious for at least two different reasons. First, we've hand-crafted definitions of transition systems, rather than just writing programs in conventional programming languages. The next chapter will clear that obstacle, by introducing operational semantics, for building transition systems automatically from programs. The other inconvenience we've faced is defining invariants manually. There isn't a silver bullet to get us out of this duty, when working with Turing-complete languages, where almost all interesting questions, this one included, are undecidable. However, when we can phrase problems in terms of transition systems with *finitely many reachable states*, we can construct invariants automatically by *exhaustive exploration of the state space*, an approach otherwise known as *model checking*. Surprisingly many real programs can be reduced to finite state spaces, using the techniques introduced in this chapter. First, though, let's formalize our intuitions about exhaustive state-space exploration as a sound way to find invariants.

6.1. Exhaustive Exploration

For an arbitrary binary relation R, we write R^n for the n-times self-composition of R. Formally, where id is the identity relation that only relates values to themselves, we have:

$$\begin{aligned} R^0 &= \mathsf{id} \\ R^{n+1} &= R \circ R^n \end{aligned}$$

For some set S and binary relation R, we also write $R(S)$ for the composition of R and S, namely $\{x \mid \exists y \in S.\ y\ R\ x\}$.

Which states of transition system $\langle S, S_0, \rightarrow \rangle$ are reachable after 0 steps? That would be precisely the initial states S_0, which we can also write as $\rightarrow^0(S_0)$.

Which states are reachable after exactly 1 step? That is $\rightarrow(S_0)$, or $\rightarrow^1(S_0)$.

How about 2, 3, and 4 steps? There we have $\rightarrow^2(S_0)$, $\rightarrow^3(S_0)$, and $\rightarrow^4(S_0)$.

It follows that the set of states reachable after n steps is:

$$\mathsf{reach}(n) = \bigcup_{i \leqslant n} \rightarrow^i(S_0)$$

This iteration process is not obviously executable yet, because, a priori, we seem to need to consider all possible n values, to characterize the state space fully. However, a crucial property allows us to terminate our search soundly under some conditions.

Invariants

THEOREM 6.1. *If* $\mathsf{reach}(n+1) = \mathsf{reach}(n)$ *for some* n*, then* $\mathsf{reach}(n)$ *is an invariant of the system.*

Here we call $\mathsf{reach}(n)$ a *fixed point* of the transition system, because it is closed under further exploration. To find a fixed point with a concrete system, we start with S_0. We repeatedly take the *single-step closure* corresponding to composition with $\rightarrow$. At each step, we check whether the expanded set is actually equal to the previous set. If so, our process of *multi-step closure* has terminated, and we have an invariant, by construction. Again, keep in mind that multi-step closure will not terminate for most transition systems, and there is an art to phrasing a problem in terms of systems where it *will* terminate.

6.2. Abstracting a Transition System

When analyzing an infinite-state system, it is not necessary to give up hope for model checking. For instance, consider this program.

```
int global = 0;

thread() {
 int local;

 while (true) {
   local = global;
   global = local + 2;
 }
}
```

If we assume infinite-precision integers, then the state space is infinite. Considering just the global variable, every even number is reachable, even if we only run a single thread. However, there is a high degree of regularity across this state space. In particular, those values really are all even. Consider this other program, which is hauntingly similar to the last one, in a way that we will make precise shortly.

```
bool global = true;

thread() {
  bool local;

  while (true) {
    local = global;
    global = local;
  }
}
```

We replaced every use of an integer with *a Boolean that is true iff the integer is even*. Notice that now the program has a finite state space, and model checking applies easily! We can formalize such a transformation via the general principle of *abstraction of a transition system*.

The key idea is that every state of the concrete system (with relatively many states) can be associated to one or more states of the abstract system (with relatively few states). We formalize this association via a *simulation relation* R, and we define what makes a choice of R sound, via a notion of *simulation* via a binary operator $\prec$, subscripted by R.

$$\frac{(\forall s \in S_0.\ \exists s' \in S_0'.\ s\ R\ s') \quad (\forall s, s', s_1.\ s\ R\ s' \wedge s \to s_1 \Rightarrow \exists s_1'.\ s' \to' s_1' \wedge s_1\ R\ s_1')}{\langle S, S_0, \to\rangle \prec_R \langle S', S_0', \to'\rangle}$$

The simpler condition is that every concrete initial state must be related to at least one abstract initial state. The second, more complex condition essentially says that every step in the concrete world must be matchable by some related step in the abstract world. A commuting diagram may express the second condition more clearly.

$$\begin{array}{ccc} s & \xrightarrow{\to} & s_1 \\ \big\downarrow R & & \big\downarrow R \\ s' & \xrightarrow{\exists \to'} & s_1' \end{array}$$

At an even higher intuitive level, what simulation says is that every execution of the concrete system may be matched, step for step, by an execution of the abstract system. The relation R explains the rules for which states match across systems. For our purposes, the key pay-off from this connection is that we may translate any invariant of the abstract system into an invariant of the concrete system.

Abstraction

THEOREM 6.2. *If $\langle S, S_0, \to\rangle \prec_R \langle S', S_0', \to'\rangle$, and if I is an invariant of $\langle S', S_0', \to'\rangle$, then $R^{-1}(I)$ is an invariant of $\langle S, S_0, \to\rangle$.*

We can apply this theorem to the two example programs from earlier in the section, now imagining that we run two parallel-thread copies of each program, using last chapter's approach to modeling threads with transition systems. The concrete system can be represented with thread-local states $\{\mathsf{Read}\} \cup \{\mathsf{Write}(n) \mid n \in \mathbb{N}\}$ and the abstract system with $\{\mathsf{BRead}\} \cup \{\mathsf{BWrite}(b) \mid b \in \mathbb{B}\}$, for the Booleans $\mathbb{B}$. We define compatibility between local states.

$$\frac{}{\mathsf{Read} \sim \mathsf{BRead}} \qquad \frac{n \text{ even} \Leftrightarrow b = \mathsf{true}}{\mathsf{Write}(n) \sim \mathsf{BWrite}(b)}$$

We also define the overall state simulation relation R, which also covers state shared by threads.

$$\frac{(n \text{ even} \Leftrightarrow b = \mathsf{true}) \quad \ell_1 \sim \ell_1' \quad \ell_2 \sim \ell_2'}{(n, (\ell_1, \ell_2))\ R\ (b, (\ell_1', \ell_2'))}$$

By proving that R is truly a simulation relation, we reduce the problem to finding an invariant for the abstract system, which is easy to do with model checking.

One crucial consequence of abstraction-by-simulation deserves mentioning: We show that every concrete execution is matched abstractly, but there may also be additional abstract executions that don't match any concrete ones. In model checking the abstract system, we may do extra work to handle these "useless" paths! If we do manage to handle them all, then Theorem 6.2 applies perfectly well. However, we should be careful, in our choices of abstractions, to bias our designs toward those that don't introduce extra complexities.

6.3. Modular Decomposition of Invariant-Finding

Many transition systems are straightforward to abstract into others, as single global steps. Other times, the right way to tame a complex system is to decompose

it into others and analyze them separately for invariants. In such cases, the key is a proof principle to combine the invariants of the component systems into an invariant of the overall system. We will refer to this style of proof decomposition as *modularity*, and this section gives our first example of modularity, for multithreaded systems.

Imagine that we have a system consisting of n different copies of a transition system $\langle S, S_0, \rightarrow \rangle$ running as concurrent threads, modeled in the way introduced in the previous chapter. It's not obvious that we can analyze each thread separately, since, during that thread's execution, the other threads are constantly interrupting and modifying global state. To make matters worse, we can only understand their patterns of state modification by analyzing their thread-local state. The situation seems inherently unmodular.

However, consider the following construction on transition systems. Given a transition relation $\rightarrow$ and an invariant I on the global state shared by all threads, we define a new transition relation $\rightarrow^I$ as follows.

$$\frac{s \rightarrow s'}{s \rightarrow^I s'} \qquad \frac{I(g')}{(g, \ell) \rightarrow^I (g', \ell)}$$

The first rule says that any step of the original relation is also a step of the new relation. However, the second rule adds a new kind of step: the global state may change *arbitrarily*, so long as the new value satisfies invariant I. We lift this operation to full transition systems, defining $\langle S, S_0, \rightarrow \rangle^I = \langle S, S_0, \rightarrow^I \rangle$.

Abstraction

This construction trivially acts as an abstraction.

THEOREM 6.3. *$\mathbb{S} \prec_{\mathsf{id}} \mathbb{S}^I$, for any system $\mathbb{S}$ and property I.*

However, we wouldn't want to make this abstraction step in a proof about a single thread. We needlessly complicate our model checking by forcing ourselves to consider all modifications of the global state that obey I. The payoff comes in analyzing multithreaded systems.

Modularity

THEOREM 6.4. *Where I is an invariant over only the shared state of a multithreaded system, let $I' = \{(g, \ell) \mid I(g)\}$ be the lifting of I to cover full states, local parts included. If I' is an invariant for both $\mathbb{S}_1^I$ and $\mathbb{S}_2^I$, then I' is also an invariant for $(\mathbb{S}_1 \mid \mathbb{S}_2)^I$.*

This theorem gives us a way to analyze the threads in a system separately. As an example, consider this program, where multiple threads will run `f()` simultaneously.

```
int global = 0;

f() {
  int local = 0;

  while (true) {
    local = global;
    local = 3 + local;
    local = 7 + local;
    global = local;
  }
}
```

Call the transition-system encoding of this code $\mathbb{S}$. We can apply the Boolean-for-evenness abstraction to model a single thread with finite state, but we are left needing to account for interference by other threads. However, we can apply Theorem 6.4 to analyze threads separately.

For instance, we want to show that "`global` is always even" is an invariant of $\mathbb{S} \mid \mathbb{S}$. By Theorem 6.3, we can switch to analyzing system $(\mathbb{S} \mid \mathbb{S})^I$, where I is the evenness invariant. By Theorem 6.4, we can switch to proving the same invariant separately for systems $\mathbb{S}^I$ and $\mathbb{S}^I$, which are, of course, the same system in this case. We apply the Boolean-for-evenness abstraction to this system, to get one with a finite state space, so we can check the invariant automatically by model checking. Following the chain of reasoning backward, we have proved the invariant for $\mathbb{S} \mid \mathbb{S}$.

Even better, that last proof includes the hardest steps that carry over to the proof for an arbitrary number of threads. Define an exponentially growing system of threads $\mathbb{S}^n$ by:

$$\begin{aligned} \mathbb{S}^0 &= \mathbb{S} \\ \mathbb{S}^{n+1} &= \mathbb{S}^n \mid \mathbb{S}^n \end{aligned}$$

THEOREM 6.5. *For any n, it is an invariant of $\mathbb{S}^n$ that the global variable is always even.*

PROOF. By induction on n, repeatedly using Theorem 6.4 to push the obligation down to the leaves of the tree of concurrent compositions, after applying Theorem 6.3 at the start to introduce the use of $\ldots^I$. Every leaf is the same system $\mathbb{S}$, for which we abstract and apply model checking, appealing to the step above where we ran the same analysis. □

CHAPTER 7

Operational Semantics

It gets tedious to define a relation from first principles, to explain the behaviors of any concrete program. We do more things with programs than just reason about them. For instance, we compile them into other languages. To get the most mileage out of our correctness proofs, we should connect them to the same program syntax that we pass to compilers. *Operational semantics* is a family of techniques for automatically defining a transition system, or other relational characterization, from program syntax.

Throughout this chapter, we will demonstrate the different operational-semantics techniques on a single source language, defined like so.

$$\begin{array}{rrcl} \text{Numbers} & n & \in & \mathbb{N} \\ \text{Variables} & x & \in & \mathsf{Strings} \\ \text{Expressions} & e & ::= & n \mid x \mid e + e \mid e - e \mid e \times e \\ \text{Commands} & c & ::= & \mathsf{skip} \mid x \leftarrow e \mid c; c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \mathsf{while}\ e\ \mathsf{do}\ c \end{array}$$

7.1. Big-Step Semantics

Big-step operational semantics explains what it means to run a program to completion. For our example language, we define a relation written $(v, c) \Downarrow v'$, for "command c, run with variable valuation v, terminates, modifying the valuation to v'."

This relation is fairly straightforward to define with inference rules.

Encoding

$$\frac{}{(v, \mathsf{skip}) \Downarrow v} \quad \frac{}{(v, x \leftarrow e) \Downarrow v[x \mapsto [\![e]\!]v]} \quad \frac{(v, c_1) \Downarrow v_1 \quad (v_1, c_2) \Downarrow v_2}{(v, c_1; c_2) \Downarrow v_2}$$

$$\frac{[\![e]\!]v \neq 0 \quad (v, c_1) \Downarrow v'}{(v, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \Downarrow v'} \quad \frac{[\![e]\!]v = 0 \quad (v, c_2) \Downarrow v'}{(v, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \Downarrow v'}$$

$$\frac{[\![e]\!]v \neq 0 \quad (v, c_1) \Downarrow v_1 \quad (v_1, \mathsf{while}\ e\ \mathsf{do}\ c_1) \Downarrow v_2}{(v, \mathsf{while}\ e\ \mathsf{do}\ c_1) \Downarrow v_2} \quad \frac{[\![e]\!]v = 0}{(v, \mathsf{while}\ e\ \mathsf{do}\ c_1) \Downarrow v}$$

Notice how the definition is quite similar to a recursive interpreter written in a high-level programming language, though we write with the language of relations instead of functional programming. For instance, consider the simple case of the rule for sequencing, ";". We first "call the interpreter" on the first subcommand c_1 with the original valuation v. The result of the "recursive call" is a new valuation v_1, which we then feed into another "recursive call" on c_2, whose result becomes the overall result.

Why write this interpreter relationally instead of as a functional program? The most relevant answer applies to situations like ours as users of Coq or even of informal mathematics, where we must be very careful that all of our recursive definitions are well-founded. The recursive version of this relation is clearly not

well-founded, as it would run forever on a nonterminating while loop. It is also easier to incorporate *nondeterminism* in the relational style, a possibility that we will return to at the end of the chapter.

The big-step semantics is easy to apply to concrete programs. For instance, define `factorial` as the program `output` $\leftarrow 1$; while `input` do (`output` $\leftarrow$ `output` $\times$ `input`; `input` $\leftarrow$ `input` $- 1$).

THEOREM 7.1. *There exists v such that* $(\bullet[\texttt{input} \mapsto 2], \texttt{factorial}) \Downarrow v$ *and* $v(\texttt{output}) = 2$.

PROOF. By repeated application of the big-step inference rules. □

We can even prove that `factorial` behaves correctly on all inputs, by way of a lemma about `factorial_loop` defined as while `input` do (`output` $\leftarrow$ `output` $\times$ `input`; `input` $\leftarrow$ `input` $- 1$).

LEMMA 7.2. *If $v(\texttt{input}) = n$ and $v(\texttt{output}) = o$, then there exists v' such that $(v, \texttt{factorial\_loop}) \Downarrow v'$ and $v'(\texttt{output}) = n! \times o$.*

PROOF. By induction on n. □

LEMMA 7.3. *If $v(\texttt{input}) = n$, then there exists v' such that $(v, \texttt{factorial}) \Downarrow v'$ and $v'(\texttt{output}) = n!$.*

PROOF. Largely by direct appeal to Lemma 7.2. □

Most of our program proofs in this book establish *safety properties*, or invariants of transition systems. However, these last two examples with big-step semantics also establish program termination, taking us a few steps into the world of *liveness properties*.

7.2. Small-Step Semantics

Often it is convenient to break a system's execution into small sequential steps, rather than executing a whole program in one go. Perhaps the most compelling example comes from concurrency, where it is difficult to give a big-step semantics directly. Nonterminating programs are the other standard example. We want to be able to establish invariants for those programs, all the same, and we need a semantics to help us state what it means to be an invariant.

The canonical solution is *small-step operational semantics*, probably the most common approach to formal program semantics in contemporary research. Now we define a single-step relation $(v, c) \rightarrow (v', c')$, meaning that one execution step transforms the first state into the second state. Each state is a valuation v and a current command c.

Encoding

These inference rules give the details.

$$\frac{}{(v, x \leftarrow e) \rightarrow (v[x \mapsto [\![e]\!]v], \mathsf{skip})} \qquad \frac{(v, c_1) \rightarrow (v', c_1')}{(v, c_1; c_2) \rightarrow (v', c_1'; c_2)} \qquad \frac{}{(v, \mathsf{skip}; c_2) \rightarrow (v, c_2)}$$

$$\frac{[\![e]\!]v \neq 0}{(v, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \rightarrow (v, c_1)} \qquad \frac{[\![e]\!]v = 0}{(v, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \rightarrow (v, c_2)}$$

$$\frac{[\![e]\!]v \neq 0}{(v, \mathsf{while}\ e\ \mathsf{do}\ c_1) \rightarrow (v, c_1; \mathsf{while}\ e\ \mathsf{do}\ c_1)} \qquad \frac{[\![e]\!]v = 0}{(v, \mathsf{while}\ e\ \mathsf{do}\ c_1) \rightarrow (v, \mathsf{skip})}$$

The intuition behind the rules may come best from working out an example.

THEOREM 7.4. *There exists valuation v such that* $(\bullet[\texttt{input} \mapsto 2], \texttt{factorial}) \to^* (v, \mathsf{skip})$ *and* $v(\texttt{output}) = 2$.

PROOF. Here is a step-by-step (literally!) derivation that finds v.

$$
\begin{aligned}
& (\bullet[\texttt{input} \mapsto 2], \texttt{output} \leftarrow 1; \texttt{factorial\_loop}) \\
\to\ & (\bullet[\texttt{input} \mapsto 2][\texttt{output} \mapsto 1], \mathsf{skip}; \texttt{factorial\_loop}) \\
\to\ & (\bullet[\texttt{input} \mapsto 2][\texttt{output} \mapsto 1], \texttt{factorial\_loop}) \\
\to\ & (\bullet[\texttt{input} \mapsto 2][\texttt{output} \mapsto 1], (\texttt{output} \leftarrow \texttt{output} \times \texttt{input}; \texttt{input} \leftarrow \texttt{input} - 1); \texttt{factorial\_loop}) \\
\to\ & (\bullet[\texttt{input} \mapsto 2][\texttt{output} \mapsto 2], (\mathsf{skip}; \texttt{input} \leftarrow \texttt{input} - 1); \texttt{factorial\_loop}) \\
\to\ & (\bullet[\texttt{input} \mapsto 2][\texttt{output} \mapsto 2], \texttt{input} \leftarrow \texttt{input} - 1; \texttt{factorial\_loop}) \\
\to\ & (\bullet[\texttt{input} \mapsto 1][\texttt{output} \mapsto 2], \mathsf{skip}; \texttt{factorial\_loop}) \\
\to\ & (\bullet[\texttt{input} \mapsto 1][\texttt{output} \mapsto 2], \texttt{factorial\_loop}) \\
\to\ & (\bullet[\texttt{input} \mapsto 1][\texttt{output} \mapsto 2], (\texttt{output} \leftarrow \texttt{output} \times \texttt{input}; \texttt{input} \leftarrow \texttt{input} - 1); \texttt{factorial\_loop}) \\
\to\ & (\bullet[\texttt{input} \mapsto 1][\texttt{output} \mapsto 2], (\mathsf{skip}; \texttt{input} \leftarrow \texttt{input} - 1); \texttt{factorial\_loop}) \\
\to\ & (\bullet[\texttt{input} \mapsto 1][\texttt{output} \mapsto 2], \texttt{input} \leftarrow \texttt{input} - 1; \texttt{factorial\_loop}) \\
\to\ & (\bullet[\texttt{input} \mapsto 0][\texttt{output} \mapsto 2], \mathsf{skip}; \texttt{factorial\_loop}) \\
\to\ & (\bullet[\texttt{input} \mapsto 0][\texttt{output} \mapsto 2], \texttt{factorial\_loop}) \\
\to\ & (\bullet[\texttt{input} \mapsto 0][\texttt{output} \mapsto 2], \mathsf{skip})
\end{aligned}
$$

Clearly the final valuation assigns output to 2. □

7.2.1. Equivalence of Big-Step and Small-Step. Different theorems are easier to prove with different semantics, so it is helpful to establish formally the intuitive connection between big and small steps.

LEMMA 7.5. *If* $(v, c_1) \to^* (v', c_1')$, *then* $(v, c_1; c_2) \to^* (v', c_1'; c_2)$,

PROOF. By induction on the derivation of $(v, c_1) \to^* (v', c_1')$. □

THEOREM 7.6. *If* $(v, c) \Downarrow v'$, *then* $(v, c) \to^* (v', \mathsf{skip})$.

PROOF. By induction on the derivation of $(v, c) \Downarrow v'$, appealing to the last lemma at two points. □

LEMMA 7.7. *If* $(v, c) \to (v', c')$ *and* $(v', c') \Downarrow v''$, *then* $(v, c) \Downarrow v''$. *In other words, we can add a small step to the beginning of any big-step derivation.*

PROOF. By induction on the derivation of $(v, c) \to (v', c')$. □

LEMMA 7.8. *If* $(v, c) \to^* (v', c')$ *and* $(v', c') \Downarrow v''$, *then* $(v, c) \Downarrow v''$. *In other words, we can add any number of small steps to the beginning of any big-step derivation.*

PROOF. By induction on the derivation of $(v, c) \to^* (v', c')$, appealing to the last lemma. □

THEOREM 7.9. *If* $(v, c) \to^* (v', \mathsf{skip})$, *then* $(v, c) \Downarrow v'$.

PROOF. Largely by appeal to the last lemma, considering that $(v', \mathsf{skip}) \Downarrow v'$. □

7.2.2. Transition Systems from Small-Step Semantics. The small-step semantics is a natural fit with our working definition of transition systems. We can define a transition system from any valuation and command, where $\mathbb{V}$ is the set of valuations and $\mathbb{C}$ the set of commands, by $\mathbb{T}(v, c) = \langle \mathbb{V} \times \mathbb{C}, \{(v, c)\}, \rightarrow \rangle$. Now we bring to bear all of our machinery about invariants and their proof methods.

Invariants

For instance, consider program $P = \mathsf{while}\ \mathtt{n}\ \mathsf{do}\ \mathtt{a} \leftarrow \mathtt{a} + \mathtt{n}; \mathtt{n} \leftarrow \mathtt{n} - 2$.

THEOREM 7.10. *Given even n, for $\mathbb{T}(\bullet[\mathtt{n} \mapsto n][\mathtt{a} \mapsto 0], P)$, it is an invariant that the valuation maps variable* a *to an even number.*

PROOF. First, we strengthen the invariant. We compute the set $\overline{P}$ of all commands that can be reached from P by stepping the small-step semantics. This set is finite, even though the set of *reachable valuations* is infinite, considering all potential n values. Our strengthened invariant is $I(v, c) = c \in \overline{P} \wedge (\exists n.\ v(\mathtt{n}) = n \wedge \mathrm{even}(n)) \wedge (\exists a.\ v(\mathtt{a}) = a \wedge \mathrm{even}(a))$. In other words, we strengthen by adding the constraints that (1) we do not stray from the expected set of reachable commands and (2) variable n also remains even.

The strengthened invariant is straightforward to prove by invariant induction, using repeated inversion on $\rightarrow$ facts. □

7.3. Contextual Small-Step Semantics

The reader may have noticed some tedium in certain rules of the small-step semantics, like this one.

$$\frac{(v, c_1) \rightarrow (v', c_1')}{(v, c_1; c_2) \rightarrow (v', c_1'; c_2)}$$

This rule is an example of a *congruence rule*, which shows how to take a step and *lift* it into a step within a larger command, whose other subcommands are unaffected. Complex languages can require many congruence rules, and yet we feel like we should be able to avoid repeating all this boilerplate logic somehow. A common way to do so is switching to *contextual small-step semantics*.

We illustrate with our running example language. The first step is to define a set of *evaluation contexts*, which formalize the spots within a larger command where steps are enabled.

Encoding

$$\text{Evaluation contexts} \quad C \quad ::= \quad \Box \mid C; c$$

We define the operator of *plugging* an evaluation context in the natural way.

$$\begin{aligned} \Box[c] &= c \\ (C; c_2)[c] &= C[c]; c_2 \end{aligned}$$

For this language, the only interesting case of evaluation contexts is the one that allows us to *descend into the left subcommand*, because the old congruence rule invoked the step relation recursively for that position.

The next ingredient is a reduced set of basic step rules, where we have dropped the congruence rule.

$$\frac{}{(v, x \leftarrow e) \rightarrow_0 (v[x \mapsto [\![e]\!]v], \mathsf{skip})} \qquad \frac{}{(v, \mathsf{skip}; c_2) \rightarrow_0 (v, c_2)}$$

$$\frac{[\![e]\!]v \neq 0}{(v, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \rightarrow_0 (v, c_1)} \qquad \frac{[\![e]\!]v = 0}{(v, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \rightarrow_0 (v, c_2)}$$

$$\frac{[\![e]\!]v \neq 0}{(v, \mathsf{while}\ e\ \mathsf{do}\ c_1) \to_0 (v, c_1; \mathsf{while}\ e\ \mathsf{do}\ c_1)} \quad \frac{[\![e]\!]v = 0}{(v, \mathsf{while}\ e\ \mathsf{do}\ c_1) \to_0 (v, \mathsf{skip})}$$

We regain the full coverage of the original rules with a new relation $\to_{\mathsf{c}}$, saying that we may apply $\to_0$ at the active subcommand within a larger command.

$$\frac{(v, c) \to_0 (v', c')}{(v, C[c]) \to_{\mathsf{c}} (v', C[c'])}$$

Let's revisit last section's example, to see contextual semantics in action, especially to demonstrate how to express an arbitrary command as an evaluation context plugged with another command.

THEOREM 7.11. *There exists valuation v such that* $(\bullet[\mathtt{input} \mapsto 2], \mathtt{factorial}) \to_{\mathsf{c}}^* (v, \mathsf{skip})$ *and* $v(\mathtt{output}) = 2$.

PROOF.

$$\begin{array}{rl}
 & (\bullet[\mathtt{input} \mapsto 2], \mathtt{output} \leftarrow 1; \mathtt{factorial_loop}) \\
= & (\bullet[\mathtt{input} \mapsto 2], (\square; \mathtt{factorial_loop})[\mathtt{output} \leftarrow 1]) \\
\to_{\mathsf{c}} & (\bullet[\mathtt{input} \mapsto 2][\mathtt{output} \mapsto 1], \mathsf{skip}; \mathtt{factorial_loop}) \\
= & (\bullet[\mathtt{input} \mapsto 2][\mathtt{output} \mapsto 1], \square[\mathsf{skip}; \mathtt{factorial_loop}]) \\
\to_{\mathsf{c}} & (\bullet[\mathtt{input} \mapsto 2][\mathtt{output} \mapsto 1], \mathtt{factorial_loop}) \\
= & (\bullet[\mathtt{input} \mapsto 2][\mathtt{output} \mapsto 1], \square[\mathtt{factorial_loop}]) \\
\to_{\mathsf{c}} & (\bullet[\mathtt{input} \mapsto 2][\mathtt{output} \mapsto 1], (\mathtt{output} \leftarrow \mathtt{output} \times \mathtt{input}; \mathtt{input} \leftarrow \mathtt{input} - 1); \mathtt{factorial_loop}) \\
= & (\bullet[\mathtt{input} \mapsto 2][\mathtt{output} \mapsto 1], ((\square; \mathtt{input} \leftarrow \mathtt{input} - 1); \mathtt{factorial_loop})[\mathtt{output} \leftarrow \mathtt{output} \times \mathtt{input}]) \\
\to_{\mathsf{c}} & (\bullet[\mathtt{input} \mapsto 2][\mathtt{output} \mapsto 2], (\mathsf{skip}; \mathtt{input} \leftarrow \mathtt{input} - 1); \mathtt{factorial_loop}) \\
= & (\bullet[\mathtt{input} \mapsto 2][\mathtt{output} \mapsto 2], (\square; \mathtt{factorial_loop})[\mathsf{skip}; \mathtt{input} \leftarrow \mathtt{input} - 1)] \\
\to_{\mathsf{c}} & (\bullet[\mathtt{input} \mapsto 2][\mathtt{output} \mapsto 2], \mathtt{input} \leftarrow \mathtt{input} - 1; \mathtt{factorial_loop}) \\
= & (\bullet[\mathtt{input} \mapsto 2][\mathtt{output} \mapsto 2], (\square; \mathtt{factorial_loop})[\mathtt{input} \leftarrow \mathtt{input} - 1]) \\
\to_{\mathsf{c}} & (\bullet[\mathtt{input} \mapsto 1][\mathtt{output} \mapsto 2], \mathsf{skip}; \mathtt{factorial_loop}) \\
= & (\bullet[\mathtt{input} \mapsto 1][\mathtt{output} \mapsto 2], \square[\mathsf{skip}; \mathtt{factorial_loop}]) \\
\to_{\mathsf{c}}^* & \ldots \\
\to_{\mathsf{c}} & (\bullet[\mathtt{input} \mapsto 0][\mathtt{output} \mapsto 2], \mathsf{skip})
\end{array}$$

Clearly the final valuation assigns `output` to 2. □

7.3.1. Equivalence of Small-Step, With and Without Evaluation Contexts. This new semantics formulation is equivalent to the other two, as we establish now.

THEOREM 7.12. *If* $(v, c) \to (v', c')$, *then* $(v, c) \to_{\mathsf{c}} (v', c')$.

PROOF. By induction on the derivation of $(v, c) \to (v', c')$. □

LEMMA 7.13. *If* $(v, c) \to_0 (v', c')$, *then* $(v, c) \to (v', c')$.

PROOF. By cases on the derivation of $(v, c) \to_0 (v', c')$. □

LEMMA 7.14. *If* $(v, c) \to_0 (v', c')$, *then* $(v, C[c]) \to (v', C[c'])$.

PROOF. By induction on the structure of evaluation context C, appealing to the last lemma. □

THEOREM 7.15. *If* $(v, c) \to_{\mathsf{c}} (v', c')$, *then* $(v, c) \to (v', c')$.

PROOF. By inversion on the derivation of $(v, c) \to_{\mathsf{c}} (v', c')$, followed by an appeal to the last lemma. □

7.3.2. Evaluation Contexts Pay Off: Adding Concurrency. To showcase the convenience of contextual semantics, let's extend our example language with a simple construct for running two commands in parallel, implicitly extending the definition of plugging accordingly.

$$\text{Commands} \quad c \quad ::= \quad \ldots \mid c||c$$

Encoding

To capture the idea that *either* command in a parallel construct is allowed to step next, we extend evaluation contexts like so:

$$\text{Evaluation contexts} \quad C \quad ::= \quad \ldots \mid C||c \mid c||C$$

We need one more basic step rule, to "garbage-collect" threads that have finished.

$$\frac{}{(v, \mathsf{skip}||c) \to_0 (v, c)}$$

And that's it! The new system faithfully captures our usual idea of threads executing in parallel. All of the theorems proved previously about contextual steps continue to hold. In fact, in the accompanying Coq code, literally the same proof scripts establish the new versions of the theorems, with no new human proof effort. It's not often that concurrency comes for free in a rigorous proof!

7.4. Determinism

Our last extension with parallelism introduced intentional nondeterminism in the semantics: a single starting state can step to multiple different next states. However, the three semantics for the original language are deterministic, and we can prove it.

THEOREM 7.16. *If* $(v, c) \Downarrow v_1$ *and* $(v, c) \Downarrow v_2$, *then* $v_1 = v_2$.

PROOF. By induction on the derivation of $(v, c) \Downarrow v_1$ and inversion on the derivation of $(v, c) \Downarrow v_2$. □

THEOREM 7.17. *If* $(v, c) \to (v_1, c_1)$ *and* $(v, c) \to (v_2, c_2)$, *then* $v_1 = v_2$ *and* $c_1 = c_2$.

PROOF. By induction on the derivation of $(v, c) \to (v_1, c_1)$ and inversion on the derivation of $(v, c) \to (v_2, c_2)$. □

THEOREM 7.18. *If* $(v, c) \to_{\mathsf{c}} (v_1, c_1)$ *and* $(v, c) \to_{\mathsf{c}} (v_2, c_2)$, *then* $v_1 = v_2$ *and* $c_1 = c_2$.

PROOF. Follows from the last theorem and the equivalence we proved between $\to$ and $\to_{\mathsf{c}}$. □

We'll stop, for now, in our tour of useful properties of operational semantics. All of the rest of the book is based on small-step semantics, with or without evaluation contexts. As we study new kinds of programming languages, we will see how to model them operationally. Almost every new proof technique is phrased as an approach to establishing invariants of transition systems based on small-step semantics.

CHAPTER 8

Abstract Interpretation and Dataflow Analysis

The last two chapters showed us both how to build a transition system from a program automatically and how to find an invariant for a transition system automatically. Let's now combine these ideas to find invariants for programs automatically, in a particular way associated with the technique of *dataflow analysis* used to drive many compiler optimizations. Throughout, we'll stick with the example of the small imperative language whose semantics we studied in the last chapter. We'll confine our attention to its basic small-step semantics via the $\rightarrow$ relation.

Model checking builds up increasingly larger finite sets of reachable states in a system. A state (v, c) of our imperative language combines *control state* c (the next command to execute) with *data state* v (the values of the variables), and so model checking will find invariants that restrict both components. We say that model checking is *path-sensitive* because its invariants can distinguish between the different data states that can be associated with the same control state, reached along different paths in the program's executions. Path-sensitive analyses tend to be much more computationally expensive than *path-insensitive* analyses, whose invariants collapse together all ways of reaching the same control state. Dataflow analysis is one such path-insensitive approach, and its underlying theory is *abstract interpretation*.

8.1. Definition of an Abstract Interpretation

An abstract interpretation is a particular sort of abstraction, of the kind we met in studying model checking. In that more general setting, we can represent concrete states with any sorts of abstract states. In abstract interpretation, we most commonly associate each variable with an independent abstract description. One example, which we'll formalize in more detail shortly, would be to label each variable as "even," "odd," or "either."

Definition 8.1. An *abstract interpretation* (for our example imperative language) is a tuple $\langle \mathbb{D}, \top, \mathcal{C}, \hat{+}, \hat{-}, \hat{\times}, \sqcup, \sim \rangle$, where $\mathbb{D}$ is a set (the domain of the analysis); $\top \in \mathbb{D}$; $\mathcal{C} : \mathbb{N} \rightarrow \mathbb{D}$; $\hat{+}, \hat{-}, \hat{\times}, \sqcup : \mathbb{D} \times \mathbb{D} \rightarrow \mathbb{D}$; and $\sim \subseteq \mathbb{N} \times \mathbb{D}$. The idea is that:

- Abstract versions of numbers are $\mathbb{D}$ values.
- $\top$ ("top") is the least specific abstract value, representing any concrete value.
- $\mathcal{C}$ maps any constant to its most precise abstraction.
- $\hat{+}$, $\hat{-}$, and $\hat{\times}$ push abstraction through arithmetic operators, calculating their most precise abstractions.
- $\sqcup$ ("join") computes the *least upper bound* of two abstract values: the most specific value that represents any value associated with either input.

- $\sim$ formalizes the idea of which concrete values are covered by which abstract values.

For $a, b \in \mathbb{D}$, define $a \sqsubseteq b$ to mean $\forall n \in \mathbb{N}.\ (n \sim a) \Rightarrow (n \sim b)$. That is, b is at least as general as a. An abstract interpretation must satisfy the following algebraic laws:

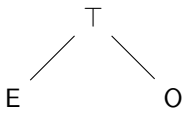
- $\forall a \in \mathbb{D}.\ a \sqsubseteq \top$
- $\forall n \in \mathbb{N}.\ n \sim \mathcal{C}(n)$
- $\forall n, m \in \mathbb{N}.\ \forall a, b \in \mathbb{D}.\ n \sim a \wedge m \sim b \Rightarrow (n + m) \sim (a \hat{+} b)$
- $\forall n, m \in \mathbb{N}.\ \forall a, b \in \mathbb{D}.\ n \sim a \wedge m \sim b \Rightarrow (n - m) \sim (a \hat{-} b)$
- $\forall n, m \in \mathbb{N}.\ \forall a, b \in \mathbb{D}.\ n \sim a \wedge m \sim b \Rightarrow (n \times m) \sim (a \hat{\times} b)$
- $\forall a, b, a', b' \in \mathbb{D}.\ a \sqsubseteq a' \wedge b \sqsubseteq b' \Rightarrow (a \hat{+} b) \sqsubseteq (a' \hat{+} b')$
- $\forall a, b, a', b' \in \mathbb{D}.\ a \sqsubseteq a' \wedge b \sqsubseteq b' \Rightarrow (a \hat{-} b) \sqsubseteq (a' \hat{-} b')$
- $\forall a, b, a', b' \in \mathbb{D}.\ a \sqsubseteq a' \wedge b \sqsubseteq b' \Rightarrow (a \hat{\times} b) \sqsubseteq (a' \hat{\times} b')$
- $\forall a, b \in \mathbb{D}.\ a \sqsubseteq (a \sqcup b)$
- $\forall a, b \in \mathbb{D}.\ b \sqsubseteq (a \sqcup b)$

As an example, consider this formalization of even-odd analysis, whose proof of soundness is left as an exercise for the reader. (While the treatment of subtraction may seem gratuitously imprecise, recall that we are working here with natural numbers and not integers, such that subtraction "sticks" at zero when the result would otherwise be negative.)

$$
\begin{array}{rcl}
\mathbb{D} & = & \{\mathsf{E}, \mathsf{O}, \top\} \\
\mathcal{C}(n) & = & \mathsf{E} \text{ or } \mathsf{O}, \text{ depending on parity of } n \\
\mathsf{E} \hat{+} \mathsf{E} & = & \mathsf{E} \\
\mathsf{E} \hat{+} \mathsf{O} & = & \mathsf{O} \\
\mathsf{O} \hat{+} \mathsf{E} & = & \mathsf{O} \\
\mathsf{O} \hat{+} \mathsf{O} & = & \mathsf{E} \\
\_ \hat{+} \_ & = & \top \\
\mathsf{E} \hat{-} \mathsf{E} & = & \mathsf{E} \\
\mathsf{O} \hat{-} \mathsf{O} & = & \mathsf{E} \\
\_ \hat{-} \_ & = & \top \\
\mathsf{E} \hat{\times} \_ & = & \mathsf{E} \\
\_ \hat{\times} \mathsf{E} & = & \mathsf{E} \\
\mathsf{O} \hat{\times} \mathsf{O} & = & \mathsf{O} \\
\_ \hat{\times} \_ & = & \top \\
\mathsf{E} \sqcup \mathsf{E} & = & \mathsf{E} \\
\mathsf{O} \sqcup \mathsf{O} & = & \mathsf{O} \\
\_ \sqcup \_ & = & \top \\
n \sim \mathsf{E} & = & n \text{ is even} \\
n \sim \mathsf{O} & = & n \text{ is odd} \\
n \sim \top & = & \text{always}
\end{array}
$$

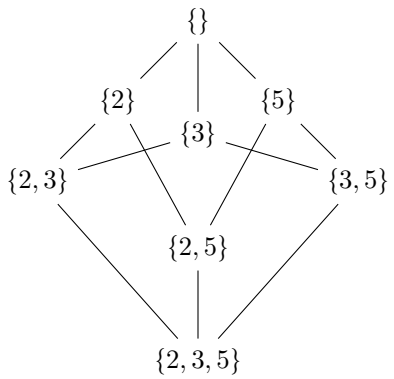
We generally think of an abstract interpretation as forming a *lattice* (actually a semilattice), which is roughly the algebraic structure characterized by operations

like $\sqcup$, when $\sqcup$ truly returns the *most specific* or *least* upper bound of its two arguments. We visualize the even-odd lattice like so.



The idea is that taking the join of two elements moves us *up* the lattice to their lowest common ancestor.

An edge going up from a to b indicates that $a \sqsubseteq b$. As another example, consider a lattice tracking prime factors of numbers, up to 5. Then the picture version might go like so:



Since $\sqsubseteq$ is clearly transitive, upward-moving paths across multiple nodes also imply $\sqsubseteq$ relationships between their endpoints. It's worth verifying quickly that any two nodes in this graph have a unique lowest common ancestor, which is the proper result of the $\sqcup$ operation on those nodes.

Another worthwhile exercise for the reader is to work out the proper definitions of $\hat{+}$, $\hat{-}$, and $\hat{\times}$ for this domain.

8.2. Flow-Insensitive Analysis

We now give our first recipe for building a program abstraction from an abstract interpretation. We apply a *flow-insensitive* abstraction, which means we find an invariant that doesn't depend at all on the control part c of a full state (v, c). Alternatively, the invariant depends only on the data part v. Concretely, with $\mathbb{V}$ the set of variables, we work with states $s \in \mathbb{V} \to \mathbb{D}$, taking the domain $\mathbb{D}$ of our chosen abstract interpretation. An abstract state s for a concrete valuation v assigns to each x an abstract value $s(x)$ such that $v(x) \sim s(x)$. We overload the operator $\sim$ to denote this compatibility via $v \sim s$.

As a preliminary, we define the abstract interpretation of an expression like so:

$$\begin{aligned}
[n]s &= \mathcal{C}(n) \\
[x]s &= s(x) \\
[e_1 + e_2]s &= [e_1]s \hat{+} [e_2]s \\
[e_1 - e_2]s &= [e_1]s \hat{-} [e_2]s \\
[e_1 \times e_2]s &= [e_1]s \hat{\times} [e_2]s
\end{aligned}$$

THEOREM 8.2. *If $v \sim s$, then $[\![e]\!]v \sim [e]s$.*

Next, we model the possible effects of commands. We already said that our flow-insensitive analysis will forget about control flow in a command, but what does that mean formally? States of this language, without control flow taken into account, are just variable valuations, and the only way a command can affect a valuation is through executing assignments. Therefore, forgetting the control flow of a command amounts to just *recording which assignments it contains syntactically*, losing all context about which Boolean tests would need to pass to reach each assignment. This simple syntactic extraction process can be formalized with an assignments-of function $\mathcal{A}$ for commands.

$$\begin{aligned}
\mathcal{A}(\mathsf{skip}) &= \{\} \\
\mathcal{A}(x \leftarrow e) &= \{(x, e)\} \\
\mathcal{A}(c_1; c_2) &= \mathcal{A}(c_1) \cup \mathcal{A}(c_2) \\
\mathcal{A}(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) &= \mathcal{A}(c_1) \cup \mathcal{A}(c_2) \\
\mathcal{A}(\mathsf{while}\ e\ \mathsf{do}\ c_1) &= \mathcal{A}(c_1)
\end{aligned}$$

As a final preliminary ingredient, for abstract states s_1 and s_2, define $s_1 \sqcup s_2$ by $(s_1 \sqcup s_2)(x) = s_1(x) \sqcup s_2(x)$.

Now we define the flow-insensitive step relation, over abstract states alone, as:

$$\frac{}{s \to^c_{\mathsf{FI}} s} \qquad \frac{(x, e) \in \mathcal{A}(c)}{s \to^c_{\mathsf{FI}} s \sqcup s[x \mapsto [e]s]}$$

We can establish formally how forgetting about the order of assignments is a valid abstraction technique.

Abstraction

THEOREM 8.3. *Given command c, initial valuation v, and initial abstract state s such that $v \sim s$. The transition system with initial state s and step relation $\to^c_{\mathsf{FI}}$ simulates the system with initial state (v, c) and step relation $\to$, according to a simulation relation enforcing $\sim$ between the valuation and abstract state.*

Now a simple procedure can find an invariant for the abstracted system. In particular:

1. Initialize s with the abstract state from the theorem statement.
2. Compute $s' = s \sqcup \bigsqcup_{(x,e)\in\mathcal{A}(c)} s[x \mapsto [e]s]$.
3. If $s' \sqsubseteq s$, then we're done; s is the invariant.
4. Otherwise, assign $s = s'$ and return to 2.

Every step in this outline is computable, since the abstract states will always be finite maps.

Invariants

THEOREM 8.4. *If the outline above terminates, then it is an invariant of the flow-insensitive abstracted system that s (its final value from the loop above) is an upper bound for every reachable state. That is, for every reachable s', $s' \sqsubseteq s$.*

To check a concrete program, we first abstract it to a flow-insensitive version with Theorem 8.3, then we find a guaranteed invariant with Theorem 8.4. One wrinkle here is that it is not obvious that our informal loop above always terminates. However, it always terminates if our abstract domain has *finite height*, meaning that there is no infinite ascending chain of distinct elements a_i such that $a_i \sqsubseteq a_{i+1}$ for all i. Our even-odd example trivially has that property, since it contains only finitely many distinct elements.

It is worth emphasizing that, when those conditions are met, our invariant-finding procedure is guaranteed to terminate, even though the underlying language is Turing-complete, so that most interesting analysis problems are uncomputable! The catch is that it is always possible that the invariant found is a trivial one, where the abstract state maps every variable to $\top$.

Here is an example of a program where flow-insensitive even-odd analysis gives the most precise answer (relative to its simplifying assumption that we must assign the same description to a variable at every step of execution).

$$n \leftarrow 10; x \leftarrow 0; \mathsf{while}\ n > 0\ \mathsf{do}\ x \leftarrow x + 2 \times n; n \leftarrow n - 1$$

The abstract state we wind up with is $\bullet[n \mapsto \top][x \mapsto \mathsf{E}]$.

8.3. Flow-Sensitive Analysis

We can only go so far with flow-insensitive invariants, which don't let us record different facts about the variables for different lines of the program code. Such an analysis will get tripped up even by straightline code where parities of variables change as we go. Here is a trivial example program where the flow-insensitive analysis returns the useless answer $\bullet[x \mapsto \top]$, when the most precise answer (about program state after execution) would be $\bullet[x \mapsto \mathsf{O}]$.

$$x \leftarrow 0; x \leftarrow 1$$

The solution to this problem can be to go to *flow-sensitive* analysis, where an abstract state S is a finite map from commands (all the intermediate "program counters" of an original command) to the abstract states of the previous section.

We define a function $\mathcal{S}(s, c, f)$ to compute all of the states of the form (s', c') reachable in a single step from (s, c). Actually, for each (s', c') covered by that informal description, this function returns a map from keys $f(c')$ to values s'. The idea is that function f wraps the step in any additional command context that isn't participating directly in this step. See how f is modified in the sequencing case below, for something of an intuition for its purpose.

$$\begin{aligned}
\mathcal{S}(s, \mathsf{skip}, f) &= \bullet \\
\mathcal{S}(s, x \leftarrow e, f) &= \bullet[f(\mathsf{skip}) \mapsto s[x \mapsto [e]s]] \\
\mathcal{S}(s, \mathsf{skip}; c_2, f) &= \bullet[f(c_2) \mapsto s] \\
\mathcal{S}(s, c_1; c_2, f) &= \mathcal{S}(s, c_1, \lambda c.\ f(c; c_2)) \\
\mathcal{S}(s, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, f) &= \bullet[f(c_1) \mapsto s][f(c_2) \mapsto s] \\
\mathcal{S}(s, \mathsf{while}\ e\ \mathsf{do}\ c_1, f) &= \bullet[f(\mathsf{skip}) \mapsto s][f(c_1; \mathsf{while}\ e\ \mathsf{do}\ c_1) \mapsto s]
\end{aligned}$$

Note that the last two cases, for conditional control flow, ignore the test expression entirely, which is certainly sound, though it may lead to imprecision in the analysis. This approximation is known as *path insensitivity*. Define $\mathcal{S}(s, c)$ as shorthand for $\mathcal{S}(s, c, \lambda c_1.\ c_1)$.

Now we can define a new abstract step relation.

$$\frac{\mathcal{S}(s, c)(c') = s'}{(s, c) \rightarrow_{\mathsf{FS}} (s', c')}$$

That is, we step from (s, c) to (s', c') precisely when, if we look up c' in the result of running c abstractly in s, we find s'.

Now we can follow an analogous path to the one we did in the last section.

Abstraction

THEOREM 8.5. *Given command c and initial valuation v. The transition system with initial state (s, c) and step relation $\rightarrow_{\mathsf{FS}}$ simulates the system with initial state (v, c) and step relation $\rightarrow$, according to a simulation relation enforcing equality of the commands, as well as $\sim$ between the valuation and abstract state.*

Now another simple procedure can find an invariant for the abstracted system. We write $S \sqcup S'$ for joining of two flow-sensitive abstract states. When c is in the domain of exactly one of S or S', $S \sqcup S'$ agrees with the corresponding mapping. When c is in neither domain, it isn't in the domain of $S \sqcup S'$ either. Finally, when c is in both domains, we have $(S \sqcup S')(c) = S(c) \sqcup S'(c)$.

Also define $S \sqsubseteq S'$ to mean that, whenever $S(c) = s$, there exists s' such that $S'(c) = s'$ and $s \sqsubseteq s'$.

Now our procedure works as follows.

(1) Initialize $S = \bullet[c \mapsto \lambda x.\ \top]$.
(2) Compute $S' = S \sqcup \bigsqcup_{S(c)=s} \mathcal{S}(s, c)$.
(3) If $S' \sqsubseteq S$, then we're done; S is the invariant.
(4) Otherwise, assign $S = S'$ and return to 2.

Again, every step in this outline is computable, for the same reason as in the prior section.

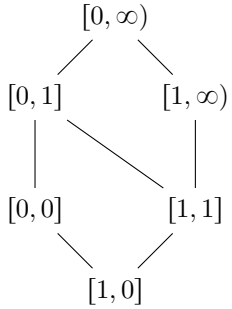
Invariants

THEOREM 8.6. *If the outline above terminates, then it is an invariant of the flow-sensitive abstracted system that, for reachable (s, c), we have $S(c) = s'$ for some s' with $s \sqsubseteq s'$.*

Again, the last two theorems together give us a recipe for computing an invariant automatically, when the loop terminates. The flow-sensitive procedure is guaranteed to give an invariant at least as strong as what the flow-insensitive procedure would come up with, and often it's much stronger. However, flow-sensitive analysis is often much more computationally expensive (in time and memory), so there is a trade-off.

8.4. Widening

Consider an abstract interpretation of *intervals*, where each elements of the domain is either $[a, b]$ or $[a, \infty)$, for $a, b \in \mathbb{N}$. Restricting our attention to a and b values between 0 and 1 for illustration purposes, we have this diagram of the domain, where the bottom element represents an empty set.



The abstract operators have intuitive and simple definitions, like, flattening the different kinds of intervals into a common notation, defining $(a_1, b_1) \sqcup (a_2, b_2) =$

$(\min(a_1, a_2), \max(b_1, b_2))$ and $(a_1, b_1)\hat{+}(a_2, b_2) = (a_1 + a_2, b_1 + b_2)$, with usual conventions about what it means to do arithmetic with ∞.

Again, the lattice diagram above was simplified to cover only 0 and 1 as legal constant values. We can define the interval lattice to draw from the full, infinite set of natural numbers. In that case, we can quickly run into trouble with abstract interpretation. For instance, consider this infinite-looping program:

$$\mathsf{a} \leftarrow 7; \mathsf{while\ a\ do\ a} \leftarrow \mathsf{a} + 3$$

One (flow-insensitive) invariant is that $\mathsf{a} \geqslant 7$, represented as the abstract state $\bullet[\mathsf{a} \mapsto [7, \infty)]$. However, even the flow-sensitive analysis will keep growing the range of a, as it traverses the loop over and over! We see a initialized to $[7, 7]$, then grown to $[7, 10]$ after one loop iteration, then to $[7, 13]$ after another, and so on indefinitely.

Notice that we wrote before that termination is guaranteed when the lattice has finite height, which we have just demonstrated is not true for general intervals, as our example program generates an infinite ascending chain of distinct intervals.

The canonical solution to this problem is to employ a *widening* operator $\triangledown$. This operator has the same soundness requirements as $\sqcup$, but we do not require that it gives the *least* upper bound of its two operands. It merely needs to give some upper bound. In fact, we don't want it to give least upper bounds; we want it to *skip ahead* in that ordering as necessary to promote termination. In general, we don't want to replace all uses of $\sqcup$ with $\triangledown$, though it is sound to do so. We might apply $\triangledown$ in place of $\sqcup$ only for commands that are the beginnings of loops, for instance, to guarantee that no infinite path in the program avoids infinitely many encounters with $\triangledown$ to tame infinite ascending chains.

For intervals, when we are working with programs that we fear will keep increasing variables indefinitely through loops, a simple form of widening is defined as follows. Set $(a_1, b_1)\triangledown(a_2, b_2) = (a_1, b_1) \sqcup (a_2, b_2)$ when $b_2 \leqslant b_1$, that is, when the upper bound of the interval hasn't increased since the last iteration. Otherwise, set $(a_1, b_1)\triangledown(a_2, b_2) = (\min(a_1, a_2), \infty)$. In other words, when an interval expands to include higher values, fast-forward its upper bound to ∞.

With this modification, analysis of our tricky example successfully finds the invariant $\mathsf{a} \geqslant 7$. In fact, flow-insensitive and flow-sensitive interval analysis with this widening operator applied at loop starts are guaranteed to terminate, for any input programs.

CHAPTER 9

Compiler Correctness via Simulation Arguments

A good application of operational semantics is correctness of compiler transformations. A compiler is composed of a series of *phases*, each of which translates programs in some *source* language into some *target* language. Usually, in most phases of a compiler, the source and target languages are the same, and such phases are often viewed as *optimizations*, which tend to improve performance of most programs in practice. The verification problem is plenty hard enough when the source and target languages are the same, so we will confine our attention in this chapter to a single language. It's almost the same as the imperative language from the last two chapters, but we add one new syntactic construction, underlined below.

$$\begin{array}{rrcl} \text{Numbers} & n & \in & \mathbb{N} \\ \text{Variables} & x & \in & \mathsf{Strings} \\ \text{Expressions} & e & ::= & n \mid x \mid e + e \mid e - e \mid e \times e \\ \text{Commands} & c & ::= & \mathsf{skip} \mid x \leftarrow e \mid c; c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \mathsf{while}\ e\ \mathsf{do}\ c \mid \underline{\mathsf{out}(e)} \end{array}$$

A command $\mathsf{out}(e)$ outputs the value of expression e, say by writing it to a terminal window. What's interesting about adding output is that now *different nonterminating programs have interestingly different behavior*: they may produce different output sequences, finite or infinite. Any compiler phase should leave output behavior intact. It's worth noticing that our workhorse technique of invariants can't help us here directly. Output equivalence can only be judged by watching full runs of programs. A nonterminating program that has behaved itself up to some point, satisfying the invariant of our choice, may still fail to follow through later on. While invariants are complete for *safety* properties, here we have our first systematic study of a class of *liveness* properties. We must also delve into establishing *relational* properties of programs, meaning that we reason about connections between executions of two different programs. In our case, such a pair will include the program fed as input into a phase, plus the program that the phase generates.

To get started phrasing the correctness condition formally, we need to modify our operational semantics to track output. We do so by adopting a *labeled transition system*, where step arrows are annotated with *labels* that explain interactions with the world. For this language, the only interaction kind is an output, which we will write as a number. We also have *silent* labels ϵ, for when no output takes place. For completeness, here are the full rules of the extended language, where the definitions of contexts and plugging are inherited unchanged.

$$\frac{}{(v, \mathsf{out}(e)) \xrightarrow{[\![e]\!]v}_0 (v, \mathsf{skip})}$$

$$\frac{}{(v, x \leftarrow e) \xrightarrow{\epsilon}_0 (v[x \mapsto [\![e]\!]v], \mathsf{skip})} \quad \frac{}{(v, \mathsf{skip}; c_2) \xrightarrow{\epsilon}_0 (v, c_2)}$$

$$\frac{[\![e]\!]v \neq 0}{(v, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \xrightarrow{\epsilon}_0 (v, c_1)} \quad \frac{[\![e]\!]v = 0}{(v, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \xrightarrow{\epsilon}_0 (v, c_2)}$$

$$\frac{[\![e]\!]v \neq 0}{(v, \mathsf{while}\ e\ \mathsf{do}\ c_1) \xrightarrow{\epsilon}_0 (v, c_1; \mathsf{while}\ e\ \mathsf{do}\ c_1)} \quad \frac{[\![e]\!]v = 0}{(v, \mathsf{while}\ e\ \mathsf{do}\ c_1) \xrightarrow{\epsilon}_0 (v, \mathsf{skip})}$$

$$\frac{(v, c) \xrightarrow{\ell}_0 (v', c')}{(v, C[c]) \xrightarrow{\ell}_{\mathsf{c}} (v', C[c'])}$$

To reason about infinite executions, we need a new abstraction, compared to what has worked in our invariant-based proofs so far. That abstraction will be *traces*, sequences of outputs (and termination events) that a program might be observed to generate. We define a command's trace set inductively. Recall that $\cdot$ is the empty list, while $\bowtie$ does list concatenation.

$$\frac{}{\cdot \in \mathsf{Tr}(s)} \quad \frac{}{\mathsf{terminate} \in \mathsf{Tr}((v, \mathsf{skip}))} \quad \frac{s \xrightarrow{\epsilon}_{\mathsf{c}} s' \quad t \in \mathsf{Tr}(s')}{t \in \mathsf{Tr}(s)} \quad \frac{s \xrightarrow{n}_{\mathsf{c}} s' \quad t \in \mathsf{Tr}(s')}{\mathsf{out}(n) \bowtie t \in \mathsf{Tr}(s)}$$

Notice that a trace is allowed to end at any point, even if the program under inspection hasn't terminated yet. Also, since our language is deterministic, for any two traces of one command, one trace is a prefix of the other. Many parts of the machinery we develop here will, however, work well for nondeterministic systems, as we will see with labeled transition systems for concurrency in Chapter 19.

Definition 9.1 (Trace inclusion). For commands c_1 and c_2, let $c_1 \preceq c_2$ iff $\mathsf{Tr}(c_1) \subseteq \mathsf{Tr}(c_2)$.

Definition 9.2 (Trace equivalence). For commands c_1 and c_2, let $c_1 \simeq c_2$ iff $\mathsf{Tr}(c_1) = \mathsf{Tr}(c_2)$.

We will enforce that a correct compiler phase respects trace equivalence. That is, the output program has the same traces as the input program. For nondeterministic languages, subtler conditions are called for, but we're happy to stay within the safe confines of determinism for this chapter.

9.1. Basic Simulation Arguments and Optimizing Expressions

As our first example compiler phase, we consider a limited form of *constant folding*, where expressions with statically known values are replaced by constants. The whole of the optimization is (1) finding all maximal program subexpressions that don't contain variables and (2) replacing each such subexpression with its known constant value. We write $\mathsf{cfold}_1(c)$ for the result of applying this optimization on command c. (For the program transformations in this chapter, we stick to informal descriptions of how they operate, leaving the details to the accompanying Coq code.)

A program optimized in this way proceeds in a very regular manner, compared to executions of the original, unoptimized program. The small steps line up one-to-one. Therefore, a very regular kind of *simulation relation* connects them. (This notion is very similar to the one from Section 6.2, though now it incorporates labels.)

Definition 9.3 (Simulation relation). We say that binary relation R over states of our object language is a *simulation relation* iff:

(1) Whenever $(v_1, \mathsf{skip})\ R\ (v_2, c_2)$, it follows that $c_2 = \mathsf{skip}$.
(2) Whenever $s_1\ R\ s_2$ and $s_1 \xrightarrow{\ell}_{\mathsf{c}} s_1'$, there exists s_2' such that $s_2 \xrightarrow{\ell}_{\mathsf{c}} s_2'$ and $s_1'\ R\ s_2'$.

The crucial second condition can be drawn like this.

$$\begin{array}{ccc} s_1 & \xrightarrow{\quad R \quad} & s_2 \\ \Big\downarrow{\scriptstyle \forall \xrightarrow{\ell}_{\mathsf{c}}} & & \Big\downarrow{\scriptstyle \exists \xrightarrow{\ell}_{\mathsf{c}}} \\ s_1' & \xleftarrow[R^{-1}]{} & s_2' \end{array}$$

Invariants

As usual, the diagram tells us that when a path along the left exists, a matching roundabout path exists, too. That is, any step on the left can be matched by a step on the right. Notice the similarity to the invariant-induction principle that we have mostly relied on so far. Instead of showing that every step preserves a one-state predicate, we show that every step preserves a two-state predicate in a particular way. The simulation approach is as general for relating programs as the invariant approach is for verifying individual programs.

THEOREM 9.4. *If there exists a simulation R such that $s_1\ R\ s_2$, then $s_1 \simeq s_2$.*

PROOF. We prove the two trace-inclusion directions separately. The left-to-right direction proceeds by induction over the definition of traces on the left, while the right-to-left direction proceeds by similar induction on the right. While most of the proof is generic in details of the labeled transition system, for the right-to-left direction we do rely on proofs of two important properties of this object language. First, the semantics is *total*, in the sense that any state whose command isn't skip can take a step. Second, the semantics is *deterministic*, in that there can be at most one label/state pair reachable in one step from a particular starting state.

In the inductive step of the right-to-left inclusion proof, we know that the righthand system has taken a step. The lefthand system might already be a skip, in which case, by the definition of simulations, the righthand system is already a skip, contradicting the assumption that the righthand side stepped. Otherwise, by totality, the lefthand system can take a step. By the definition of simulation, there exists a matching step on the righthand side. By determinism, the matching step is the same as the one we were already aware of. Therefore, we have a new R relationship to connect to that step and apply the induction hypothesis. □

We can apply this very general principle to constant folding.

THEOREM 9.5. *For any v and c, $(v, c) \simeq (v, \mathsf{cfold}_1(c))$.*

PROOF. By a simulation argument using this relation:

$$(v_1, c_1)\ R\ (v_2, c_2) \quad = \quad v_1 = v_2 \wedge c_2 = \mathsf{cfold}_1(c_1)$$

What we have done is translate the original theorem statement into the language of binary relations, as this simple case needs no equivalent of strengthening the induction hypothesis. Internally to the proof, we need to define constant folding of evaluation contexts C, and we need to prove that primitive steps $\rightarrow_0$ may be lifted to apply over constant-folded states, this second proof by case analysis on $\rightarrow_0$ derivations. Another more obvious workhorse is a lemma showing that constant folding of expressions preserves interpretation results. □

9.2. Simulations That Allow Skipping Steps

Consider an evolution of our constant-folding optimization to take advantage of known values of if test expressions. Depending on whether the value is zero, we can replace the whole if with one of its two cases. We will write $\mathsf{cfold}_2(c)$ for this expanded optimization and work up to proving it sound, too. However, we can no longer use last section's definition of simulation! The reason is that optimizations intentionally cut down on steps that a program needs to execute. Some steps of the source program now have no matching steps of the target program, say when we are stepping an if whose test expression had a known value.

Let's take a first crack at making simulation more flexible.

Definition 9.6 (Simulation relation with skipping (*faulty* version!)). We say that binary relation R over states of our object language is a *simulation relation with skipping* iff:

(1) Whenever $(v_1, \mathsf{skip})\ R\ (v_2, c_2)$, it follows that $c_2 = \mathsf{skip}$.
(2) Whenever $s_1\ R\ s_2$ and $s_1 \xrightarrow{\ell}_{\mathsf{c}} s_1'$, then either:
   (a) there exists s_2' such that $s_2 \xrightarrow{\ell}_{\mathsf{c}} s_2'$ and $s_1'\ R\ s_2'$,
   (b) or $\ell = \epsilon$ and $s_1'\ R\ s_2$.

In other words, to match a silent step, it suffices to do nothing, so long as R still holds afterward.

We didn't mark the definition as *faulty* for nothing. It actually does not imply trace equivalence. Consider a questionable "optimization" defined as $\mathsf{withAds}(\mathsf{while}\ 1\ \mathsf{do}\ \mathsf{skip}) = \mathsf{while}\ 1\ \mathsf{do}\ \mathsf{out}(0)$, and $\mathsf{withAds}(c) = c$ for all other c. It adds a little extra advertisement into a particular infinite loop. Now we define a candidate simulation relation.

$$(v_1, c_1)\ R\ (v_2, c_2) \quad = \quad c_1 \in \{\mathsf{while}\ 1\ \mathsf{do}\ \mathsf{skip}, (\mathsf{skip}; \mathsf{while}\ 1\ \mathsf{do}\ \mathsf{skip})\}$$

This suspicious relation records nothing about c_2. The skip condition of simulations is handled trivially, as we can see by inspection that R does not allow c_1 to be skip. Checking the execution-matching condition of simulations, c_1 is either while 1 do skip or (skip; while 1 do skip), each of which steps silently to the other. We may match either step by keeping c_2 in place, as R does not constrain c_2 at all. Thus, R is a simulation relation with skipping, and, for $c =$ while 1 do skip, it relates c to $\mathsf{withAds}(c)$.

From here we expect to conclude trace equivalence. However, clearly withAds can turn a program that never outputs into a program that outputs infinitely often!

Let's patch our definition.

Definition 9.7 (Simulation relation with skipping). We say that an $\mathbb{N}$-indexed family of binary relations R_n over states of our object language is a *simulation relation with skipping* iff:

(1) Whenever $(v_1, \mathsf{skip})\ R_n\ (v_2, c_2)$, it follows that $c_2 = \mathsf{skip}$.
(2) Whenever $s_1\ R_n\ s_2$ and $s_1 \xrightarrow{\ell}_{\mathsf{c}} s_1'$, then either:
   (a) there exist n' and s_2' such that $s_2 \xrightarrow{\ell}_{\mathsf{c}} s_2'$ and $s_1'\ R_{n'}\ s_2'$,
   (b) or $n > 0$, $\ell = \epsilon$, and $s_1'\ R_{n-1}\ s_2$.

This new version imposes a finite limit n at any point, on how many times the righthand side may match lefthand steps without stepping itself. Our bad counterexample fails to satisfy the conditions, because eventually the starting step

count n will be used up, and the incorrect "optimized" program will be forced to reveal itself by taking a step that outputs.

THEOREM 9.8. *If there exists a simulation with skipping R such that $s_1\ R_n\ s_2$, then $s_1 \simeq s_2$.*

PROOF. The proof is fairly similar to that of Theorem 9.4. To show termination preservation in the backward direction, we find ourselves proving a lemma by induction on n. □

THEOREM 9.9. *For any v and c, $(v, c) \simeq (v, \mathsf{cfold}_2(c))$.*

PROOF. By a simulation argument (with skipping) using this relation:

$$(v_1, c_1)\ R_n\ (v_2, c_2) \quad = \quad v_1 = v_2 \wedge c_2 = \mathsf{cfold}_2(c_1) \wedge \mathsf{countIfs}(c_1) < n$$

We rely on a simple helper function $\mathsf{countIfs}(c)$ to count how many If nodes appear in the syntax of c. This notion turns out to be a conservative upper bound on how many times in a row we will need to let lefthand steps go unmatched on the right. The rest of the proof proceeds essentially the same way as in Theorem 9.5. □

9.3. Simulations That Allow Taking Multiple Matching Steps

Consider our final example compiler phase: flattening expressions into sequences of assignments to temporaries, using only noncompound subexpressions, where the arguments to every binary operator are variables or constants. Now a single step at the source level must be matched by many steps at the target level. We write $\mathsf{flatten}(c)$ for the flattening of command c. How can we prove that this transformation is correct?

DEFINITION 9.10 (Simulation relation with multiple matching steps). We say that a binary relation R over states of our object language is a *simulation relation with multiple matching steps* iff:

(1) Whenever $(v_1, \mathsf{skip})\ R\ (v_2, c_2)$, it follows that $c_2 = \mathsf{skip}$.
(2) Whenever $s_1\ R\ s_2$ and $s_1 \xrightarrow{\ell}_{\mathsf{c}} s_1'$, there exists s_2' such that $s_2 \xrightarrow{\ell}_{\mathsf{c}}{}^{*} s_2'$ and $s_1'\ R\ s_2'$.

We write $s \xrightarrow{\ell}_{\mathsf{c}}{}^{*} s'$ to indicate that s steps to s' via zero or more silent steps and then one step with label ℓ (which might also be silent).

THEOREM 9.11. *If there exists a simulation with multiple matching steps R such that $s_1\ R\ s_2$, then $s_1 \simeq s_2$.*

PROOF. The backward direction is the interesting part of this proof. The key lemma proceeds by strong induction on the number of steps needed to generate the trace on the right. □

THEOREM 9.12. *For any v and c where c doesn't use any names that are reserved for temporaries, $(v, c) \simeq (v, \mathsf{flatten}(c))$.*

PROOF. By a simulation argument (with multiple matching steps) using this relation:

$$\begin{aligned}(v_1, c_1)\ R\ (v_2, c_2) \quad = \quad & c_1 \text{ doesn't use any names reserved for temporaries} \\ & \wedge\ v_1 \cong v_2 \wedge c_2 = \mathsf{flatten}(c_1)\end{aligned}$$

The heart of this relation is a subrelation $\cong$ over valuations, capturing when they agree on all variables that are not reserved for temporaries, since the flattened program will feel free to scribble all over the temporaries. The details of $\cong$ are especially important to the key lemma, showing that flattening of expressions is sound, both taking in a $\cong$ premise and drawing a related $\cong$ conclusion. The overall proof is not short, with quite a few lemmas, found in the Coq code. □

It might not be clear why we bothered to define simulation with multiple matching steps, when we already had simulation with skipping. After all, we use simulation to conclude completely symmetric facts about two commands, so why not just verify this section's example by applying simulation with skipping, with the operand order reversed?

Consider the heart of the proof approach that we *did* adopt. We need to show that any step of c can be matched suitably by $\mathsf{flatten}(c)$. The proof is divided into cases by inversion on a premise $(v, c) \xrightarrow{\ell}_{\mathsf{c}} (v', c')$. Each case naturally fixes the top-level structure of c, from which we can apply straightforward algebraic simplification to find the top-level structure of $\mathsf{flatten}(c)$ and therefore the step rules that apply to it.

Now consider applying simulation with skipping, with the commands passed as operands in the reverse order. The crucial inversion is on $(v, \mathsf{flatten}(c)) \xrightarrow{\ell}_{\mathsf{c}} (v', c')$. Unfortunately, the top-level structure of $\mathsf{flatten}(c)$ does not imply the top-level structure of c, but we need to show that c can take a matching step. We need to prove a whole set of bothersome special-case inversion lemmas by induction, essentially to invert the action of what is, in the general case, an arbitrarily complex compiler.

CHAPTER 10

Lambda Calculus and Simple Type Safety

We'll now take a break from the imperative language we've been studying for the last three chapters, instead looking at a classic sort of small language that distills the essence of *functional* programming. That's the language paradigm that we've been using throughout this book, as we coded executable versions of algorithms. Its distinctive characteristics are first, a computation style based on simplifying terms instead of running step-by-step instructions that modify state; and second, use of functions as first-class values. Functional programming went mainstream in the early 21st century, influencing widely adopted languages from JavaScript, where first-class functions are routinely used as callbacks in asynchronous event processing; to Scala, a hybrid language that melds functional-programming ideas with object-oriented programming for the Java platform; to Haskell, a purely functional language that has become popular with programming hobbyists and is seeing increasing adoption in industry.

The heart of functional programming persists even in λ-*calculus* (or lambda calculus), the simplest version of which contains just three syntactic forms, but which provides probably the simplest of the widely known Turing-complete languages that is (nearly!) pleasant to program in directly.

10.1. Untyped Lambda Calculus

Here is the syntax of the original λ-calculus.

$$\begin{array}{rrcl} \text{Variables} & x & \in & \mathsf{Strings} \\ \text{Expressions} & e & ::= & x \mid \lambda x.\ e \mid e\ e \end{array}$$

An expression $\lambda x.\ e$ is a first-class, anonymous function, also called a *function abstraction* or λ-*abstraction*. When called, it replaces its formal-argument variable x with the actual argument within e and continues evaluating. The third syntactic form $e\ e$ uses *juxtaposition*, or writing one term after another, for function application.

A simple example of an expression is $\lambda x.\ x$, for an identity function. When we apply it to itself, like $(\lambda x.\ x)\ (\lambda x.\ x)$, it reduces again to itself.

We can give a simple big-step operational semantics to λ-terms. The key auxiliary operation is *substitution*, where we write $[e'/x]e$ for replacing all *free* occurrences of x in e with e'. Here we refer to a notion of *free variables*, which we should define first, as a recursive function.

$$\begin{array}{rcl} \mathsf{FV}(x) & = & \{x\} \\ \mathsf{FV}(\lambda x.\ e) & = & \mathsf{FV}(e) - \{x\} \\ \mathsf{FV}(e_1\ e_2) & = & \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2) \end{array}$$

Intuitively, a variable is free in an expression iff it doesn't occur inside the scope of a λ binding the same variable.

Next we define substitution.

$$\begin{array}{rcl} [e'/x]x & = & e' \\ [e'/x]y & = & y, \text{ if } y \neq x \\ [e'/x]\lambda x.\ e & = & \lambda x.\ e \\ [e'/x]\lambda y.\ e & = & \lambda y.\ [e'/x]e, \text{ if } y \neq x \\ [e'/x]e_1\ e_2 & = & [e'/x]e_1\ [e'/x]e_2 \end{array}$$

Notice a peculiar property of this definition when we work with *open* terms, whose free-variable sets are nonempty. According to the definition $[x/y]\lambda x.\ y = \lambda x.\ x$. In this example, we say that λ-bound variable x has been *captured* unintentionally, where substitution created a reference to that λ where none existed before. Such a problem can only arise when replacing a variable with an open term. In this case, that term is x, where $\mathsf{FV}(x) = \{x\} \neq \emptyset$.

More general investigations into λ-calculus will define a more involved notion of *capture-avoiding* substitution. Instead, in this book, we carefully steer clear of the λ-calculus applications that require substituting open terms for variables, letting us stick with the simpler definition. When it comes to formal encoding of this style of syntax in proof assistants, surprisingly many complications arise, leading to what is still an active research area in encodings of language syntax with local variable binding. Since we aim more for broad than deep coverage of the field of formal program reasoning, we are happy to avoid those complexities.

With substitution in hand, a big-step semantics is easy to define. We use the syntactic shorthand v for a *value*, or term that needs no further evaluation, which in this case includes just the λ-abstractions.

Encoding

$$\frac{}{\lambda x.\ e \Downarrow \lambda x.\ e} \qquad \frac{e_1 \Downarrow \lambda x.\ e \quad e_2 \Downarrow v \quad [v/x]e \Downarrow v'}{e_1\ e_2 \Downarrow v'}$$

A value evaluates to itself. To evaluate an application, evaluate both the function and the argument. The function value must be some λ-abstraction. Substitute the argument value in the body of the abstraction, evaluate the result, and return that value as the overall value. Note that we only ever need to evaluate *closed* terms, meaning terms that are not open, so we obey the restriction on substitution sketched above.

It may be surprising that these two rules are enough to define the full semantics of a Turing-complete language! Indeed, λ-calculus is Turing-complete, and we must be able to find nonterminating programs. Here is one example.

$$\Omega \quad = \quad (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$$

THEOREM 10.1. *Ω does not evaluate to anything. In other words, $\Omega \Downarrow v$ implies a contradiction.*

PROOF. By induction on the derivation of $\Omega \Downarrow v$. □

10.2. A Quick Case Study in Program Verification: Church Numerals

Since λ-calculus is Turing-complete, it must be able to represent numbers and all the usual arithmetic operations. The classic representation is *Church numerals*, where every natural number n is represented as a particular λ-term $\underline{n}$ that, when passed a function f as input, returns f^n, the n-way self-composition of f. In some sense, repeating a process is the fundamental use of a natural number, and it turns out that we can recover all of the usual operations atop this primitive.

Two λ-calculus functions are sufficient to build up all the naturals as Church numerals.

$$\begin{array}{rcl} \mathsf{zero} & = & \lambda f.\ \lambda x.\ x \\ \mathsf{plus1} & = & \lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x) \end{array}$$

Our representation of 0 returns an identity function, no matter which f it is passed. Our successor operation takes in a number n and returns a new one that first runs n and then applies f one extra time. Now we have $\underline{0} = \mathsf{zero}$, $\underline{1} = \mathsf{plus1}\ \mathsf{zero}$, $\underline{2} = \mathsf{plus1}\ (\mathsf{plus1}\ \mathsf{zero})$, and so on.

These Church numerals are not values yet. Let us formalize which values they evaluate to and tweak the encoding to use the values instead. We write $\lfloor n \rfloor$ for the body of a λ-abstraction that we are building to represent n, where variables f and x are in scope.

$$\begin{array}{rcl} \lfloor 0 \rfloor & = & x \\ \lfloor n+1 \rfloor & = & f\ ((\lambda f.\ \lambda x.\ \lfloor n \rfloor)\ f\ x) \end{array}$$

The $n + 1$ case may seem wastefully large, but, in fact, this is the precise form of the values produced by evaluating repeated applications of $\mathsf{plus1}$ to zero, as the reader can verify using the big-step semantics. We define $\underline{n} = \lambda f.\ \lambda x.\ \lfloor n \rfloor$, giving a canonical encoding for each number.

Now we notate correctness of an encoding e for number n by $e \sim n$, defining it as $e \Downarrow \underline{n}$, meaning that e evaluates to the Church encoding of n. Two first easy results show that our primitive constructors are correct.

Theorem 10.2. $\mathsf{zero} \sim 0$.

Theorem 10.3. *If* $e_n \sim n$, *then* $\mathsf{plus1}\ e_n \sim n + 1$.

Things get more interesting as we start to code up the arithmetic operations.

$$\mathsf{add} \quad = \quad \lambda n.\ \lambda m.\ n\ \mathsf{plus1}\ m$$

That is, addition of n to m is calculated by applying n $\mathsf{plus1}$ operations to m.

Theorem 10.4. *If* $e_n \sim n$ *and* $e_m \sim m$, *then* $\mathsf{add}\ e_n\ e_m \sim n + m$.

Proof. After a few steps applying the big-step rules directly, we finish by induction on n. A silly-seeming but necessary lemma proves that $[e/m]\lfloor n \rfloor = \lfloor n \rfloor$, since $\lfloor n \rfloor$ does not contain free occurrences of m. □

Multiplication proceeds in much the same way.

$$\mathsf{mult} \quad = \quad \lambda n.\ \lambda m.\ n\ (\mathsf{add}\ m)\ \mathsf{zero}$$

Theorem 10.5. *If* $e_n \sim n$ *and* $e_m \sim m$, *then* $\mathsf{mult}\ e_n\ e_m \sim n \times m$.

Proof. After a few steps applying the big-step rules directly, we finish by induction on n, within which we appeal to Theorem 10.4. □

An enjoyable (though not entirely trivial) exercise for the reader is to generalize the methods of Church encoding to encoding of other inductive datatypes, including the syntax of λ-calculus itself. A hallmark of a Turing-complete language is that it can host an interpreter for itself, and λ-calculus is no exception!

10.3. Small-Step Semantics

λ-calculus is also straightforward to formalize with a small-step semantics and evaluation contexts, following the method of Section 7.3.2. One might argue that the technique is even simpler for λ-calculus, since we must deal only with expressions, not also imperative variable valuations.

$$\text{Evaluation contexts} \quad C \quad ::= \quad \Box \mid C\ e \mid v\ C$$

Note the one subtlety: the last form of evaluation context requires the term in a function position to be a *value*. This innocuous-looking restriction enforces *call-by-value evaluation order*, where, upon encountering a function application, we must first evaluate the function, then evaluate the argument, and only then call the function. Tweaks to the definition of C produce other evaluation orders, like *call-by-name*, but we will say no more about those alternatives.

We assume a standard definition of what it means to plug an expression into the hole in a context, and now we can give the sole small-step evaluation rule for basic λ-calculus, conventionally called the *β-reduction* rule.

Encoding

$$\frac{}{C[(\lambda x.\ e)\ v] \to C[[v/x]e]}$$

That is, we find a suitable position within the expression where a λ-expression is applied to a value, and we replace that position with the appropriate substitution result.

Following a very similar outline to what we used in Chapter 7, we establish equivalence between the two semantics for λ-calculus.

Theorem 10.6. *If $e \to^* v$, then $e \Downarrow v$.*

Theorem 10.7. *If $e \Downarrow v$, then $e \to^* v$.*

There are a few proof subtleties beyond what we encountered before, and the Coq formalization may be worth reading, to see those details.

Again as before, we have a natural way to build a transition system from any λ-term e, where $\mathcal{L}$ is the set of λ-terms. We define $\mathbb{T}(e) = \langle \mathcal{L}, \{e\}, \to \rangle$. The next section gives probably the most celebrated λ-calculus result based on the transition-system perspective.

10.4. Simple Types and Their Soundness

Let's spruce up the language with some more constructs.

$$\begin{array}{rrcl} \text{Variables} & x & \in & \mathsf{Strings} \\ \text{Numbers} & n & \in & \mathbb{N} \\ \text{Expressions} & e & ::= & n \mid e + e \mid x \mid \lambda x.\ e \mid e\ e \\ \text{Values} & v & ::= & n \mid \lambda x.\ e \end{array}$$

We've added natural numbers as a primitive feature, supported via constants and addition. Numbers may be intermixed with functions, and we may, for instance, write first-class functions that take numbers as input or return numbers.

Our language of evaluation contexts expands a bit.

$$\text{Evaluation contexts} \quad C \quad ::= \quad \Box \mid C\ e \mid v\ C \mid C + e \mid v + C$$

Now we want to define two kinds of basic small steps, so it is worth defining a separate relation for them. Here we face a classic nuisance in writing rules that combine explicit syntax with standard mathematical operators, and we write + for the syntactic construct and + for the mathematical addition operator.

$$\frac{}{(\lambda x.\ e)\ v \rightarrow_0 [v/x]e} \quad \frac{}{n + m \rightarrow_0 n{+}m}$$

Here is the overall step rule.

$$\frac{e \rightarrow_0 e'}{C[e] \rightarrow C[e']}$$

What would be a useful property to prove about our new expressions? For one thing, we don't want them to "crash," as in the expression $(\lambda x.\ x) + 7$ that tries to add a function and a number. No rule of the semantics knows what to do with that case, but it also isn't a value, so we shouldn't consider it as finished with evaluation. Define an expression as *stuck* when it is not a value and it cannot take a small step. For "reasonable" expressions e, we should be able to prove that it is an invariant of $\mathbb{T}(e)$ that no expression is ever stuck.

To define "reasonable," we formalize the popular idea of a static type system. Every expression will be assigned a type, capturing which sorts of contexts it may legally be dropped into. Our language of types is simple.

Abstraction

$$\text{Types} \quad \tau \quad ::= \quad \mathbb{N} \mid \tau \rightarrow \tau$$

We have trees of function-space constructors, where all the leaves are instances of the natural-number type $\mathbb{N}$. Note that, with type assignment, we have yet another case of *abstraction*, approximating a potentially complex expression with a type that only records enough information to rule out crashes.

To assign types to closed terms, we must recursively define what it means for an open term to have a type. To that end, we use *typing contexts* Γ, finite maps from variables to types. To mimic standard notation, we write $\Gamma, x : \tau$ as shorthand for $\Gamma[x \mapsto \tau]$, overriding of key x with value τ in Γ. Now we define typing as a three-place relation, written $\Gamma \vdash e : \tau$, to indicate that, assuming Γ as an assignment of types to e's free variables, we conclude that e has type τ.

We define the relation inductively, with one case per syntactic construct.

Modularity

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad \frac{}{\Gamma \vdash n : \mathbb{N}} \quad \frac{\Gamma \vdash e_1 : \mathbb{N} \quad \Gamma \vdash e_2 : \mathbb{N}}{\Gamma \vdash e_1 + e_2 : \mathbb{N}}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \rightarrow \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

We write $\vdash e : \tau$ as shorthand for $\bullet \vdash e : \tau$, meaning that closed term e has type τ, with no typing context required. Note that this style of typing rules provides another instance of *modularity*, since we can separately type-check different subexpressions of a large expression, using just their types to coordinate expectations among subexpressions.

It should be an invariant of $\mathbb{T}(e)$ that every reachable expression has the same type as the original, so long as the original was well-typed. This observation is the key to proving that it is also an invariant that no reachable expression is stuck,

using a proof technique called *the syntactic approach to type soundness*, which turns out to be just another instance of our general toolbox for invariant proofs.

We work our way through a suite of standard lemmas to support that invariant proof.

LEMMA 10.8 (Progress). *If* $\vdash e : \tau$*, then* e *isn't stuck.*

PROOF. By induction on the derivation of $\vdash e : \tau$. □

LEMMA 10.9 (Weakening). *If* $\Gamma \vdash e : \tau$ *and every mapping in* Γ *is also included in* Γ'*, then* $\Gamma' \vdash e : \tau$*.*

PROOF. By induction on the derivation of $\Gamma \vdash e : \tau$. □

LEMMA 10.10 (Substitution). *If* $\Gamma, x : \tau' \vdash e : \tau$ *and* $\vdash e' : \tau'$*, then* $\Gamma \vdash [e'/x]e : \tau$*.*

PROOF. By induction on the derivation of $\Gamma, x : \tau' \vdash e : \tau$, with appeal to Lemma 10.9. □

LEMMA 10.11. *If* $e \to_0 e'$ *and* $\vdash e : \tau$*, then* $\vdash e' : \tau$*.*

PROOF. By inversion on the derivation of $e \to_0 e'$, with appeal to Lemma 10.10. □

LEMMA 10.12. *If any type of* e_1 *is also a type of* e_2*, then any type of* $C[e_1]$ *is also a type of* $C[e_2]$*.*

PROOF. By induction on the structure of C. □

LEMMA 10.13 (Preservation). *If* $e_1 \to e_2$ *and* $\vdash e_1 : \tau$*, then* $\vdash e_2 : \tau$*.*

PROOF. By inversion on the derivation of $e_1 \to e_2$, with appeal to Lemmas 10.11 and 10.12. □

Invariants

THEOREM 10.14 (Type Soundness). *If* $\vdash e : \tau$*, then* $\neg$*stuck is an invariant of* $\mathbb{T}(e)$*.*

PROOF. First, we strengthen the invariant to $I(e) = \vdash e : \tau$, justifying the implication by Lemma 10.8, Progress. Then we apply invariant induction, where the base case is trivial. The induction step is a direct match for Lemma 10.13, Preservation. □

The syntactic approach to type soundness is often presented as a proof technique in isolation, but what we see here is that it follows very directly from our general invariant proof technique. Usually syntactic type soundness is presented as fundamentally about proving Progress and Preservation conditions. The Progress condition maps to invariant strengthening, and the Preservation condition maps to invariant induction, which we have used in almost every invariant proof so far. Since the basic proof structure matches our standard one, the main insight is the usual one: a good choice of a strengthened invariant. In this case, invariant $I(e) = \vdash e : \tau$ is that crucial insight, including the original design of the set of types and the typing relation.

CHAPTER 11

Types and Mutation

The syntactic approach to type soundness continues to apply to *impure* functional languages, which combine imperative side effects with first-class functions. We'll study the general domain through its most common exemplar: λ-calculus with *mutable references*.

11.1. Simply Typed Lambda Calculus with Mutable References

Here is an extension of the lambda-calculus syntax from last chapter, with additions underlined.

$$\begin{array}{rrcl} \text{Variables} & x & \in & \mathsf{Strings} \\ \text{Numbers} & n & \in & \mathbb{N} \\ \text{Expressions} & e & ::= & n \mid e + e \mid x \mid \lambda x.\ e \mid e\ e \mid \underline{\mathsf{new}(e) \mid\ !e \mid e := e} \end{array}$$

The three new expression forms deal with *references*, which act like, for instance, Java objects that only have single public fields. We write $\mathsf{new}(e)$ to allocate a new reference initialized with value e, we write $!e$ for reading the value stored in reference e, and we write $e_1 := e_2$ for overwriting the value of reference e_1 with e_2. An example is worth a thousand words, so let's consider a concrete program. We'll use two notational shorthands:

$$\begin{array}{rcl} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 & \triangleq & (\lambda x.\ e_2)\ e_1 \\ e_1; e_2 & \triangleq & \mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2 \text{ (for } \_ \text{ a variable not used anywhere else)} \end{array}$$

Here is a simple program that uses references.

$$\mathsf{let}\ r = \mathsf{new}(0)\ \mathsf{in}\ r := !r + 1; !r$$

This program (1) allocates a new reference r storing the value 0; (2) increments r's value by 1; and (3) returns the new r value, which is 1.

To be more formal about the meanings of all programs, we extend the operational semantics from last chapter. First, we add some new kinds of evaluation contexts.

$$\begin{array}{rrcl} \text{Evaluation contexts} & C & ::= & \square \mid C\ e \mid v\ C \mid C + e \mid v + C \\ & & & \mid\ \underline{\mathsf{new}(C) \mid\ !C \mid C := e \mid v := C} \end{array}$$

Next we define the basic reduction steps of the language. In contrast to last chapter's semantics for pure λ-calculus, here we work with states that include not just expressions but also *heaps* h, partial functions from references to their current stored values. We begin by copying over the two basic-step rules from last chapter, threading through the heap h unchanged.

$$\frac{}{(h, (\lambda x.\ e)\ v) \rightarrow_0 (h, [v/x]e)} \qquad \frac{}{(h, n + m) \rightarrow_0 (h, n{+}m)}$$

To write out the rules that are specific to references, it's helpful to extend our language syntax with a form that will never appear in original programs, but which does show up at intermediate execution steps. In particular, let's add an expression form for *locations*, the runtime values of references, and let's say that locations also count as values.

$$\begin{array}{rrcl} \text{Locations} & \ell & \in & \mathbb{N} \\ \text{Expressions} & e & ::= & n \mid e + e \mid x \mid \lambda x.\ e \mid e\ e \mid \mathsf{new}(e) \mid\ !e \mid e := e \mid \underline{\ell} \\ \text{Values} & v & ::= & n \mid \lambda x.\ e \mid \underline{\ell} \end{array}$$

Now we can write the rules for the three reference primitives.

$$\frac{\ell \notin \mathsf{dom}(h)}{(h, \mathsf{new}(v)) \rightarrow_0 (h[\ell \mapsto v], \ell)} \quad \frac{h(\ell) = v}{(h, !\ell) \rightarrow_0 (h, v)} \quad \frac{h(\ell) = v}{(h, \ell := v') \rightarrow_0 (h[\ell \mapsto v'], v')}$$

To evaluate a reference allocation $\mathsf{new}(e)$, we nondeterministically pick some unused location ℓ and initialize it with the requested value. To read from a reference in $!e$, we just look up the location in the heap; the program will be *stuck* if the location is not already included in h. Finally, to write to a reference with $e_1 := e_2$, we check that the requested location is already in the heap (we're stuck if not), then we overwrite its value with the new one.

Here is the overall step rule, which looks just like the one for basic λ-calculus, with a heap wrapped around everything.

$$\frac{(h, e) \rightarrow_0 (h', e')}{(h, C[e]) \rightarrow (h', C[e'])}$$

As a small exercise for the reader, it may be worth using this judgment to derive that our example program from before always returns 1. Even fixing the empty heap in the starting state, there is some nondeterminism in which final heap it returns: the possibilities are all the single-location heaps, mapping their single locations to value 1. It is natural to allow this nondeterminism in allocation, since typical memory allocators in real systems don't give promises about predictability in the addresses that they return. However, we will be able to prove that, for instance, any program returning a number *gives the same answer, independently of nondeterministic choices made by the allocator*. That property is not true in programming languages like C that are not *memory safe*, as they allow arithmetic and comparisons on pointers, the closest C equivalent of our references.

11.2. Type Soundness

For λ-calculus with references, we can prove a similar type-soundness theorem to what we proved last chapter, though the proof has a twist or two. To start with, we should define our extended type system, with one new case for references.

$$\text{Types} \quad \tau \quad ::= \quad \mathbb{N} \mid \tau \rightarrow \tau \mid \underline{\tau\ \mathsf{ref}}$$

Here are the rules from last chapter's basic λ-calculus, which we can keep unchanged.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad \frac{}{\Gamma \vdash n : \mathbb{N}} \quad \frac{\Gamma \vdash e_1 : \mathbb{N} \quad \Gamma \vdash e_2 : \mathbb{N}}{\Gamma \vdash e_1 + e_2 : \mathbb{N}}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \rightarrow \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

We also need a rule for each of the reference primitives.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{new}(e) : \tau\ \mathsf{ref}} \quad \frac{\Gamma \vdash e : \tau\ \mathsf{ref}}{\Gamma \vdash !e : \tau} \quad \frac{\Gamma \vdash e_1 : \tau\ \mathsf{ref} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \tau}$$

That's enough notation to let us state type soundness, which is indeed provable.

THEOREM 11.1 (Type Soundness). *If $\vdash e : \tau$, then $\neg$stuck is an invariant of $\mathbb{T}(e)$.*

However, we will need to develop some more machinery to let us state the strengthened invariant that makes the proof go through.

The trouble with our typing rules is that they disallow location constants, but those constants *will* arise in intermediate states of program execution. To prepare for them, we introduce *heap typings* Σ, partial functions from locations to types. The idea is that a heap typing Σ models a heap h by giving the intended type for each of its locations. We define an expanded typing judgment of the form $\Sigma; \Gamma \vdash e : \tau$, with a new parameter included solely to enable the following rule.

$$\frac{\Sigma(\ell) = \tau}{\Sigma; \Gamma \vdash \ell : \tau}$$

We must also extend every typing rule we gave before, adding an extra "Σ;" prefix, threaded mindlessly through everything. We never extend Σ as we recurse into subexpressions, and we only examine it in leaves of derivation trees, corresponding to ℓ expressions.

We have made some progress toward stating an inductive invariant for the type-soundness theorem. The essential idea of the proof is found in the invariant choice $I(h, e) = \exists \Sigma.\ \Sigma; \bullet \vdash e : \tau$. However, we can tell that something is suspicious with this invariant, since it does not mention h. We should also somehow characterize the relationship between Σ and h.

Here is a first cut at defining a relation $\Sigma \vdash h$.

$$\frac{\forall \ell, \tau.\ \Sigma(\ell) = \tau \Rightarrow \exists v.\ h(\ell) = v \wedge \Sigma; \bullet \vdash v : \tau}{\Sigma \vdash h}$$

In other words, whenever Σ announces the existence of location ℓ meant to store values of type τ, the heap h actually stores some value v for ℓ, and that value has the right type. Note the tricky recursion inherent in typing v with respect to the very same Σ.

This rule as stated is not *quite* sufficient to make the invariant inductive. We could get stuck on a $\mathsf{new}(e)$ expression if the heap h becomes *infinite*, with no free addresses left to allocate. Of course, we know that finite executions, started in the empty heap, only produce finite intermediate heaps. Let's remember that fact with another condition in the $\Sigma \vdash h$ relation.

$$\frac{(\forall \ell, \tau.\ \Sigma(\ell) = \tau \Rightarrow \exists v.\ h(\ell) = v \wedge \Sigma; \bullet \vdash v : \tau) \quad (\exists\ \mathsf{bound}.\ \forall \ell \geqslant \mathsf{bound}.\ \ell \notin \mathsf{dom}(h))}{\Sigma \vdash h}$$

The rule requires the existence of some upper bound bound on the already-allocated locations. By construction, whenever we need to allocate a fresh location, we may choose bound, or indeed any location greater than it.

Invariants

We now have the right machinery to define an inductive invariant, namely:

$$I(h, e) = \exists \Sigma.\ \Sigma; \bullet \vdash e : \tau \wedge \Sigma \vdash h$$

We prove variants of all of the lemmas behind last chapter's type-safety proof, with a few new ones and twists on the originals. Here we give some highlights.

LEMMA 11.2 (Heap Weakening). *If* $\Sigma; \Gamma \vdash e : \tau$ *and every mapping in* Σ *is also included in* Σ'*, then* $\Sigma'; \Gamma \vdash e : \tau$*.*

LEMMA 11.3. *If* $(h, e) \rightarrow_0 (h', e')$*,* $\Sigma; \bullet \vdash e : \tau$*, and* $\Sigma \vdash h$*, then there exists* Σ' *such that* $\Sigma'; \bullet \vdash e' : \tau$*,* $\Sigma' \vdash h'$*, and* Σ' *preserves all mappings from* Σ*.*

LEMMA 11.4. *If* $\Sigma; \bullet \vdash C[e_1] : \tau$*, then there exists* τ_0 *such that* $\Sigma; \bullet \vdash e_1 : \tau_0$ *and, for all* e_2 *and* Σ'*, if* $\Sigma'; \bullet \vdash e_2 : \tau_0$ *and* Σ' *preserves mappings from* Σ*, then* $\Sigma'; \bullet \vdash C[e_2] : \tau$*.*

LEMMA 11.5 (Preservation). *If* $(h, e) \rightarrow (h', e')$*,* $\Sigma; \bullet \vdash e : \tau$*, and* $\Sigma \vdash h$*, then there exists* Σ' *such that* $\Sigma'; \bullet \vdash e' : \tau$ *and* $\Sigma' \vdash h'$*.*

11.3. Garbage Collection

Functional languages like ML and Haskell include features very similar to the mutable references that we study in this chapter. However, their execution models depart in an important way from the operational semantics we just defined: they use *garbage collection* to deallocate unused references, whereas our last semantics allows references to accumulate forever in the heap, even if it is clear that some of them will never be needed again. Worry not! We can model garbage collection with one new rule of the operational semantics, and then our type-safety proof adapts and shows that we still avoid stuckness, when the garbage collector can snatch *unreachable* locations away from us at any moment.

To define *unreachable*, we start with a way to compute the *free locations* of an expression.

$$\begin{aligned}
\mathsf{freeloc}(x) &= \varnothing \\
\mathsf{freeloc}(n) &= \varnothing \\
\mathsf{freeloc}(e_1 + e_2) &= \mathsf{freeloc}(e_1) \cup \mathsf{freeloc}(e_2) \\
\mathsf{freeloc}(\lambda x.\ e_1) &= \mathsf{freeloc}(e_1) \\
\mathsf{freeloc}(e_1\ e_2) &= \mathsf{freeloc}(e_1) \cup \mathsf{freeloc}(e_2) \\
\mathsf{freeloc}(\mathsf{new}(e_1)) &= \mathsf{freeloc}(e_1) \\
\mathsf{freeloc}(!e_1) &= \mathsf{freeloc}(e_1) \\
\mathsf{freeloc}(e_1 := e_2) &= \mathsf{freeloc}(e_1) \cup \mathsf{freeloc}(e_2) \\
\mathsf{freeloc}(\ell) &= \{\ell\}
\end{aligned}$$

Next, we define a relation to capture *which locations are reachable from some starting expression, relative to a particular heap?* For each expression e and heap h, we define $\mathcal{R}_h(e)$ as the set of locations reachable from e via h.

$$\frac{}{\ell \in \mathcal{R}_h(\ell)} \qquad \frac{h(\ell) = v \quad \ell' \in \mathcal{R}_h(v)}{\ell' \in \mathcal{R}_h(\ell)} \qquad \frac{\ell \in \mathsf{freeloc}(e) \quad \ell' \in \mathcal{R}_h(\ell)}{\ell' \in \mathcal{R}_h(e)}$$

In order, the rules say: any location reaches itself; any location reaches anywhere reachable from the value assigned to it by h; and any expression reaches anywhere reachable from any of its free locations.

Now we add one new top-level rule to the operational semantics, saying *unreachable locations may be removed at any time.*

$$\frac{\begin{array}{c}\forall \ell, v.\ \ell \in \mathcal{R}_h(e) \wedge h(\ell) = v \Rightarrow h'(\ell) = v \\ \forall \ell, v.\ h'(\ell) = v \Rightarrow h(\ell) = v \\ h' \neq h\end{array}}{(h, e) \rightarrow (h', e)}$$

Let us explain each premise in more detail. The first premise says that, going from the old heap h to the new heap h', *the value of every reachable reference is preserved*. The second premise says that *the new heap is a subheap of the original, not spontaneously adding any new mappings*. The final premise says that we have actually done some useful work: the new heap isn't just the same as the old one.

It may not be clear why we must include the last premise. The reason has to do with our formulation of type safety, by saying that programs never get *stuck*. We defined that e is *stuck* if it is not a value, but it also can't take a step. If we omitted from the garbage-collection rule the premise $h' \neq h$, then this rule would *always* apply, for any term, simply by setting $h' = h$. That is, *no* term would ever be stuck, and type safety would be meaningless! Since the rule also requires that h' be *no larger than* h (with the second premise), additionally requiring $h' \neq h$ forces h' to *shrink*, garbage-collecting at least one location. Thus, in any execution state, we can "kill time" by running garbage collection only finitely many times before we need to find some "real" step to run. More precisely, the limit on how many times we can run garbage collection in a row, starting from heap h, is $|\mathsf{dom}(h)|$, the number of locations in h.

The type-safety proof is fairly straightforward to update. We prove progress by *ignoring* the garbage-collection rule, since the existing rules were already enough to find a step for every nonvalue. A bit more work is needed to update the proof of preservation; its cases for the existing rules follow the same way as before, while we must prove a few lemmas on the way to handling the new rule.

LEMMA 11.6 (Transitivity for reachability). *If* $\mathsf{freeloc}(e_1) \subseteq \mathsf{freeloc}(e_2)$, *then* $\mathcal{R}_h(e_1) \subseteq \mathcal{R}_h(e_2)$.

LEMMA 11.7 (Irrelevance of unreachable locations for typing). *If* $\Sigma \vdash h$, $\Sigma; \Gamma \vdash e : \tau$, *then* $\Sigma'; \Gamma \vdash e : \tau$, *if we also know that, for all* ℓ *and* τ', *when* $\ell \in \mathcal{R}_h(e)$ *and* $\Sigma(\ell) = \tau'$, *it follows that* $\Sigma'(\ell) = \tau'$.

LEMMA 11.8 (Reachability sandwich). *If* $\ell \in \mathcal{R}_h(e)$, $h(\ell) = v$, *and* $\ell' \in \mathcal{R}_h(v)$, *then* $\ell' \in \mathcal{R}_h(e)$.

To extend the proof of preservation, we need to show that the strengthened invariant still holds after garbage collection. A key element is choosing the new heap typing. We pick *the restriction of the old heap typing* Σ *to the domain of the new heap* h'. That is, we drop from the heap typing all locations that have been garbage collected, preserving the types of the survivors. Some work is required to show that this strategy is sound, given the definition of reachability, but the lemmas above work out the details, leaving just a bit of bookkeeping in the preservation proof. The final safety proof then proceeds in exactly the same way as before.

Our proof here hasn't quite covered all the varieties of garbage collectors that exist. In particular, *copying collectors* may *move references to different locations*, while we only allow collectors to delete some references. It may be an edifying

exercise for the reader to extend our proof in a way that also supports reference relocation.

CHAPTER 12

Hoare Logic: Verifying Imperative Programs

We now take a step away from the last chapters in two dimensions: we switch back from functional to imperative programs, and we return to proofs of deep correctness properties, rather than mere absence of type-related crashes. Nonetheless, the essential proof structure winds up being the same, as we once again prove invariants of transition systems!

12.1. An Imperative Language with Memory

To provide us with an interesting enough playground for program verification, let's begin by defining an imperative language with an infinite mutable heap. For reasons that will become clear shortly, we do a strange bit of mixing of syntax and semantics. In certain parts of the syntax, we include *assertions* a, which are arbitrary mathematical predicates over program state, split between heaps h and variable valuations v.

$$\begin{array}{rrcl}
\text{Numbers} & n & \in & \mathbb{N} \\
\text{Variables} & x & \in & \mathsf{Strings} \\
\text{Expressions} & e & ::= & n \mid x \mid e + e \mid e - e \mid e \times e \mid *[e] \\
\text{Boolean expressions} & b & ::= & e = e \mid e < e \\
\text{Commands} & c & ::= & \mathsf{skip} \mid x \leftarrow e \mid *[e] \leftarrow e \mid c; c \\
 & & & \mid \mathsf{if}\ b\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \{a\}\mathsf{while}\ b\ \mathsf{do}\ c \mid \mathsf{assert}(a)
\end{array}$$

Beside assertions, we also have memory-read operations $*[e]$ and memory-write operations $*[e_1] \leftarrow e_2$, which are written suggestively, as if the memory were a global array named $*$. Loops have sprouted an extra assertion in their syntax, which we will actually ignore in the language semantics, but which becomes important as part of the proof technique we will learn, especially in automating it.

Expressions have a standard recursive semantics.

$$\begin{array}{rcl}
[\![n]\!](h, v) & = & n \\
[\![x]\!](h, v) & = & v(x) \\
[\![e_1 + e_2]\!](h, v) & = & [\![e_1]\!](h, v) + [\![e_2]\!](h, v) \\
[\![e_1 - e_2]\!](h, v) & = & [\![e_1]\!](h, v) - [\![e_2]\!](h, v) \\
[\![e_1 \times e_2]\!](h, v) & = & [\![e_1]\!](h, v) \times [\![e_2]\!](h, v) \\
[\![*[e]]\!](h, v) & = & h([\![e]\!](h, v)) \\
[\![e_1 = e_2]\!](h, v) & = & [\![e_1]\!](h, v) = [\![e_2]\!](h, v) \\
[\![e_1 < e_2]\!](h, v) & = & [\![e_1]\!](h, v) < [\![e_2]\!](h, v)
\end{array}$$

We finish up with a big-step semantics in the style of those we've seen before, with the added complication of threading a heap through.

Encoding

$$\frac{}{(h, v, \mathsf{skip}) \Downarrow (h, v)} \quad \frac{}{(h, v, x \leftarrow e) \Downarrow (h, v[x \mapsto [\![e]\!](h, v)])}$$

$$\frac{}{(h, v, *[e_1] \leftarrow e_2) \Downarrow (h[[\![e_1]\!](h, v) \mapsto [\![e_2]\!](h, v)], v)}$$

$$\frac{(h, v, c_1) \Downarrow (h_1, v_1) \quad (h_1, v_1, c_2) \Downarrow (h_2, v_2)}{(h, v, c_1; c_2) \Downarrow (h_2, v_2)}$$

$$\frac{[\![b]\!](h, v) \quad (h, v, c_1) \Downarrow (h', v')}{(h, v, \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \Downarrow (h', v')} \quad \frac{\neg[\![b]\!](h, v) \quad (h, v, c_2) \Downarrow (h', v')}{(h, v, \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \Downarrow (h', v')}$$

$$\frac{[\![b]\!](h, v) \quad (h, v, c; \{I\}\mathsf{while}\ b\ \mathsf{do}\ c) \Downarrow (h', v')}{(h, v, \{I\}\mathsf{while}\ b\ \mathsf{do}\ c) \Downarrow (h', v')} \quad \frac{\neg[\![b]\!](h, v)}{(h, v, \{I\}\mathsf{while}\ b\ \mathsf{do}\ c) \Downarrow (h, v)}$$

$$\frac{a(h, v)}{(h, v, \mathsf{assert}(a)) \Downarrow (h, v)}$$

Reasoning directly about operational semantics can get tedious, so let's develop some machinery for proving program correctness automatically.

12.2. Hoare Triples

Much as we did with type systems, we define a syntactic predicate and prove it sound once and for all. Afterward, we can automatically show that particular programs and their specifications inhabit the predicate. This time, predicate instances will be written like $\{P\}c\{Q\}$, with c the command being verified, P its *precondition* (assumption about the program state before we start running c), and Q its *postcondition* (obligation about the program state after c finishes). We call any such fact a *Hoare triple*, and the overall predicate is an instance of *Hoare logic*.

Encoding

A first rule for skip is easy: anything that was true before is also true after.

$$\frac{}{\{P\}\mathsf{skip}\{P\}}$$

A rule for assignment is slightly more involved: to state what we know is true after, we recall that there existed a prestate satisfying the precondition, which then evolved into the poststate in the expected way.

$$\frac{}{\{P\}x \leftarrow e\{\lambda(h, v).\ \exists v'.\ P(h, v') \wedge v = v'[x \mapsto [\![e]\!](h, v')]\}}$$

The memory-write command is treated symmetrically.

$$\frac{}{\{P\}*[e_1] \leftarrow e_2\{\lambda(h, v).\ \exists h'.\ P(h', v) \wedge h = h'[[\![e_1]\!](h', v) \mapsto [\![e_2]\!](h', v)]\}}$$

To model sequencing, we thread predicates through in an intuitive way.

$$\frac{\{P\}c_1\{Q\} \quad \{Q\}c_2\{R\}}{\{P\}c_1; c_2\{R\}}$$

For conditional statements, we start from the basic approach of sequencing, adding two twists. First, since the two subcommands run after different outcomes of the test expression, we extend their preconditions. Second, since we may reach

the end of the command after running either subcommand, we take the disjunction of their postconditions.

$$\frac{\{\lambda s.\ P(s) \wedge [\![b]\!](s)\}c_1\{Q_1\} \quad \{\lambda s.\ P(s) \wedge \neg[\![b]\!](s)\}c_2\{Q_2\}}{\{P\}\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\{\lambda s.\ Q_1(s) \vee Q_2(s)\}}$$

Invariants

Coming to loops, we at last have a purpose for the assertion annotated on each one. We call those assertions *loop invariants*; one of these is meant to be true every time a loop iteration begins. We will try to avoid confusion with the more fundamental concept of invariant for transition systems, though in fact the two are closely related formally, which we will see in the last section of this chapter. Essentially, the loop invariant gives the *induction hypothesis* that makes the program correctness proof go through. We encapsulate the induction reasoning once and for all, in the proof of soundness for Hoare triples. To verify an individual program, it is only necessary to prove the premises of the rule, which we give now.

$$\frac{(\forall s.\ P(s) \Rightarrow I(s)) \quad \{\lambda s.\ I(s) \wedge [\![b]\!](s)\}c\{I\}}{\{P\}\{I\}\mathsf{while}\ b\ \mathsf{do}\ c\{\lambda s.\ I(s) \wedge \neg[\![b]\!](s)\}}$$

In words: the loop invariant is true when we begin the loop, and every iteration preserves the invariant, given the extra knowledge that the loop test succeeded. If the loop finishes, we know that the invariant is still true, but now the test is false.

The final command-specific rule, for assertions, is a bit anticlimactic. The precondition is carried over as postcondition, if it is strong enough to prove the assertion.

$$\frac{\forall s.\ P(s) \Rightarrow I(s)}{\{P\}\mathsf{assert}(I)\{P\}}$$

One more essential rule remains, this time not specific to any command form. The rules we've given deduce specific kinds of precondition-postcondition pairs. For instance, the skip rule forces the precondition and postcondition to match. However, we expect to be able to prove $\{\lambda(h,v).\ v(x) > 0\}\mathsf{skip}\{\lambda(h,v).\ v(x) \geqslant 0\}$, because the postcondition is *weaker* than the precondition, meaning the precondition implies the postcondition. Alternatively, the precondition is *stronger* than the postcondition, because the precondition keeps all restrictions from the postcondition while adding new ones. Hoare Logic's *rule of consequence* allows us to build a new Hoare triple from an old one by *strengthening the precondition* and *weakening the postcondition*.

$$\frac{\{P\}c\{Q\} \quad (\forall s.\ P'(s) \Rightarrow P(s)) \quad (\forall s.\ Q(s) \Rightarrow Q'(s))}{\{P'\}c\{Q'\}}$$

These rules together are *complete*, in the sense that any intuitively correct precondition-postcondition pair for a command is provable. Here we only go into detail on a proof of the dual property, *soundness*.

LEMMA 12.1. *Assume the following fact: Together,* $(h,v,c) \Downarrow (h',v')$*,* $I(h,v)$*, and* $[\![b]\!](h,v)$ *imply* $I(h',v')$*. Then, given* $(h,v,\{I\}\mathsf{while}\ b\ \mathsf{do}\ c) \Downarrow (h',v')$*, it follows that* $I(h',v')$ *and* $\neg[\![b]\!](h',v')$*.*

PROOF. By induction on the derivation of $(h,v,\{I\}\mathsf{while}\ b\ \mathsf{do}\ c) \Downarrow (h',v')$. □

That lemma encapsulates once and for all the use of induction in reasoning about the many iterations of loops.

THEOREM 12.2 (Soundness of Hoare logic). *If* $\{P\}c\{Q\}$, $(h, v, c) \Downarrow (h', v')$, *and* $P(h, v)$, *then* $Q(h', v')$.

PROOF. By induction on the derivation of $\{P\}c\{Q\}$ and inversion on the derivation of $(h, v, c) \Downarrow (h', v')$, appealing to Lemma 12.1 in the appropriate case. □

We leave concrete example derivations to the accompanying Coq code, as that level of fiddly detail deserves to be machine-checked. Note that there is a rather effective automated proof procedure lurking behind the rules introduced in this section: To prove a Hoare triple, first try applying the rule associated with its top-level syntax-tree constructor (e.g., assignment or loop rule). If the conclusion of that rule does unify with the goal, apply the rule and proceed recursively on its premises. Otherwise, apply the rule of consequence to replace the postcondition with one matching that from the matching rule; note that all rules accept arbitrarily shaped preconditions, so we don't actually need to do work to massage the precondition. After a step like this one, it is guaranteed that the "fundamental" rule now applies.

This process creates a pile of side conditions to be proved by other means, corresponding to the assertion implications generated by the rules for loops, assertions, and consequence. Many real-world tools based on Hoare logic discharge such goals using solvers for satisfiability modulo theories, otherwise known as SMT solvers. The accompanying Coq code just uses a modest Coq automation tactic definition building on the proof steps we have been using all along. It is not complete by any means, but it does surprisingly well in the examples we step through, of low to moderate complexity.

Abstraction

Modularity

Before closing our discussion of the basics of Hoare logic, let's consider how it brings to bear some more of the general principles that we have met before. A command's precondition and postcondition serve as an *abstraction* of the command: it is safe to model a command with its specification, if it has been proved using a Hoare triple. Furthermore, the Hoare rules themselves take advantage of *modularity* to analyze subcommands separately, mediating between them using only the specifications. The implementation details of a subcommand don't matter for any other subcommands in the program, so long as that subcommand has been connected to a specification that preserves enough information about its behavior. It is an art to choose the right specification for each piece of a program. Detailed specifications minimize the chance that some other part of the program winds up unprovable, despite its correctness, but more detailed specifications also tend to be harder to prove in the first place.

12.3. Small-Step Semantics

Last section's soundness theorem only lets us draw conclusions about programs that terminate. We call such guarantees *partial correctness*. Other forms of Hoare triples guarantee *total correctness*, which includes termination. However, sometimes programs aren't meant to terminate, yet we still want to gain confidence about their behavior. To that end, we first give a small-step semantics for the same programming language. Then we prove a different soundness theorem for the same Hoare-triple predicate, showing that it also implies a useful invariant for programs as transition systems.

Encoding

The small-step relation is quite similar to the one from last chapter, though now our states are triples (h, v, c), of heap h, variable valuation v, and command c.

$$\frac{}{(h, v, x \leftarrow e) \to (h, v[x \mapsto [\![e]\!](h, v)], \mathsf{skip})}$$

$$\frac{}{(h, v, *[e_1] \leftarrow e_2) \to (h[[\![e_1]\!](h, v) \mapsto [\![e_2]\!](h, v)], v, \mathsf{skip})}$$

$$\frac{}{(h, v, \mathsf{skip}; c_2) \to (h, v, c_2)} \quad \frac{(h, v, c_1) \to (h', v', c_1')}{(h, v, c_1; c_2) \to (h', v', c_1'; c_2)}$$

$$\frac{[\![b]\!](h, v)}{(h, v, \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \to (h, v, c_1)} \quad \frac{\neg[\![b]\!](h, v)}{(h, v, \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \to (h, v, c_2)}$$

$$\frac{[\![b]\!](h, v)}{(h, v, \{I\}\mathsf{while}\ b\ \mathsf{do}\ c) \to (h, v, c; \{I\}\mathsf{while}\ b\ \mathsf{do}\ c)} \quad \frac{\neg[\![b]\!](h, v)}{(h, v, \{I\}\mathsf{while}\ b\ \mathsf{do}\ c) \to (h, v, \mathsf{skip})}$$

$$\frac{a(h, v)}{(h, v, \mathsf{assert}(a)) \to (h, v, \mathsf{skip})}$$

12.4. Transition-System Invariants from Hoare Triples

Even an infinite-looping program must satisfy its assert commands, every time it passes one of them. For that reason, it's interesting to consider how to show that a command never gets stuck on a false assertion. We work up to that result with a few intermediate ones. First, we define *stuck* much the same way as in the last two chapters: a state (h, v, c) is stuck if c is not skip, but there is also nowhere to step to from this state. An example of a stuck state would be one beginning with an assert of an assertion that does not hold on h and v. In fact, we can prove that any other state is unstuck, though we won't bother here.

LEMMA 12.3 (Progress). *If $\{P\}c\{Q\}$ and $P(h, v)$, then (h, v, c) is unstuck.*

PROOF. By induction on the derivation of $\{P\}c\{Q\}$. □

LEMMA 12.4. *If $\{P\}\mathsf{skip}\{Q\}$, then $\forall s.\ P(s) \Rightarrow Q(s)$.*

PROOF. By induction on the derivation of $\{P\}\mathsf{skip}\{Q\}$. □

LEMMA 12.5 (Preservation). *If $\{P\}c\{Q\}$, $(h, v, c) \to (h', v', c')$, and $P(h, v)$, then $\{\lambda s.\ s = (h', v')\}c'\{Q\}$.*

PROOF. By induction on the derivation of $\{P\}c\{Q\}$, appealing to Lemma 12.4 in one case. Note how we conclude a very specific precondition, forcing exact state equality with the one we have stepped to. □

Invariants

THEOREM 12.6 (Invariant Safety). *If $\{P\}c\{Q\}$ and $P(h, v)$, then unstuckness is an invariant for the small-step transition system starting at (h, v, c).*

PROOF. First we weaken the invariant to $I(h, v, c) = \{\lambda s.\ s = (h, v)\}c\{\lambda\_.\ \top\}$. That is, we focus in on the most specific applicable precondition, and we forget everything that the postcondition was recording for us. Note that postconditions are still an essential part of Hoare triples for this proof, but we have already done our detailed analysis of them in the earlier lemmas. Lemma 12.3 gives the needed implication from the new invariant to the old.

Next, we apply invariant induction, whose base case follows trivially. The induction step follows by Lemma 12.5. □

CHAPTER 13

Deep Embeddings, Shallow Embeddings, and Options in Between

So far, in this book, we have followed the typographic conventions of ordinary mathematics and logic, as they would be worked out on whiteboards. In parallel, we have mechanized all of the definitions and proofs in Coq. Often little tidbits of encoding challenge show up in mechanizing the proofs. As formal languages get more complex, it becomes more and more important to choose the right encoding. For instance, in the previous chapter, we repeatedly jumped through hoops to track the local variables of programs, threading variable valuations v throughout everything. Coq already has built into it a respectable notion of variables; can we somehow reuse that mechanism, rather than roll our own new one? This chapter gives a "yes" answer, working toward redefining last chapter's Hoare logic in a lighter-weight manner, along the way introducing some key terminology that is used to classify encoding choices.

Since whiteboard math doesn't usually bother with encoding details, here we must break with our convention of using only standard notation in the book. Instead, we will use notation closer to literal Coq code, and, in fact, more of the technical action than usual is only found in the accompanying Coq source file.

13.1. The Basics

Recall some terminology introduced in Section 2.5: every formal proof is carried out in some *metalanguage*, which, in our case, is Coq's logic and programming language called Gallina. A syntactic language that we formalize is called an *object language*. Often it is convenient to do reasoning without any particular object language, as in this simple arithmetic function that can be defined directly in Gallina.

$$\mathsf{foo} \quad = \quad \lambda(x, y).\ \mathsf{let}\ u = x + y\ \mathsf{in}\ \mathsf{let}\ v = u \times y\ \mathsf{in}\ u + v$$

However, it is difficult to prove some important facts about terms encoded directly in the metalanguage. For instance, we can't easily do induction over the syntax of all such terms. To allow that kind of induction, we can define an object language inductively.

Encoding

$$\begin{array}{rcl} \mathsf{Const} & : & \mathbb{N} \to \mathsf{exp} \\ \mathsf{Var} & : & \mathbb{V} \to \mathsf{exp} \\ \mathsf{Plus} & : & \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp} \\ \mathsf{Times} & : & \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp} \\ \mathsf{Let} & : & \mathbb{V} \to \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp} \end{array}$$

That last example program, with implicit *free variables* x and y, may now be redefined in the exp type.

$$\begin{array}{rcl} \mathsf{foo}' & = & \mathsf{Let}\ (\mathsf{Var}\ \text{“u”})\ (\mathsf{Plus}\ (\mathsf{Var}\ \text{“x”})\ (\mathsf{Var}\ \text{“y”}))\ (\mathsf{Let}\ (\mathsf{Var}\ \text{“v”}) \\ & & \quad (\mathsf{Times}\ (\mathsf{Var}\ \text{“u”})\ (\mathsf{Var}\ \text{“y”}))\ (\mathsf{Plus}\ (\mathsf{Var}\ \text{“u”})\ (\mathsf{Var}\ \text{“v”}))) \end{array}$$

As in Chapter 4, we can define a recursive interpreter, mapping exp programs and variable valuations to numbers. Using that interpreter, we can prove equivalence of foo and foo′.

We say that foo uses a *shallow embedding*, because it is coded directly in the metalanguage, with no extra layer of syntax. Conversely, foo′ uses a *deep embedding*, since it goes via the inductively defined exp type.

These extremes are not our only options. In higher-order logics like Coq's, we may also choose what might be called *mixed embeddings*, which define syntax-tree types that allow some use of general functions from the metalanguage. Here's an example, as an alternative definition of exp.

Encoding

$$\begin{array}{rcl} \mathsf{Const} & : & \mathbb{N} \to \mathsf{exp} \\ \mathsf{Var} & : & \mathbb{V} \to \mathsf{exp} \\ \mathsf{Plus} & : & \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp} \\ \mathsf{Times} & : & \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp} \\ \mathsf{Let} & : & \mathsf{exp} \to (\mathbb{N} \to \mathsf{exp}) \to \mathsf{exp} \end{array}$$

The one change is in the type of the Let constructor, where now no variable name is given, and instead *the body of the "let" is represented as a Gallina function from numbers to expressions*. The intent is that the body is called on the number that results from evaluating the first expression. This style is called *higher-order abstract syntax*. Though that term is often applied to a more specific instance of the technique, which is not exactly the one used here, we will not be so picky.

As an illustration of the technique in action, here's our third encoding of the simple example program.

$$\begin{array}{rcl} \mathsf{foo}'' & = & \mathsf{Let}\ (\mathsf{Plus}\ (\mathsf{Var}\ \text{“x”})\ (\mathsf{Var}\ \text{“y”}))\ (\lambda u. \\ & & \quad \mathsf{Let}\ (\mathsf{Times}\ (\mathsf{Const}\ u)\ (\mathsf{Var}\ \text{“y”}))\ (\lambda v. \\ & & \quad\quad \mathsf{Plus}\ (\mathsf{Const}\ u)\ (\mathsf{Const}\ v))) \end{array}$$

With a bit of subtlety, we can define an interpreter for this language, too.

$$\begin{array}{rcl} [\![\mathsf{Const}\ n]\!]v & = & n \\ [\![\mathsf{Var}\ x]\!]v & = & v(x) \\ [\![\mathsf{Plus}\ e_1\ e_2]\!]v & = & [\![e_1]\!]v + [\![e_2]\!]v \\ [\![\mathsf{Times}\ e_1\ e_2]\!]v & = & [\![e_1]\!]v \times [\![e_2]\!]v \\ [\![\mathsf{Let}\ e_1\ e_2]\!]v & = & [\![e_2([\![e_1]\!]v)]\!]v \end{array}$$

Note how, in the Let case, since the body e_2 is a function, before evaluating it, we call it on the result of evaluating e_1. This language would actually be sufficient even if we removed the Var constructor and the v argument of the interpreter. Coq's normal variable binding is enough to let us model interesting programs and prove things about them by induction on syntax.

It is important here that Coq's induction principles give us useful induction hypotheses, for constructors whose recursive arguments are functions. The second

argument of Let above is an example. When we do induction on expression syntax to establish $\forall e.\ P(e)$, the case for Let $e_1\ e_2$ includes two induction hypotheses. The first one is standard: $P(e_1)$. The second one is more interesting: $\forall n : \mathbb{N}.\ P(e_2(n))$. That is, the theorem holds on all results of applying body e_2 to arguments.

13.2. A Mixed Embedding for Hoare Logic

This general strategy also applies to modeling imperative languages like the one from last chapter. We can define a polymorphic type family cmd of commands, indexed by the type of value that a command is meant to return.

Encoding

$$\begin{array}{rcl}
\mathsf{Return} & : & \forall \alpha.\ \alpha \to \mathsf{cmd}\ \alpha \\
\mathsf{Bind} & : & \forall \alpha, \beta.\ \mathsf{cmd}\ \beta \to (\beta \to \mathsf{cmd}\ \alpha) \to \mathsf{cmd}\ \alpha \\
\mathsf{Read} & : & \mathbb{N} \to \mathsf{cmd}\ \mathbb{N} \\
\mathsf{Write} & : & \mathbb{N} \to \mathbb{N} \to \mathsf{cmd\ unit}
\end{array}$$

We use notation $x \leftarrow c_1; c_2$ as shorthand for Bind c_1 $(\lambda x.\ c_2)$, making it possible to write some very natural-looking programs in this type. Here are two examples.

$$\begin{array}{rcl}
\mathsf{array_max}(0, a) & = & \mathsf{Return}\ a \\
\mathsf{array_max}(i+1, a) & = & v \leftarrow \mathsf{Read}\ i; \mathsf{array_max}\ i\ (\max(v, a))
\end{array}$$

$$\begin{array}{rcl}
\mathsf{increment_all}(0) & = & \mathsf{Return}\ () \\
\mathsf{increment_all}(i+1) & = & v \leftarrow \mathsf{Read}\ i; _ \leftarrow \mathsf{Write}\ i\ (v+1); \mathsf{increment_all}\ i
\end{array}$$

Function array_max computes the highest value found in the first i slots of memory, using an accumulator a. Function increment_all adds 1 to every one of the first i memory slots.

Note that we are not writing programs directly as syntax trees, but rather working with recursive functions that *compute syntax trees*. We are able to do so despite the fact that we built no support for recursion into the cmd type family. Likewise, we didn't need to build in any support for max, addition, or any of the other operations that are easy to code up in Gallina.

It is straightforward to implement an interpreter for this object language, where each command's interpretation maps input heaps to pairs of output heaps and results. Note that we have no need for an explicit variable valuation.

$$\begin{array}{rcl}
[\![\mathsf{Return}\ v]\!]h & = & (h, v) \\
[\![\mathsf{Bind}\ c_1\ c_2]\!]h & = & \mathsf{let}\ (h', v) = [\![c_1]\!]h\ \mathsf{in}\ [\![c_2(v)]\!]h' \\
[\![\mathsf{Read}\ a]\!]h & = & (h, h(a)) \\
[\![\mathsf{Write}\ a\ v]\!]h & = & (h[a \mapsto v], ())
\end{array}$$

We can also define a syntactic Hoare-logic relation for this type, where preconditions are predicates over initial heaps, and postconditions are predicates over *result values* and final heaps.

$$\frac{}{\{P\}\mathsf{Return}\ v\{\lambda r, h.\ P(h) \wedge r = v\}} \qquad \frac{\{P\}c_1\{Q\} \quad \forall r.\ \{Q(r)\}c_2(r)\{R\}}{\{P\}\mathsf{Bind}\ c_1\ c_2\{R\}}$$

$$\frac{}{\{P\}\mathsf{Read}\ a\{\lambda r, h.\ P(h) \wedge r = h(a)\}} \qquad \frac{}{\{P\}\mathsf{Write}\ a\ v\{\lambda r, h.\ \exists h'.\ P(h') \wedge h = h'[a \mapsto v]\}}$$

$$\frac{\{P\}c\{Q\} \quad (\forall h.\ P'(h) \Rightarrow P(h)) \quad (\forall r, h.\ Q(r,h) \Rightarrow Q'(r,h))}{\{P'\}c\{Q'\}}$$

Much of the details are the same as last chapter, including in a rule of consequence at the end. The most interesting new wrinkle is in the rule for Bind, where the premise about the body command c_2 starts with universal quantification over all possible results r of executing c_1. That result is passed off, via function application, both to the body c_2 and to Q, which serves as the postcondition of c_1 and the precondition of c_2.

This Hoare logic can be used to verify the two example programs from earlier in this section; see the accompanying Coq code for details. We also have a standard soundness theorem.

THEOREM 13.1. *If $\{P\}c\{Q\}$ and $P(h)$ for some heap h, then let $(h', r) = [\![c]\!]h$. It follows that $Q(r, h')$.*

13.3. Adding More Effects

We can continue to enhance our object language with different kinds of side effects that are not supported natively by Gallina. First, we add *nontermination*, in the form of unbounded loops. For a type α, we define $\mathbb{O}(\alpha)$ as the type of *loop-body outcomes*, either $\mathsf{Done}(a)$ to indicate that the loop should terminate or $\mathsf{Again}(a)$ to indicate that the loop should keep running. Our loops are functional, maintaining accumulators as they run, and the a argument gives the latest accumulator value in each case. So we add this constructor:

$$\mathsf{Loop} \quad : \quad \forall \alpha.\ \alpha \to (\alpha \to \mathsf{cmd}\ (\mathbb{O}(\alpha))) \to \mathsf{cmd}\ \alpha$$

Here's an example of looping in action, in a program that returns the address of the first occurrence of a value in memory, or loops forever if that value is not found in the whole infinite memory.

$$\mathsf{index\_of}(n) \quad = \quad \mathsf{Loop}\ 0\ (\lambda i.\ v \leftarrow \mathsf{Read}\ i; \mathsf{if}\ v = n\ \mathsf{then}\ \mathsf{Return}\ (\mathsf{Done}(i))\ \mathsf{else}\ \mathsf{Return}\ (\mathsf{Again}(i+1)))$$

With the addition of nontermination, it's no longer straightforward to write an interpreter for the language. Instead, we implement a small-step operational semantics $\rightarrow$; see the accompanying Coq code for details. We build an extended Hoare logic, keeping all the rules from last section and adding this new one. Like before, it is parameterized on a loop invariant, but now the loop invariant takes a loop-body outcome as parameter.

Invariants

$$\frac{\forall a.\ \{I(\mathsf{Again}(a))\}c(a)\{I\}}{\{I(\mathsf{Again}(i))\}\mathsf{Loop}\ i\ c\{\lambda r.\ I(\mathsf{Done}(r))\}\}}$$

This new Hoare logic is usable to verify the example program from above and many more, and we can also prove a soundness theorem. The operational semantics gives us the standard way of interpreting our programs as transition systems, with states (c, h).

Invariants

THEOREM 13.2. *If $\{P\}c\{Q\}$ and $P(h)$ for some heap h, then it is an invariant of (c, h) that, if the command ever becomes* Return *r in a heap h', then $Q(r, h')$. That is, if the program terminates, the postcondition is satisfied.*

We can add a further side effect to the language: *exceptions*. Actually, we stick to a simple variant of this classic side effect, where there is just one exception, and

it cannot be caught. We associate this exception with *program failure*, and the Hoare logic will ensure that programs never actually fail.

The extension to program syntax is easy:

$$\mathsf{Fail} \quad : \quad \forall \alpha.\ \mathsf{cmd}\ \alpha$$

That is, a failing program can be considered to return any result type, since it will never actually return normally, instead throwing an uncatchable exception.

The operational semantics is also easily extended to signal failures, with a new special system state called Failed. We also add this Hoare-logic rule.

$$\frac{}{\{\lambda\_.\ \bot\}\mathsf{Fail}\{\lambda\_, \_.\ \bot\}}$$

That is, failure can only be verified against an unsatisfiable precondition, so that we know that the failure is unreachable.

With this extension, we can prove a soundness-theorem variant, capturing the impossibility of failure.

Invariants

THEOREM 13.3. *If* $\{P\}c\{Q\}$ *and* $P(h)$ *for some heap* h*, then it is an invariant of* (c, h) *that the state never becomes* Failed*.*

Note that this version of the theorem still tells us interesting things about programs that run forever. It is easy to implement runtime assertion checking with code that performs some test and runs Fail if the test does not pass. An infinite-looping program may perform such tests infinitely often, and we learn that none of the tests ever fail.

The accompanying Coq code demonstrates another advantage of this mixed-embedding style: we can extract our programs to OCaml and run them efficiently. That is, rather than using functional programming to implement our three kinds of side effects, we implement them directly with OCaml's mutable heap, unbounded recursion, and exceptions, respectively. As a result, our extracted programs achieve the asymptotic performance that we would expect, thinking of them as C-like code, where interpreters in a pure functional language like Gallina would necessarily add at least an extra logarithmic factor in the modeling of unboundedly growing heaps.

CHAPTER 14

Separation Logic

In our Hoare-logic examples so far, we have intentionally tread lightly when it comes to the potential aliasing of pointer variables in a program. Generally, we have only worked with, for instance, a single array at a time. Reasoning about multi-array programs usually depends on the fact that the arrays don't overlap in memory at all. Things are even more complicated with linked data structures, like linked lists and trees, which we haven't even attempted up to now. However, by using *separation logic*, a popular variant of Hoare logic, we will find it quite pleasant to prove programs that used linked structures, with no need for explicit reasoning about aliasing, assuming that we keep all of our data structures disjoint from each other through simple coding patterns.

14.1. An Object Language with Dynamic Memory Allocation

Before we get into proofs, let's fix a mixed-embedding object language.

$$\begin{array}{rrcl} \text{Commands} & c & ::= & \mathsf{Return}\ v \mid x \leftarrow c; c \mid \mathsf{Loop}\ i\ f \mid \mathsf{Fail} \\ & & & \mid \mathsf{Read}\ n \mid \mathsf{Write}\ n\ n \mid \mathsf{Alloc}\ n \mid \mathsf{Free}\ n\ n \end{array}$$

A small-step operational semantics explains what these commands mean.

$$\frac{(h, c_1) \to (h', c_1')}{(h, x \leftarrow c_1; c_2(x)) \to (h', x \leftarrow c_1'; c_2(x))} \quad \frac{}{(h, x \leftarrow \mathsf{Return}\ v; c(x)) \to (h, c(v))}$$

$$\frac{}{(h, \mathsf{Loop}\ i\ f) \to (h, x \leftarrow f(i); \mathsf{match}\ x\ \mathsf{with}\ \mathsf{Done}(a) \Rightarrow \mathsf{Return}\ a \mid \mathsf{Again}(a) \Rightarrow \mathsf{Loop}\ a\ f)}$$

$$\frac{h(a) = v}{(h, \mathsf{Read}\ a) \to (h, \mathsf{Return}\ v)} \quad \frac{h(a) = v}{(h, \mathsf{Write}\ a\ v') \to (h[a \mapsto v'], \mathsf{Return}\ ())}$$

$$\frac{\mathsf{dom}(h) \cap [a, a+n) = \varnothing}{(h, \mathsf{Alloc}\ n) \to (h[a \mapsto 0^n], \mathsf{Return}\ a)} \quad \frac{}{(h, \mathsf{Free}\ a\ n) \to (h - [a, a+n), \mathsf{Return}\ ())}$$

A few remarks about the last four rules: The basic Read and Write operations now get *stuck* when accessing unmapped addresses. The premise of the rule for Alloc enforces that address a denotes a currently unmapped memory region of size n. We use a variety of convenient notations that we won't define in detail here, referring instead to the accompanying Coq code. Another notation uses 0^n to refer informally to a sequence of n zeroes to write into memory. Similarly, the conclusion of the Free rule unmaps a whole size-n region, starting at a. We could also have chosen to enforce in this rule that the region starts out as mapped into h.

14.2. Assertion Logic

Separation logic is based on two big ideas. The first one has to do with the *assertion logic*, which we use to write invariants; while the second one has to do with the *program logic*, which we use to prove that programs satisfy specifications. The assertion logic is based on predicates over *partial memories*, or finite maps from addresses to stored values. Because they are finite, they omit infinitely many addresses, and it is crucial that we are able to describe heaps that intentionally leave addresses out of their domains. Informally, a predicate *claims ownership* of addresses in the domains of matching heaps.

We can describe the connectives of separation logic in terms of the sets of partial heaps that they accept.

$$\begin{array}{rcl} \mathsf{emp} & = & \{\bullet\} \\ p \mapsto v & = & \{\bullet[p \mapsto v]\} \\ [\phi] & = & \{h \mid \phi \wedge h = \bullet\} \\ \exists x.\ P(x) & = & \{h \mid \exists x.\ h \in P(x)\} \\ P * Q & = & \{h_1 \uplus h_2 \mid h_1 \in P \wedge h_2 \in Q\} \end{array}$$

The formula emp accepts only the empty heap, while formula $p \mapsto v$ accepts only the heap whose only address is p, mapped to value v. We overload the $\mapsto$ operator in that second line above, to denote "points-to" on the lefthand side of the equality and finite-map overriding on the righthand side. Notation $[\phi]$ is *lifting* a *pure* (i.e., regular old mathematical) proposition ϕ into an assertion, enforcing both that the heap is empty and that ϕ is true. We also adapt the normal existential quantifier to this setting.

The essential definition is the last one, of the *separating conjunction* $*$. We use the notation $h_1 \uplus h_2$ for *disjoint union* of heaps h_1 and h_2, implicitly enforcing $\mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \varnothing$. The intuition of separating conjunction is that we *partition* the overall heap into two subheaps, each of which matches one of the respective conjuncts P and Q. This connective implicitly enforces *lack of aliasing*, leading to separation logic's famous conciseness of specifications that combine data structures.

We can also define natural comparison operators between assertions, overloading the usual notations for equivalence and implication of propositions.

$$\begin{array}{rcl} P \Leftrightarrow Q & = & \forall h.\ h \in P \Leftrightarrow h \in Q \\ P \Rightarrow Q & = & \forall h.\ h \in P \Rightarrow h \in Q \end{array}$$

The core connectives satisfy a number of handy algebraic laws. Here is a sampling (where we note that the first one takes advantage of the overloading of implication $\Rightarrow$ in two different senses).

$$\frac{\phi \Rightarrow (P \Rightarrow Q)}{P * [\phi] \Rightarrow Q} \qquad \frac{\phi \quad P \Rightarrow Q}{P \Rightarrow Q * [\phi]} \qquad \frac{\phi}{P \Leftrightarrow [\phi] * P}$$

$$\frac{}{P * Q \Leftrightarrow Q * P} \qquad \frac{}{P * (Q * R) \Leftrightarrow (P * Q) * R} \qquad \frac{P_1 \Rightarrow P_2 \quad Q_1 \Rightarrow Q_2}{P_1 * Q_1 \Rightarrow P_2 * Q_2}$$

$$\frac{}{(P * \exists x.\ Q(x)) \Leftrightarrow \exists x.\ P * Q(x)} \quad \frac{\forall x.\ P(x) \Rightarrow Q}{(\exists x.\ P(x)) \Rightarrow Q} \quad \frac{P \Rightarrow Q(v)}{P \Rightarrow \exists x.\ Q(x)}$$

This set of algebraic laws has a very special consequence: it supports automated proof of implications by *cancellation*, where we repeatedly "cross out" matching subformulas on the two sides of the arrow. Consider this example formula that we might want to prove.

$$(\exists q.\ p \mapsto q * \exists r.\ q \mapsto r * r \mapsto 0) \Rightarrow (\exists a.\ \exists b.\ \exists c.\ b \mapsto c * p \mapsto a * a \mapsto b)$$

First, the laws above allow us to bubble all quantifiers to the fronts of formulas.

$$(\exists q, r.\ p \mapsto q * q \mapsto r * r \mapsto 0) \Rightarrow (\exists a, b, c.\ b \mapsto c * p \mapsto a * a \mapsto b)$$

Next, all $\exists$ to the left can be replaced with fresh free variables, while all $\exists$ to the right can be replaced with fresh *unification variables*, whose values, in terms of the free-variable values, we can deduce in the course of the proof.

$$p \mapsto q * q \mapsto r * r \mapsto 0 \Rightarrow ?b \mapsto ?c\ *\ p \mapsto ?a * ?a \mapsto ?b$$

Next, we find matching subformulas to *cancel*. We start by matching $p \mapsto q$ with $p \mapsto ?a$, learning that $?a = q$ and reducing to the following formula. This crucial step relies on the three key properties of $*$, given in the second row of rules above: commutativity, associativity, and cancellativity.

$$q \mapsto r * r \mapsto 0 \Rightarrow ?b \mapsto ?c\ *\ q \mapsto ?b$$

We run another cancellation step of $q \mapsto r$ against $q \mapsto ?b$, learning $?b = r$.

$$r \mapsto 0 \Rightarrow\ r \mapsto ?c$$

Now we can finish the proof by reflexivity of $\Rightarrow$, learning $?c = 0$.

14.3. Program Logic

We use our automatic cancellation procedure to discharge some of the premises from the rules of the program logic, which we present now. First, here are the rules that are (almost) exactly the same as from last chapter.

$$\frac{}{\{P\}\mathsf{Return}\ v\{\lambda r.\ P * [r = v]\}} \quad \frac{\{P\}c_1\{Q\} \quad (\forall r.\ \{Q(r)\}c_2(r)\{R\})}{\{P\}x \leftarrow c_1; c_2(x)\{R\}}$$

$$\frac{\forall a.\ \{I(\mathsf{Again}(a))\}f(a)\{I\}}{\{I(\mathsf{Again}(i))\}\mathsf{Loop}\ i\ f\{\lambda r.\ I(\mathsf{Done}(r))\}} \quad \frac{}{\{[\bot]\}\mathsf{Fail}\{\lambda\_.\ [\bot]\}}$$

$$\frac{\{P\}c\{Q\} \quad P' \Rightarrow P \quad \forall r.\ Q(r) \Rightarrow Q'(r)}{\{P'\}c\{Q'\}}$$

More interesting are the rules for primitive memory operations. First, we have the rule for Read.

$$\frac{}{\{\exists v.\ a \mapsto v * R(v)\}\mathsf{Read}\ a\{\lambda r.\ a \mapsto r * R(r)\}}$$

In words: before reading from address a, it must be the case that a points to some value v, and predicate $R(v)$ records what else we know about the memory at that point. Afterward, we know that a points to the result r of the read operation, and R is still present. We call R a *frame predicate*, recording what we know about

parts of memory that the command does not touch directly. We might also say that the *footprint* of this command is the singleton set $\{a\}$. In general, frame predicates record preserved facts about addresses outside a command's footprint. The next few rules don't have frame predicates baked in; we finish with a rule that adds them back, in a generic way for arbitrary Hoare triples.

$$\overline{\{\exists v.\ a \mapsto v\}\mathsf{Write}\ a\ v'\{\lambda\_.\ a \mapsto v'\}}$$

This last rule, for Write, is even simpler. We see a straightforward illustration of overwriting a's old value v with the new value v'.

$$\overline{\{\mathsf{emp}\}\mathsf{Alloc}\ n\{\lambda r.\ r \mapsto 0^n\}} \quad \overline{\{a \mapsto ?^n\}\mathsf{Free}\ a\ n\{\lambda\_.\ \mathsf{emp}\}}$$

The rules for allocation and deallocation deploy a few notations that we don't explain in detail here, with 0^n for sequences of n zeroes and $?^n$ for sequences of n arbitrary values.

The next rule, the *frame rule*, gives the second key idea of separation logic, supporting the *small-footprint* reasoning style.

$$\frac{\{P\}c\{Q\}}{\{P * R\}c\{\lambda r.\ Q(r) * R\}}$$

In other words, any Hoare triple can be extended by conjoining an arbitrary predicate R in both precondition and postcondition. Even more intuitively, when a program satisfies a spec, it also satisfies an extended spec that records the state of some other part of memory that is untouched (i.e., is outside the command's footprint).

Modularity

For the pragmatics of proving particular programs, we defer to the accompanying Coq code. However, for modular proofs, the frame rule has such an important role that we want to emphasize it here. It is possible to define (recursively) a predicate $\mathsf{llist}(\ell, p)$, capturing the idea that the heap contains exactly an imperative linked list, rooted at pointer p, representing functional linked list ℓ. We can also prove a general specification for a list-reversal function:

$$\forall \ell, p.\ \{\mathsf{llist}(\ell, p)\}\texttt{reverse}(p)\{\lambda r.\ \mathsf{llist}(\mathsf{rev}(\ell), r)\}$$

Now consider that we have the roots p_1 and p_2 of two disjoint lists, respectively representing ℓ_1 and ℓ_2. It is easy to instantiate the general theorem and get $\{\mathsf{llist}(\ell_1, p_1)\}\texttt{reverse}(p_1)\{\lambda r.\ \mathsf{llist}(\mathsf{rev}(\ell_1), r)\}$ and $\{\mathsf{llist}(\ell_2, p_2)\}\texttt{reverse}(p_2)\{\lambda r.\ \mathsf{llist}(\mathsf{rev}(\ell_2), r)\}$. Applying the frame rule to the former theorem, with $R = \mathsf{llist}(\ell_2, p_2)$, we get:

$$\{\mathsf{llist}(\ell_1, p_1) * \mathsf{llist}(\ell_2, p_2)\}\texttt{reverse}(p_1)\{\lambda r.\ \mathsf{llist}(\mathsf{rev}(\ell_1), r) * \mathsf{llist}(\ell_2, p_2)\}$$

Similarly, applying the frame rule to the latter, with $R = \mathsf{llist}(\mathsf{rev}(\ell_1), r)$, we get:

$$\{\mathsf{llist}(\ell_2, p_2) * \mathsf{llist}(\mathsf{rev}(\ell_1), r)\}\texttt{reverse}(p_2)\{\lambda r'.\ \mathsf{llist}(\mathsf{rev}(\ell_2), r') * \mathsf{llist}(\mathsf{rev}(\ell_1), r)\}$$

Now it is routine to derive the following spec for a larger program:

$$\begin{array}{l}\{\mathsf{llist}(\ell_1, p_1) * \mathsf{llist}(\ell_2, p_2)\} \\ \quad r \leftarrow \texttt{reverse}(p_1); r' \leftarrow \texttt{reverse}(p_2);\ \mathsf{Return}(r, r') \\ \{\lambda(r, r').\ \mathsf{llist}(\mathsf{rev}(\ell_1), r) * \mathsf{llist}(\mathsf{rev}(\ell_2), r')\}\end{array}$$

Note that this specification would be incorrect if the two input lists could share any memory cells! The separating conjunction $*$ in the precondition implicitly formalizes our expectation of nonaliasing. The proof internals require only the basic

rules for Return and sequencing, in addition to the rule of consequence, whose side conditions we discharge using the cancellation approach sketched in the previous section.

Note also that this highly automatable proof style works just as well when calling functions associated with several different data structures in memory. The frame rule provides a way to show that any function, in any library, preserves arbitrary memory state outside its footprint.

14.4. Soundness Proof

Our Hoare logic is sound with respect to the object language's operational semantics.

Invariants

THEOREM 14.1. *If $\{P\}c\{Q\}$ and $P(\bullet)$, then it is an invariant of the transition system starting from $(\bullet, c)$ that either the command has become a* Return *or another execution step is possible.*

As usual, the key to the proof is to find a stronger invariant that can be proved by invariant induction. In this case, we use the invariant $\lambda(h, c).\ \{\{h\}\}c\{Q\}$. That is, assert a Hoare triple where the precondition enforces exact heap equality with the current heap. The postcondition can remain the same throughout execution.

A few key lemmas are interesting enough to mention here; we leave other details to the Coq code.

First, we need to prove that this fancier invariant implies the one from the theorem statement, and the most direct statement needs to be strengthened, to get the induction to go through.

LEMMA 14.2 (Progress). *If $\{P\}c\{Q\}$ and $P(h_1)$, then either c is a* Return *or it is possible to take a step from $(h_1 \uplus h_2, c)$, for any disjoint h_2.*

PROOF. By induction on the derivation of $\{P\}c\{Q\}$. □

Note the essential inclusion of a disjoint union with the auxiliary heap h_2. Without this strengthening of the obvious property, we would get stuck in the case of the proof for the frame rule.

LEMMA 14.3 (Preservation). *If $(h, c) \to (h', c')$ and $\{\{h\}\}c\{Q\}$, then $\{\{h'\}\}c'\{Q\}$.*

PROOF. By induction on the derivation of $(h, c) \to (h', c')$. □

The different cases of the proof depend on some not-entirely-obvious inversion lemmas. For instance, here is the one we prove for Write.

LEMMA 14.4. *If $\{P\}$*Write *$a\ v'\{Q\}$, then there exists R such that:*

- $P \Rightarrow \exists v.\ a \mapsto v * R$
- $a \mapsto v' * R \Rightarrow Q(())$

PROOF. By induction on the derivation of $\{P\}$Write $a\ v'\{Q\}$. □

Again, without the introduction of the R variable, we would get stuck proving the case for the frame rule.

CHAPTER 15

Connecting to Real-World Programming Languages

Our exercises so far have been confined to proofs of programs in idealized programming languages. How can proof assistants be used to derive results about full-scale languages? Developers are already used to coordinating build processes that plumb together multiple languages and tools. A proof assistant like Coq can become one tool in such a build flow. However, the interesting wrinkle that arises is the chance to do more than just get tools cooperating on building executable code. We can also get tools cooperating on generating proofs of parts of the system.

It is worth emphasizing that this whole subject is a very active area of research. There are many competing approaches, and it is not clear which will be most practical in the long run. With this chapter, we survey the interesting design dimensions and approaches to them that are known today. We also go into more detail on one avant-garde approach, of verified compilation of shallowly embedded programs to deeply embedded programs in Coq.

15.1. Where Does the Buck Stop?

For any system where we really care about correctness (or its special case of security), it is common to delineate a *trusted code base (TCB)*: the parts of the system where bugs could invalidate correctness. By implication, system components outside the TCB can be arbitarily buggy without endangering correctness.

When we use Coq, the Coq proof checker itself is in the TCB. Implicitly, all the infrastructure below the proof checker is also trusted. That includes the OCaml compiler (since Coq is implemented in OCaml), the operating-system kernel, the processor beneath, and so on. Interestingly, most of Coq is *not* trusted. For instance, the tactic engine outputs proof terms in a core language, and we only need to trust the (relatively small) checker for that language.

The point is, we always draw the trust boundary somewhere, and we always reason informally starting at some layer of a system. Our real-world-connecting examples of prior chapters have stopped at extracting OCaml code from Coq developments. We could avoid trusting extraction (and the OCaml compiler) by instead proving properties of C abstract syntax trees. However, then we are still trusting the C compiler! So we could verify that compiler and even the operating system its outputs run on. However, then we are still trusting the computer processor! So we could verify the processor, even down to the level of circuits with analog dynamics modeled using differential equations. However, then we are still trusting our characterization of the laws of physics!

In summary, it is naive to suggest that some formal-methods developments "prove the whole system" while others do not.

Encoding

15.2. Modes of Connecting to Real Artifacts

Any strategy in this space involves some connection between a proved component and an unproved component. Oversimplifying a bit, we are considering what is the "last level" of a proof that spans multiple abstraction layer. (The "first level" will be the top-level specification of an application or whatever other piece has a proof that is not used as a lemma for some other proof.)

15.2.1. Modes Based on Extraction. The easiest "last level" connection is via *extraction*, which translates Gallina programs into functional programs in languages like OCaml, Haskell, and Scheme. Other proof assistants than Coq tend to include similar extraction facilities. When the formal object of study is already a purely functional program, extraction is a very natural fit. Its main downside are TCB size and performance. On the TCB front, a variety of compilers, including the one for extraction itself, remain trusted. On the performance front, it is often possible to make a functional program run much faster or use much less memory by translating it to, say, a C program that doesn't rely on garbage collection.

Extraction can be part of other, less-obvious connections, where we bring certain kinds of *side effects* and real-world interactions into scope. For instance, a common class of applications is *reactive systems*. The system is viewed as an object, in the sense of object-oriented programming, with encapsulated private state and public methods that are allowed to read and modify that state. Some methods are designated as *event handlers*: they are called when various input events take place in the real world, like when a packet is received from a network. An event handler in turn signals output actions to take in response. This model is actually quite easy to implement using extraction: a reactive system is a choice of private state type plus a set of pure functions, each taking in the state and returning the new modified state. The pure functions are the methods of the object. Expressive power in this style is very high, though the TCB-size and performance objections remain.

In Chapter 13, we already saw another approach to adding side effects atop extracted code: extract syntax trees, perhaps in a mixed embedding, run with an interpreter coded directly in the target language of extraction. That interpreter is free to use whatever side effects the host language supports. If anything, the TCB-size and performance objections increase with this approach, given the additional level of indirection that comes from using syntax trees and interpretation. However, the flexibility can be very appealing, with a straightforward means of allowing most any side effect.

15.2.2. Modes Based on Explicit Rendering. Another "last level" strategy is to do explicit translation between abstract syntax trees and concrete, textual code formats. These translations usually have nothing proved about them, meaning they belong to TCBs, but often the translators are simple enough that it is relatively unworrying to trust them.

In one direction, we implement a tool that takes in, say, the textual syntax of C code, outputting Coq source code for corresponding explicit syntax trees. Now we can reason about these trees like any other mathematical objects coded manually in Coq. A downside of this approach is that it is relatively complex to parse mainstream programming languages, yet we are trusting just such a parser. However, this style often supports the smoothest integration with legacy code bases.

In the other direction, we write a Coq function from syntax trees to strings of concrete code in some widely used language. This function is still trusted, but it tends to be much shorter and worthy of trust than its inverse. Coq can be run as part of a build process, printing to the screen the string that has been computed from a syntax tree. A scripting language can be used to extract the string from Coq's output, write it to a file, and call a conventional compiler. A major challenge of this approach is that only deeply embedded languages have straightforward printing to concrete syntax, in practice, while shallowly embedded languages tend to be easier to do proofs about.

15.3. The Importance of Modularity

Modularity

Our discussion so far leaves out an important dimension. Often significant projects are divided into libraries, and we want to be able to prove libraries independently of each other. We have a few choices for facing this reality of large-scale development.

The easiest approach is to use Coq pipelines to generate single libraries, which are linked outside of Coq. Our libraries may even be linked with modules written in other languages or that would otherwise resist whatever proof methods we used. These connections across languages may enlarge the TCB significantly, but we can boost performance by linking with crucial but unverified code.

We may also want to give modules first-class status in Coq and prove the correctness of linking mechanisms. In concert with verified compilers, possibilities open up to do linking across languages without expanding the TCB. All languages can be compiled to some common format, like assembly language, and the compiled version of each libary can be given a specification in a common logical format, like a particular Hoare logic. With all the pieces in place, we wind up with the semantics of the common language in the TCB, but the compilers and all aspects of their source languages stay outside the TCB.

15.4. Case Study: From a Mixed Embedding to a Deep Embedding

This chapter's associated Coq code works out a case study of verified compilation from a mixed embedding to a deep embedding, realizing one of the "last level" options above that keeps the best of both worlds: straightforward program proof with a mixed embedding, but the smaller TCB and improved performance that comes from outputting concrete C syntax from Coq.

15.4.1. Source and Target Languages. Our mixed-embedding source language will be a simplification of last chapter's language.

$$\text{Commands} \quad c \quad ::= \quad \mathsf{Return}\ v \mid x \leftarrow c; c \mid \mathsf{Loop}\ i\ f \mid \mathsf{Read}\ n \mid \mathsf{Write}\ n\ n$$

Our deep-embedding target language will expose essentially the same features but more in the traditional style of C and related languages.

$$\begin{array}{rrcl} \text{Expressions} & e & ::= & x \mid n \mid e + e \mid *[e] \\ \text{Statements} & s & ::= & \mathsf{skip} \mid x \leftarrow e \mid *[e] \leftarrow e \mid s; s \mid \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s \mid \mathsf{while}\ e\ \mathsf{do}\ s \end{array}$$

We assume standard small-step semantics for both languages, referring to the Coq code for details. A state of the source language takes the form (h, c) for a heap h, while a state of the target language adds a variable valuation v, for triples (h, v, c).

15.4.2. Formal Compilation. It is not at all obvious how to translate from a mixed embedding to a deep embedding. We can't just write a compiler as a Gallina function, because the Bind construct is encoded using functions of the meta-language. As a result, there is no way to "recurse under a binder"!

However, inductive predicate definitions give us all the power we need, when we mix in the power of logical quantifiers. We will define a judgment $v \vdash c \hookrightarrow s$, indicating that mixed-embedded command c can be compiled to statement s, assuming that we start running s when the valuation includes every mapping from v.

A good warmup is defining a related judgment $v \vdash n \hookrightarrow e$, compiling normal Gallina numeric expressions n into syntactic expressions e.

$$\frac{v(x) = n}{v \vdash n \hookrightarrow x} \quad \frac{v \vdash n_1 \hookrightarrow e_1 \quad v \vdash n_2 \hookrightarrow e_2}{v \vdash n_1 + n_2 \hookrightarrow e_1 + e_2}$$

So far, the logic is straightforward. When we want to mention a number, it suffices to find a variable that has already been assigned that number. Translation recurses through addition in a natural way. (Note that, in the addition rule, the "+" on the left of the arrow is normal Gallina addition, while the "+" on the right is syntactic "plus" of the deeply embedded language!)

Another rule may appear at first to be overly powerful.

$$\frac{}{v \vdash n \hookrightarrow n}$$

That is, any numeric expression may be injected into the deep embedding *as a constant*. How can we hope to embed all Gallina expressions in C? The details of the command-compilation rules reveal why we are safe, so let us turn to those rules.

The rules we show here are simplified from the full set in the Coq development, supporting an even smaller subset of the source language, to make the presentation easier to understand. The rule for lone Return commands is simple, delegating most of the work to expression compilation, using a designated variable `result` to hold the final answer of a command.

$$\frac{v \vdash n \hookrightarrow e}{v \vdash \mathsf{Return}\ n \hookrightarrow \texttt{result} \leftarrow e}$$

The most interesting rules cover uses of Bind on various primitive operations directly. Here is the simplest such rule, where the primitive is simple Return.

$$\frac{y \notin \mathsf{dom}(v) \quad v \vdash n \hookrightarrow e \quad (\forall w.\ v[y \mapsto w] \vdash c(w) \hookrightarrow s)}{v \vdash x \leftarrow \mathsf{Return}\ n; c(x) \hookrightarrow y \leftarrow e; s}$$

This rule selects a deep-embedding variable name y to correspond to x in the source program. (Actually, the source-program Bind is encoded as a call to a higher-order function, so no choice of a particular x is present.) The name y must not already be present in the valuation v – it must not have been chosen already for a command executed earlier. Next, we find an expression e that represents the value n being bound. Finally, we reach the trickiest part. The statement s needs to represent the Bind body c, *for any value w that n might evaluate to*. For every such choice w, we extend v to record that w was assigned to y. In parallel, we pass w into the body c. As a result, the expression-translation rule for variables can pick up on this connection and compile mentions of w into uses of y!

Here is where we see why it wasn't problematic earlier to include a rule that translates any number n into a constant in the deep embedding. Most numeric expressions in a source program will depend on results of earlier Bind operations. However, *the quantified premise of the last rule enforces that the output statement s is not allowed to depend on w directly*! The values of introduced variables can only be accessed through deeply embedded variable names. In other words, if we tried to use the constant rule directly to prove the quantified premise of that last rule, when the body involves a Return of a complex expression that mentions the bound variable, we would generate s that includes w, which is not allowed.

Similar rules are present for Read and Write, which interestingly are compiled the same way as Return. From a syntactic, compilation standpoint, they do not behave differently from pure computation. More involved and raising new complications is the rule for loops; see the Coq code for details.

15.4.3. Soundness Proof.

THEOREM 15.1. *If $v \vdash c \hookrightarrow s$, then the source-language transition system starting at (h, c) simulates the target-language transition system starting at (h, v, s).*

PROOF. In other words, every execution of the target-language system can be mimicked by one of the source-language system, where the states are connected throughout by some relation that we choose. A good choice is the relation $\sim$ defined by this inference rule.

$$\frac{v \vdash c \hookrightarrow s}{(h, v \uplus v', s) \sim (h, c)}$$

Note that the only departure from the theorem statement itself is the allowance for compiled program s to run in a *larger* valuation than we compiled it against. It is safe to provide extra variables v' (merged into v with disjoint union), so long as the program never reads them, which our translation judgment enforces. We need to allow for extra variables because loop bodies run multiple times, with later iterations technically being exposed to temporary-variable values set by earlier iterations.

Actually, we need to modify our translation judgment so that it also applies to "silly" intermediate states of execution in the target language. For instance, we wind up with skips that are quickly stepped away, yet those configurations must be related to source configurations by $\sim$. Here is one example of the extra rules that we need to add to make our induction hypothsis strong enough.

$$\frac{v(y) = n \quad v \vdash c(n) \hookrightarrow s}{v \vdash x \leftarrow \mathsf{Return}\ n; c(x) \hookrightarrow \mathsf{skip}; s}$$

The premises encode our expectation that an assignment of n to y "just ran." □

This result can be composed with soundness of any Hoare logic for the source language. The associated Coq code defines one, essentially following our separation logic from last chapter.

THEOREM 15.2. *If $\{P\}c\{Q\}$, $P(h)$, $v \vdash c \hookrightarrow s$, and $\texttt{result} \notin \mathsf{dom}(v)$, then it is an invariant of the transition system starting in (h, v, s) that execution never gets stuck.*

Proof. First, we switch to proving an invariant of the system (h, c) using the simulation from Theorem 15.1. Next, we use the soundness theorem of the Hoare logic to weaken the invariant proved in that way into the one we want in the end. □

At this point, we can verify the high-level program conveniently while arriving at a low-level program automatically. That low-level program is easy to print as a string of concrete C code, as the associated Coq code demonstrates. We only trust the simple printing process, not the compiler that got us from a mixed embedding to a C-like syntax tree.

CHAPTER 16

Deriving Programs from Specifications

We have generally focused so far on proving that programs meet specifications. What if we could generate programs from their specifications, in ways that guarantee correctness? Let's explore that direction, in the tradition of *program derivation* via *stepwise refinement*.

16.1. Sets as Computations

The heart of stepwise refinement is to start with a specification and gradually transform it until it deserves to be called an implementation. It will help to use a common program format for specifications, implementations, and intermediate states on the path from the former to the latter. One convenient choice is *sets of allowable answers*. A specification is naturally considered as a relation R between inputs and outputs, where the set-based version of the specification for inputs x is $\mathsf{spec}(x) = \{y \mid x\ R\ y\}$. An implementation is naturally considered as a function f from an input to an output, which can be modeled with singleton sets as $\mathsf{impl}(x) = \{f(x)\}$. Intermediate terms in our derivations may still be sets with multiple elements, but we aim to winnow down to single choices eventually.

Computations of this kind form a *monad* with respect to two particular operators. Monads are an abstraction of sequential computation, popular in functional programming. They require definitions of "return" and "bind" operators, which we give here, writing $\mathcal{P}$ for the "powerset" operator that lifts types into sets.

$$\begin{array}{rcl} \mathsf{ret} & : & \forall \alpha.\ \alpha \to \mathcal{P}(\alpha) \\ \mathsf{ret} & = & \lambda x.\ \{x\} \\ \mathsf{bind} & : & \forall \alpha, \beta.\ \mathcal{P}(\alpha) \to (\alpha \to \mathcal{P}(\beta)) \to \mathcal{P}(\beta) \\ \mathsf{bind} & = & \lambda c_1.\ \lambda c_2.\ \bigcup_{x \in c_1} c_2(x) \end{array}$$

We write $x \leftarrow c_1; c_2(x)$ as shorthand for $\mathsf{bind}\ c_1\ c_2$.

A valid monad must also satisfy three algebraic laws. We will state just one of those laws here, with respect to the superset relation $\supseteq$, which we read as "refines into." That is, the lefthand operand is a more specification-like computation, which we want to replace with the righthand operand, which should be more concrete. In other words, any legal answer for the new computation is also legal for the old one. However, we may decide that the new computation rules out some answers that were previously under consideration. If we rule out all the possible answers, then we will be stuck, if we ever want to refine to a singleton set!

With our notion of refinement in place, we can state three key properties, the first of which is one of the monad laws.

THEOREM 16.1. $\mathsf{bind}\ (\mathsf{ret}\ v)\ c \supseteq c(v)$.

THEOREM 16.2. *If* $c_1 \supseteq c_1'$, *then* $\mathsf{bind}\ c_1\ c_2 \supseteq \mathsf{bind}\ c_1'\ c_2$.

THEOREM 16.3. *If* $\forall x.\ c_2(x) \supseteq c_2'(x)$, *then* $\mathsf{bind}\ c_1\ c_2 \supseteq \mathsf{bind}\ c_1\ c_2'$.

Together with the well-known reflexivity and transitivity of $\supseteq$, these laws set us up for convenient *equational reasoning*. That is, we start from a specification and repeatedly *rewrite* in it using $\supseteq$ facts, until we arrive at an acceptable implementation (singleton set whose element reads as an efficient computation). Rewriting requires us to descend inside the structure of a term to find a match for the lefthand side of a $\supseteq$ fact. When we descend into the first argument or second argument of a bind, we appeal to Theorem 16.2 or 16.3, respectively. We also use transitivity of $\supseteq$ to chain together multiple rewritings. Finally, we use Theorem 16.1 whenever we have reduced a prefix of a computation into deterministic code.

The associated Coq code contains an example of this kind of refinement in action. There are enough details that mechanized assistance is especially worthwhile.

16.2. Refinement for Abstract Data Types

Abstract data types (ADTs) are an important program-encapsulation feature that we studied in Chapter 3. Recall that they package private state together with public methods that can manipulate it, somewhat in the style of object-oriented programming. Let us now study how to start from an ADT specification and refine it gradually into an efficient implementation, in a way that leaves a "proof trail" justifying correctness.

For simplicity, we will force all methods to take $\mathbb{N}$ as input and return $\mathbb{N}$ as output, in addition to the implicit threading-through of an object's private state. The whole theory generalizes to methods of varying type.

DEFINITION 16.4. An *abstract data type (ADT)* over a set M of methods is a triple $\langle \mathcal{S}, \mathcal{C}, \mathcal{M}_{m \in M} \rangle$, where $\mathcal{S}$ is the set of private states, $\mathcal{C} : \mathcal{P}(\mathcal{S})$ is a *constructor* that initializes the state, and each $\mathcal{M}_m : \mathcal{S} \times \mathbb{N} \to \mathcal{P}(\mathcal{S} \times \mathbb{N})$ is a method.

Note that constructor and method bodies live in the computation monad, allowing them to be nondeterminstic and to mix program-style and specification-style code.

DEFINITION 16.5. Consider two ADTs $\mathcal{T}^1 = \left\langle \mathcal{S}^1, \mathcal{C}^1, \mathcal{M}^1_{m \in M} \right\rangle$ and $\mathcal{T}^2 = \left\langle \mathcal{S}^2, \mathcal{C}^2, \mathcal{M}^2_{m \in M} \right\rangle$ over the same methods M. We say that $\mathcal{T}^2$ refines $\mathcal{T}^1$ (written, with overloaded notation, as $\mathcal{T}^1 \supseteq \mathcal{T}^2$) when there exists binary relation R on $\mathcal{S}^1$ and $\mathcal{S}^2$ such that:

(1) $\forall s_2 \in \mathcal{C}^2.\ \exists s_1 \in \mathcal{C}^1.\ s_1\ R\ s_2$

(2) $\forall m, s_1, s_2.\ s_1\ R\ s_2 \Rightarrow \forall x, y, s_2'.\ (s_2', y) \in \mathcal{M}^2_m(s_2, x)$
$\Rightarrow \exists s_1'.\ (s_1', y) \in \mathcal{M}^1_m(s_1, x) \wedge s_1'\ R\ s_2'$

In fact, the relation R here is a *simulation*, in the sense of Chapter 9! Intuitively, any sequence of method calls on $\mathcal{T}^2$ can be *simulated* with the same sequence of method calls on $\mathcal{T}^1$ yielding the same answers. The private states in the two worlds needn't be precisely equal, but at each step they must remain related by R.

A number of very handy refinement principles apply.

THEOREM 16.6 (Reflexivity). $\mathcal{T} \supseteq \mathcal{T}$.

PROOF. Justified by choosing the simulation relation to be equality. □

THEOREM 16.7 (Transitivity). *If $\mathcal{T}_1 \supseteq \mathcal{T}_2$ and $\mathcal{T}_2 \supseteq \mathcal{T}_3$, then $\mathcal{T}_1 \supseteq \mathcal{T}_3$.*

PROOF. Justified by choosing the simulation relation for the conclusion to be the composition of the relations for the premises. □

THEOREM 16.8 (Focusing on a constructor). *If $\mathcal{C}^1 \supseteq \mathcal{C}^2$, then $\langle \mathcal{S}, \mathcal{C}^1, \mathcal{M}_{m \in M} \rangle \supseteq \langle \mathcal{S}, \mathcal{C}^2, \mathcal{M}_{m \in M} \rangle$.*

PROOF. Justified by choosing the simulation relation to be equality. □

THEOREM 16.9 (Focusing on a method). *Let m be one of the methods for $\mathcal{T}$, and let the body of that method be c. Let $\mathcal{T}'$ be the result of replacing m's body in $\mathcal{T}$ with a new function c'. If $\forall s, x.\ c(s, x) \supseteq c'(s, x)$, then $\mathcal{T} \supseteq \mathcal{T}'$.*

PROOF. Justified by choosing the simulation relation to be equality. □

The next simulation principle is one of the most powerful.

THEOREM 16.10 (Change of representation). *Let $\mathcal{T} = \langle \mathcal{S}, \mathcal{C}, \mathcal{M}_{m \in M} \rangle$ be an ADT, and pick $A : \mathcal{S}' \to \mathcal{S}$ (for some new state set $\mathcal{S}'$) as an* abstraction function. *Now define $\mathcal{T}' = \langle \mathcal{S}', \mathcal{C}', \mathcal{M}'_{m \in M} \rangle$, where:*

(1) $\mathcal{C}' = s \leftarrow \mathcal{C}; \{s' \mid A(s') = s\}$
(2) $\mathcal{M}'_m = \lambda s'_0, x.\ (s, y) \leftarrow M_m(A(s'_0), x); s' \leftarrow \{s' \mid A(s') = s\}; \mathsf{ret}\ (s', y)$

Then $\mathcal{T} \supseteq \mathcal{T}'$.

PROOF. Justified by choosing the simulation relation $\{(A(s), s) \mid s \in \mathcal{S}'\}$. □

The intuition of representation change is that we choose $\mathcal{S}'$ as some clever new data structure. We are responsible for a formal characterization of how it relates to the original, more obvious data structure. Abstraction function A shows how to "undo our cleverness," computing the old version of a state. It would generally be inefficient to run this conversion on every method call. Luckily, the new method bodies generated by this rule can be subjected to further optimization! For instance, we can use Theorem 16.9 to rewrite method bodies further. We will especially want to do so to replace subcomputations of the form $\{s' \mid A(s') = s\}$, which stand for calling A^{-1} on particular values. Of course not every function has an inverse as a total relation, let alone a total function, so there is no purely mechanical way to rewrite inverse function calls into executable code.

See the associated Coq code for some examples of these rules in action for concrete program derivations. It turns out that Theorems 16.6 through 16.10 are *complete*: any correct refinement fact on ADTs can be proved using them alone.

16.3. Another Example Refinement Principle: Adding a Cache

Still, it can be helpful to formulate additional ADT-refinement principles, capturing common optimization strategies. As an example, we formalize the idea of *adding a cache to a data structure*, which is also known as *finite differencing* in the literature.

THEOREM 16.11 (Adding a cache). *Let $\mathcal{T} = \langle \mathcal{S}, \mathcal{C}, \mathcal{M}_{m \in M} \rangle$ be an ADT, and pick a method m such that $\mathcal{M}_m = \lambda s, x.\ \mathsf{ret}\ (s, f(s))$ for some pure function $f : \mathcal{S} \to \mathbb{N}$. Now define $\mathcal{T}' = \langle \mathcal{S} \times \mathbb{N}, \mathcal{C}', \mathcal{M}'_{m \in M} \rangle$, where:*

(1) $\mathcal{C}' = s \leftarrow \mathcal{C}; \mathsf{ret}\ (s, f(s))$
(2) $\mathcal{M}'_m = \lambda(s, c), x.\ \mathsf{ret}\ ((s, c), c)$

(3) *For* $m' \neq m$, $\mathcal{M}'_{m'} = \lambda(s, c), x.\ (s', y) \leftarrow M_m(s, x); c' \leftarrow \{c' \mid f(s) = c \Rightarrow f(s') = c'\}; \mathsf{ret}\ ((s', c'), y)$

Then $\mathcal{T} \supseteq \mathcal{T}'$.

PROOF. Justified by choosing the simulation relation $\{(s, (s, f(s))) \mid s \in \mathcal{S}\}$. □

Intuitively, method m is a pure *observer*, not changing the state, only returning some pure function f of it. We change the state set from $\mathcal{S}$ to $\mathcal{S} \times \mathbb{N}$, so that the second component of a state *caches* the output of f on the first component. Like in a change of representation, method bodies are all rewritten automatically, but pick-from-set operations are inserted, and we must refine them away to arrive at a final implementation.

Here the crucial such pattern is $\{c' \mid f(s) = c \Rightarrow f(s') = c'\}$. Intuitively, we are asked to choose a cache value c' that is correct for the new state s', while we are allowed to *assume* that the prior cache value c was accurate for the old state s. Therefore, it is natural to give an efficient formula for computing c' in terms of c.

CHAPTER 17

Introduction to Reasoning About Shared-Memory Concurrency

Separation logic tames sharing of a mutable memory across libraries and data structures. We will need some additional techniques when we add concurrency to the mix, resulting in the *shared-memory* style of concurrency. This chapter introduces a basic style of operational semantics for shared memory, also studying its use in model checking, including with an important optimization called partial-order reduction. The next chapter shows how to prove deeper properties of fancier programs, by extending the Hoare-logic approach to shared-memory concurrency. Then the chapter after that shows how to formalize and reason about a different style of concurrency, message passing.

17.1. An Object Language with Shared-Memory Concurrency

For the next two chapters, we work with this object language.

$$\text{Commands} \quad c \quad ::= \quad \mathsf{Fail} \mid \mathsf{Return}\ v \mid x \leftarrow c; c \mid \mathsf{Read}\ a \mid \mathsf{Write}\ a\ v \mid \mathsf{Lock}\ a \mid \mathsf{Unlock}\ a \mid c||c$$

In addition to the basic structure of the languages from the last two chapters, we have three features specific to concurrency. We follow the common "threads and locks" style of synchronization, with commands $\mathsf{Lock}\ a$ and $\mathsf{Unlock}\ a$ for acquiring and releasing locks, respectively. We also have $c_1||c_2$ for running commands c_1 and c_2 in parallel, giving a scheduler free reign to interleave their atomic steps.

The operational semantics is small-step, especially because big-step semantics is notoriously awkward for concurrency. Each state of the system is a triple (h, l, c), with h and c the heap and current command from our usual semantics. New component l is a *lockset*, recording which locks are currently held, without distinguishing between different threads that might have taken them.

$$\frac{(h, l, c_1) \to (h', l', c_1')}{(h, l, x \leftarrow c_1; c_2(x)) \to (h', l', x \leftarrow c_1'; c_2(x))} \quad \frac{}{(h, l, x \leftarrow \mathsf{Return}\ v; c_2(x)) \to (h, k, c_2(v))}$$

$$\frac{}{(h, l, \mathsf{Read}\ a) \to (h, l, \mathsf{Return}\ h(a))} \quad \frac{}{(h, l, \mathsf{Write}\ a\ v) \to (h[a \mapsto v], l, \mathsf{Return}\ 0)}$$

$$\frac{a \notin l}{(h, l, \mathsf{Lock}\ a) \to (h, l \cup \{a\}, \mathsf{Return}\ 0)} \quad \frac{a \in l}{(h, l, \mathsf{Unlock}\ a) \to (h, l \backslash \{a\}, \mathsf{Return}\ 0)}$$

$$\frac{(h, l, c_1) \to (h', l', c_1')}{(h, l, c_1||c_2) \to (h', l', c_1'||c_2)} \quad \frac{(h, l, c_2) \to (h', l', c_2')}{(h, l, c_1||c_2) \to (h', l', c_1||c_2')}$$

Note that the last two rules are the only source of *nondeterminism* in this semantics, where a single state can step to multiple different next states. This nondeterminism corresponds to the freedom we give to a scheduler that may pick which thread runs next. Though this kind of concurrent programming is very expressive and often achieves very high performance, it comes at a cost in reasoning, as there may be *exponentially many different schedules* for a single program, measured with respect to the textual length of the program. A popular name for this pitfall is *the state-explosion problem*.

Note also that we have omitted any looping constructs from this object language, so all programs terminate. The Coq formalization uses the mixed-embedding style, making it not entirely obvious that all programs really do terminate. In any case, if we must tame the state-explosion problem, we already have our work cut out for us, even when the state space rooted at any concrete state is finite!

17.2. Shrinking the State Space via Local Actions

Recall our study of *model checking* in Chapter 6. With a little cleverness, many problems in program verification can be reduced to exploration of finite state spaces of transition systems. In particular, we looked at *safety properties*, which can be expressed as invariants of transition systems. One simply follows all the edges in the graph determined by a transition system, accepting the program if that process terminates without finding a state that violates the invariant. For our object language in this chapter, a good safety property is that commands are *not about to fail*, formalized as:

$$\begin{aligned}
\mathsf{natf}(\mathsf{Fail}) &= \bot \\
\mathsf{natf}(x \leftarrow c_1; c_x(x)) &= \mathsf{natf}(c_1) \\
\mathsf{natf}(c_1 || c_2) &= \mathsf{natf}(c_1) \wedge \mathsf{natf}(c_2) \\
\mathsf{natf}(_) &= \top
\end{aligned}$$

Here is an example of a program execution that avoids failures.

$$\begin{aligned}
(\bullet[0 \mapsto 1], \varnothing, n \leftarrow \mathsf{Read}\ 0; \mathsf{Write}\ 0\ (n+1)) &\rightarrow (\bullet[0 \mapsto 1], \varnothing, n \leftarrow \mathsf{Return}\ 1; \mathsf{Write}\ 0\ (n+1)) \\
&\rightarrow (\bullet[0 \mapsto 1], \varnothing, \mathsf{Write}\ 0\ (1+1)) \\
&\rightarrow (\bullet[0 \mapsto 2], \varnothing, \mathsf{Return}\ 0)
\end{aligned}$$

When exploring the state space of this program, a näive model checker will generate each of these states explicitly, even the "silly" second one that reduces to the third without reading or writing the shared state. We can short-circuit those extra states by writing a simple function that makes all appropriate purely local reductions, everywhere within a command.

$$\begin{aligned}
\lfloor x \leftarrow c_1; c_2(x) \rfloor &= \lfloor c_2(v) \rfloor, \text{ when } \lfloor c_1 \rfloor = \mathsf{Return}\ v \\
\lfloor x \leftarrow c_1; c_2(x) \rfloor &= x \leftarrow \lfloor c_1 \rfloor; \lfloor c_2(x) \rfloor, \text{ when } \lfloor c_1 \rfloor \text{ is not } \mathsf{Return} \\
\lfloor c_1 || c_2 \rfloor &= \lfloor c_1 \rfloor || \lfloor c_2 \rfloor \\
\lfloor c \rfloor &= c
\end{aligned}$$

Using this relation, we can define an alternative step relation that short-circuits local steps.

$$\frac{(h, l, c) \rightarrow (h', l', c')}{(h, l, c) \rightarrow_L (h', l', \lfloor c' \rfloor)}$$

The base semantics can be used to define transition systems in the usual way, with $\mathbb{T}(h, l, c) = \langle \{(h, l, c)\}, \rightarrow \rangle$. We can also define short-circuiting transition systems with $\mathbb{T}_L(h, l, c) = \langle \{(h, l, \lfloor c \rfloor)\}, \rightarrow_L \rangle$. A theorem shows that the latter overapproximates the former.

Abstraction

THEOREM 17.1. *If* natf *is an invariant of* $\mathbb{T}_L(h, l, c)$*, then it is also an invariant of* $\mathbb{T}(h, l, c)$*.*

PROOF. By induction on a trace $(h, l, c) \rightarrow^* (h', l', c')$, matching each original step with zero or one alternative steps. We appeal to a number of lemmas, some of which are summarized below. □

LEMMA 17.2. *For all* c*,* $\lfloor \lfloor c \rfloor \rfloor = \lfloor c \rfloor$*.*

PROOF. By induction on the structure of c. □

LEMMA 17.3. *If* $(h, l, c) \rightarrow (h', l', c')$*, then either* $(h', l') = (h, l)$ *and* $\lfloor c' \rfloor = \lfloor c \rfloor$ *(the step was local), or there exists* c'' *where* $(h, l, \lfloor c \rfloor) \rightarrow (h', l', c'')$ *and* $\lfloor c'' \rfloor = \lfloor c' \rfloor$ *(the step was not local).*

PROOF. By induction on the derivation of $(h, l, c) \rightarrow (h', l', c')$, appealing in places to to Lemma 17.2. □

LEMMA 17.4. *If* $\mathsf{natf}(\lfloor c \rfloor)$*, then* $\mathsf{natf}(c)$*.*

PROOF. By induction on the structure of c. □

17.3. Basic Partial-Order Reduction

What made the reduction in Theorem 17.1 sound? It was that local actions *commute* with all actions in other threads. A particular run of a system in the base semantics might indeed choose to run a nonlocal action before a local action that is enabled. However, we can *reorder* any such action to instead come after every enabled local action, without affecting the final state. This reordering is an example of commutativity in action.

By recognizing and exploiting other varieties of commutativity, we can shrink state spaces even further, even reducing the spaces of certain interesting program families from exponential size to linear size. A popular technique of this kind is *partial-order reduction*. We formalize a simple variant of it in this section (and in the accompanying Coq code), then sketch a less formal generalization in the chapter's final section.

To check commutativity more flexibly, we must use more than just the fact that a local action commutes with any action in another thread. For instance, we should take advantage of the fact that any two Read actions commute. We will do some *static analysis* of programs to overapproximate which kinds of atomic actions they might perform. Such an analysis is designed to be trivially computable. Here's an example of one analysis, formulated as a relation $\mathsf{summarize}(c, (r, w, \ell))$, which asserts that the only globally visible actions that could be performed by thread c are reads to addresses in r, writes to addresses in w, and acquires or releases of locks in ℓ.

$$\frac{}{\mathsf{summarize}(\mathsf{Return}\ r, s)} \qquad \frac{}{\mathsf{summarize}(\mathsf{Fail}, s)} \qquad \frac{\mathsf{summarize}(c_1, s) \quad \forall r.\ \mathsf{summarize}(c_2(r), s)}{\mathsf{summarize}(x \leftarrow c_1; c_2(x), s)}$$

$$\frac{a \in r}{\mathsf{summarize}(\mathsf{Read}\ a, (r, w, \ell))} \quad \frac{a \in w}{\mathsf{summarize}(\mathsf{Write}\ a\ v, (r, w, \ell))}$$

$$\frac{a \in \ell}{\mathsf{summarize}(\mathsf{Lock}\ a, (r, w, \ell))} \quad \frac{a \in \ell}{\mathsf{summarize}(\mathsf{Unlock}\ a, (r, w, \ell))}$$

$$\frac{\mathsf{summarize}(c_1, s) \quad \mathsf{summarize}(c_2, s)}{\mathsf{summarize}(c_1 || c_2, s)}$$

Those relations do all we need to do to record which actions a thread might not commute with. The other key ingredient is an extractor for the next atomic action in a thread, written as a partial function.

$$\begin{aligned}
\mathsf{nextAction}(\mathsf{Return}\ r) &= \mathsf{Return}\ r \\
\mathsf{nextAction}(\mathsf{Fail}) &= \mathsf{Fail} \\
\mathsf{nextAction}(\mathsf{Read}\ a) &= \mathsf{Read}\ a \\
\mathsf{nextAction}(\mathsf{Write}\ a\ v) &= \mathsf{Write}\ a\ v \\
\mathsf{nextAction}(\mathsf{Lock}\ a) &= \mathsf{Lock}\ a \\
\mathsf{nextAction}(\mathsf{Unlock}\ a) &= \mathsf{Unlock}\ a \\
\mathsf{nextAction}(x \leftarrow c_1; c_2(x)) &= \mathsf{nextAction}(c_1)
\end{aligned}$$

Given a next atomic action and a summary of another thread, it is now easy to define commutativity of the two.

$$\begin{aligned}
\mathsf{commutes}(\mathsf{Return}\ _, _) &= \top \\
\mathsf{commutes}(\mathsf{Fail}, _) &= \top \\
\mathsf{commutes}(\mathsf{Read}\ a, (_, w, _)) &= a \notin w \\
\mathsf{commutes}(\mathsf{Write}\ a\ _, (r, w, _)) &= a \notin r \cup w \\
\mathsf{commutes}(\mathsf{Lock}\ a, (_, _, \ell)) &= a \notin \ell \\
\mathsf{commutes}(\mathsf{Unlock}\ a, (_, _, \ell)) &= a \notin \ell \\
\mathsf{commutes}(_, _) &= \bot
\end{aligned}$$

With these ingredients, we can define a predicate porSafe that figures out when a state is eligible for the partial-order reduction optimization, which is to force the first thread to run next, ignoring the other threads for now. In working out the formal details, we will confine ourselves to commands $c_1||c_2$ with distinguished "first threads" c_1, though everything can be generalized to other settings (and doing that generalization could be a worthwhile exercise for the reader, though it requires a lot of logical bookkeeping). This optimization is only safe when the first thread can take a step and when that step commutes with any action that other threads (combined into c_2) might perform. Formally, we define $\mathsf{porSafe}(h, l, c_1, c_2, s)$ as follows, where s should be a valid summary of c_2.

- There is some c_0 where $\mathsf{nextAction}(c_1) = c_0$. That is, thread c_1 has some uniquely determined atomic action lined up to run next.
- There exist h', l', and c_1' such that $(h, l, c_1) \to (h', l', c_1')$. That is, thread c_1 is actually able to take a step, which might not be possible if e.g. trying to take a lock that is already held.

- And the crucial compatibility condition: $\mathsf{commutes}(c_0, s)$. That is, all actions that other threads might perform commute with c_0, the first action of c_1.

With the applicability condition defined, it is now straightforward to define an optimized step relation, parameterized on an accurate summary s for c_2.

$$\frac{(h, l, c_1) \to (h', l', c_1')}{(h, l, c_1||c_2) \to_C^s (h', l', c_1'||c_2)}$$

$$\frac{\neg\mathsf{porSafe}(h, l, c_1, c_2, s) \quad (h, l, c_2) \to (h', l', c_2')}{(h, l, c_1||c_2) \to_C^s (h', l', c_1||c_2')}$$

The whole thing is wrapped up into transition systems as $\mathbb{T}_C(h, l, c_1, c_2, s) = \langle\{(h, l, c_1||c_2)\}, \to_C^s\rangle$.

Our proof of soundness for this reduction will depend on having some constant upper bound on program execution time. This relation computes a conservative overapproximation.

$$\frac{}{\mathsf{timeOf}(\mathsf{Return}\ r, n)} \quad \frac{}{\mathsf{timeOf}(\mathsf{Fail}, n)} \quad \frac{}{\mathsf{timeOf}(\mathsf{Read}\ a, n+1)} \quad \frac{}{\mathsf{timeOf}(\mathsf{Write}\ a\ v, n+1)}$$

$$\frac{}{\mathsf{timeOf}(\mathsf{Lock}\ a, n+1)} \quad \frac{}{\mathsf{timeOf}(\mathsf{Unlock}\ a, n+1)}$$

$$\frac{\mathsf{timeOf}(c_1, n_1) \quad \forall r.\ \mathsf{timeOf}(c_2(r), n_2)}{\mathsf{timeOf}(x \leftarrow c_1; c_2(x), n_1 + n_2 + 1)} \quad \frac{\mathsf{timeOf}(c_1, n_1) \quad \mathsf{timeOf}(c_2, n_2)}{\mathsf{timeOf}(c_1||c_2, n_1 + n_2 + 1)}$$

It may be surprising that, in our formal mixed embedding, there exist commands with no provable upper bounds, according to this relation. We leave it as an exercise to the reader to find a concrete example. (Actually, the Coq code includes an example and its proof of unboundedness.)

One last ingredient is to work with a relation $\to^i$, which is the i-way self-composition of $\to$. It is easy to show that whenever $x \to^* y$, there exists i such that $x \to^i y$.

With these ingredients, we can state the reduction theorem. **Abstraction**

THEOREM 17.5. *If* $\mathsf{summarize}(c_2, s)$ *and* $\mathsf{timeOf}(c_1||c_2, n)$*, then to prove* natf *as an invariant of* $\mathbb{T}(h, l, c_1||c_2)$*, it suffices to prove* natf *as an invariant of* $\mathbb{T}_C(h, l, c_1, c_2, s)$.

PROOF. Setting $c = c_1||c_2$, we assume for the sake of contradiction that there exists some derivation $(h, l, c) \to^* (h', l', c')$, where $\neg\mathsf{natf}(h', l', c')$. First, since c runs in bounded time, by Lemma 17.6, we can *complete* that execution to continue running to some (h'', l'', c''), which is a stuck state. By Lemma 17.7, $\neg\mathsf{natf}(h'', l'', c'')$. Next, we conclude that there exists i such that $(h, l, c) \to^i (h'', l'', c'')$. By Lemma 17.8, there exist h''', l''', and c''' where $(h, l, c_1||c_2) \to_C^{s*} (h''', l''', c''')$ and $\neg\mathsf{natf}(c''')$. These facts contradict our assumption that natf is an invariant of $\mathbb{T}_C(h, l, c_1, c_2, s)$. □

LEMMA 17.6. *If* $\mathsf{timeOf}(c, n)$*, then there exist* h'*,* l'*, and* c' *where* $(h, l, c) \to^* (h', l', c')$*, such that* (h', l', c') *is a stuck state.*

PROOF. By strong induction on n. □

LEMMA 17.7. *If* $(h, l, c) \rightarrow^* (h', l', c')$ *and* $\neg\mathsf{natf}(c)$*, then* $\neg\mathsf{natf}(c')$*.*

PROOF. By induction on the derivation of $(h, l, c) \rightarrow^* (h', l', c')$, with a nested induction on the derivations of individual steps. □

LEMMA 17.8. *If* $(h, l, c_1||c_2) \rightarrow^i (h', l', c')$*, and* (h', l', c') *is stuck and about to fail, and* $\mathsf{summarize}(c_2, s)$*, then there exist* h''*,* l''*, and* c''' *such that* $(h, l, c_1||c_2) \rightarrow_C^{s*} (h'', l'', c''')$ *and* $\neg\mathsf{natf}(c''')$*.*

PROOF. By induction on i. Note that induction on the structure of a derivation $(h, l, c_1||c_2) \rightarrow^* (h', l', c')$ would *not* be sufficient here, as we will see in the proof sketch below that we sometimes invoke the induction hypothesis on an execution trace that is not just the tail of the one we started with.

If $i = 0$, then $(h, l, c_1||c_2)$ is already about to fail, and the conclusion follows trivially.

Otherwise, $i = i' + 1$ for some i'. We proceed by cases on the truth of $\mathsf{porSafe}(h, l, c_1, c_2, s)$.

If $\neg\mathsf{porSafe}(h, l, c_1, c_2, s)$, then we invert the derivation $(h, l, c_1||c_2) \rightarrow^i (h', l', c')$ to conclude $(h, l, c_1||c_2) \rightarrow (h'', l'', c'')$ and $(h'', l'', c'') \rightarrow^{i'} (h', l', c')$ for some intermediate state. $\rightarrow_C$ is easily able to match that first step, as the optimization is disabled, and the rest follows directly by appeal to the induction hypothesis.

Otherwise, $\mathsf{porSafe}(h, l, c_1, c_2, s)$, and the key optimization is enabled, so that $\rightarrow_C$ only allows the first thread to run. The next deduction is not immediate, because the first original step $(h, l, c_1||c_2) \rightarrow (h'', l'', c'')$ may have chosen a thread beside c_1. However, the trace $(h, l, c_1||c_2) \rightarrow^{i'+1} (h', l', c')$ *must* eventually pick the first thread to run, and we apply Lemma 17.9 to *commute* that eventual step to the front of the derivation, showing its equivalence to one that runs the first thread and then takes i' additional steps to (h', l', c'). At this point the induction hypothesis applies to those i' steps, to finish the proof. □

LEMMA 17.9. *If* $(h, l, c_1||c_2) \rightarrow^{i+1} (h', l', c')$*, where that last state is stuck, and if* $\mathsf{summarize}(c_2, s)$*,* $\mathsf{nextAction}(c_1) = x$*,* $\mathsf{commutes}(x, s)$*, and* $(h, l, c_1) \rightarrow (h_0, l_0, c_1')$*, then* $(h_0, l_0, c_1'||c_2) \rightarrow^i (h', l', c')$*.*

PROOF. By induction on the derivation of $(h, l, c_1||c_2) \rightarrow^{i+1} (h', l', c')$, appealing to a few crucial lemmas, such as single-step determinism of any state in nextAction's domain, plus the soundness of commutes with respect to single steps of pairs of commands, plus the fact that single steps preserve the accuracy of summaries. □

17.4. Partial-Order Reduction More Generally

The key insights of the prior section can be adapted to prove soundness of a whole family of optimizations by partial-order reduction. In general, we apply the optimization to remove edges from a state-space graph, whose nodes are states and whose edges are labeled with *actions* α. In our setting, α is the identifier of the thread scheduled to run next. To do model checking, the graph need not be materialized in memory in one go. Instead, as an optimization, the graph tends to be constructed on-the-fly, during state-space exploration.

The proofs from the last two sections only apply to check the invariant that no thread is about to fail. However, the results easily generalize to arbitrary *safety*

properties, which can be expressed as decidable invariants on states. Another important class of specifications is *liveness properties*, the most canonical example of which is *termination*, phrased in terms of reachability of some subsets of states designated as *finished*. There are many other useful liveness properties. Another example applies to a producer-consumer system, where one thread continually enqueues new work into a queue, and another thread continually dequeues work items and does something with them. A good liveness property for that system could be that, whenever the producer enqueues an item, the consumer eventually dequeues it, and from there the consumer eventually takes some visible action based on the value of the item. Our general treatment of partial-order reduction is parameterized on some property ϕ over states, and it may be safety, liveness, or a combination of the two.

Every state s of the transition system has an associated set $\mathcal{E}(s)$ of identifiers for threads that are enabled to run in s. The partial-order reduction optimization conceptually is based on picking a function $\mathcal{A}$, mapping each state s to an *ample set* $\mathcal{A}(s)$ of threads to consider in state-space exploration. A few eligibility criteria apply, for every state s.

Readiness: $\mathcal{A}(s) \subseteq \mathcal{E}(s)$. That is, we do not select any threads that are not actually ready to run.

Progress: If $\mathcal{A}(s) = \varnothing$, then $\mathcal{E}(s) = \varnothing$. That is, so long as any thread at all can step, we select at least one thread.

Commutativity: Consider all executions starting at s and taking steps only with the threads *not* in $\mathcal{A}(s)$. These executions only include actions that commute with the next actions of the threads in $\mathcal{A}(s)$. As a consequence, any actions that run before elements of the ample set can be reordered to follow the execution of any ample-set element.

Invisibility: If $\mathcal{A}(s) \neq \mathcal{E}(s)$, then no action in $\mathcal{A}(s)$ modifies the truth of ϕ.

Any ample-set algorithm leads to a different variant of $\rightarrow_C$ from the prior section, and it is possible to prove that any such transition system is a sound abstraction of the original.

As an example of a different heuristic, consider a weakness of the one from the prior section: when we pick a thread c as the only one to consider running next, c's first action must commute with *any action that any other thread might ever run, for the entire rest of the execution*. However, imagine that thread c holds some lock a. We might formalize that notion by saying that (1) a is in the lockset, and (2) the other threads, running independently, will never manage to run an Unlock a command. A computable static analysis can verify this statement for many nontrivial programs. Now consider which summaries of the other threads we can get away with comparing against c for commutativity. We only need to collect the actions that other threads can run *before each one reaches its first occurrence of* Lock a. The reason is that, if c holds lock a and hasn't run yet, no other thread can progress past its first Lock a. Now threads may share addresses for read and write access, yet still take advantage of the optimization, when accesses are properly protected by locks.

The conditions above are only sufficient because we left unbounded loops out of our object language. What happens if we add them back in? Consider this program:

$$(\mathsf{while}\ (\mathsf{true})\ \{\ \mathsf{Write}\ 0\ 0\ \})\ ||\ (n \leftarrow \mathsf{Read}\ 1; \mathsf{Fail})$$

An optimization in the spirit of our original from the prior section would happily decree that it is safe always to pick the first thread to run. This reduced state-transition system never gets around to running the second thread, so exploring the state space never finds the failure! To plug this soundness hole, we add a final condition on the ample sets.

Fairness: If there is a cycle in the finite state space where α is enabled at some point, then $\alpha \in \mathcal{A}(s)$ for some s in the cycle.

This condition effectively forces the ample set for the example program above to include the second thread.

CHAPTER 18

Concurrent Separation Logic

Chapters 14 and 17 respectively introduced techniques for reasoning about two tricky aspects of programs: heap-allocated linked data structures and shared-memory concurrency. When we add concurrency to the mix for a program-reasoning problem, we are often surprised at how much more complex it becomes. This chapter introduces a pleasant exception to the rule, *concurrent separation logic*, a rather small addition to separation logic that supports invariant-based reasoning about threads-and-locks shared-memory programs.

18.1. Object Language: Loops and Locks

Here's the object language we adopt, which should be old hat by now, just mixing together features of the object languages from Chapters 14 and 17.

$$\begin{array}{rrcl} \text{Commands} & c & ::= & \mathsf{Fail} \mid \mathsf{Return}\ v \mid x \leftarrow c; c \mid \mathsf{Loop}\ i\ f \\ & & & \mid \mathsf{Read}\ a \mid \mathsf{Write}\ a\ v \mid \mathsf{Alloc}\ n \mid \mathsf{Free}\ a\ n \mid \mathsf{Lock}\ a \mid \mathsf{Unlock}\ a \mid c||c \end{array}$$

$$\frac{(h, l, c_1) \to (h', l', c_1')}{(h, l, x \leftarrow c_1; c_2(x)) \to (h', l', x \leftarrow c_1'; c_2(x))} \quad \frac{}{(h, l, x \leftarrow \mathsf{Return}\ v; c_2(x)) \to (h, k, c_2(v))}$$

$$\frac{}{(h, l, \mathsf{Loop}\ i\ f) \to (h, l, x \leftarrow f(i); \mathsf{match}\ x\ \mathsf{with}\ \mathsf{Done}(a) \Rightarrow \mathsf{Return}\ a \mid \mathsf{Again}(a) \Rightarrow \mathsf{Loop}\ a\ f)}$$

$$\frac{h(a) = v}{(h, l, \mathsf{Read}\ a) \to (h, l, \mathsf{Return}\ v)} \quad \frac{h(a) = v}{(h, l, \mathsf{Write}\ a\ v') \to (h[a \mapsto v'], l, \mathsf{Return}\ ())}$$

$$\frac{\mathsf{dom}(h) \cap [a, a+n) = \varnothing}{(h, \mathsf{Alloc}\ n) \to (h[a \mapsto 0^n], \mathsf{Return}\ a)} \quad \frac{}{(h, \mathsf{Free}\ a\ n) \to (h - [a, a+n), \mathsf{Return}\ ())}$$

$$\frac{a \notin l}{(h, l, \mathsf{Lock}\ a) \to (h, l \cup \{a\}, \mathsf{Return}\ ())} \quad \frac{a \in l}{(h, l, \mathsf{Unlock}\ a) \to (h, l \backslash \{a\}, \mathsf{Return}\ ())}$$

$$\frac{(h, l, c_1) \to (h', l', c_1')}{(h, l, c_1||c_2) \to (h', l', c_1'||c_2)} \quad \frac{(h, l, c_2) \to (h', l', c_2')}{(h, l, c_1||c_2) \to (h', l', c_1||c_2')}$$

18.2. The Program Logic

We will build on basic separation logic, using the same kind of assertions and even adopting all of the original rules unchanged. Here they are again, for easy reference.

$$\frac{}{\{\mathsf{emp}\}\mathsf{Return}\ v\{\lambda r.\ [r = v]\}} \quad \frac{\{P\}c_1\{Q\} \quad (\forall r.\ \{Q(r)\}c_2(r)\{R\})}{\{P\}x \leftarrow c_1; c_2(x)\{R\}}$$

$$\frac{\forall a.\ \{I(\mathsf{Again}(a))\}f(a)\{I\}}{\{I(\mathsf{Again}(i))\}\mathsf{Loop}\ i\ f\{\lambda r.\ I(\mathsf{Done}(r))\}} \quad \frac{}{\{[\bot]\}\mathsf{Fail}\{\lambda\_.\ [\bot]\}}$$

$$\frac{}{\{\exists v.\ a \mapsto v * R(v)\}\mathsf{Read}\ a\{\lambda r.\ a \mapsto r * R(r)\}} \quad \frac{}{\{\exists v.\ a \mapsto v\}\mathsf{Write}\ a\ v'\{\lambda\_.\ a \mapsto v'\}}$$

$$\frac{}{\{\mathsf{emp}\}\mathsf{Alloc}\ n\{\lambda r.\ r \mapsto 0^n\}} \quad \frac{}{\{a \mapsto ?^n\}\mathsf{Free}\ a\ n\{\lambda\_.\ \mathsf{emp}\}}$$

$$\frac{\{P\}c\{Q\} \quad P' \Rightarrow P \quad \forall r.\ Q(r) \Rightarrow Q'(r)}{\{P'\}c\{Q'\}} \quad \frac{\{P\}c\{Q\}}{\{P * R\}c\{\lambda r.\ Q(r) * R\}}$$

Modularity

When two threads use disjoint regions of memory, it is trivial to apply this rule of Concurrent Separation Logic to verify the threads independently.

$$\frac{\{P_1\}c_1\{Q_1\} \quad \{P_2\}c_2\{Q_2\}}{\{P_1 * P_2\}c_1||c_2\{\lambda\_.\ [\bot]\}}$$

The separating conjunction $*$ turned out to be just the right way to express the idea of "splitting the heap into a part for the first thread and a part for the second thread." Because c_1 and c_2 touch disjoint memory regions, all of their memory operations commute, so that we need not worry about the state-explosion problem, in all the ways that the scheduler might interleave their steps. Note that, since for simplicity our running example family of concurrent object languages includes no way for parallel compositions to terminate, it makes sense to assign a contradictory overall postcondition.

However, with realistic shared-memory programs, we don't get off as easy as the parallel-composition rule suggests. Threads *do* share memory regions, using *synchronization* to tame the state-explosion problem. Our object language includes locks as its example of synchronization, and Concurrent Separation Logic is specialized to locks. We may keep the simplistic-seeming rule for parallel composition and implicitly enrich its power by adding a twist, in the form of some other rules.

Invariants

The big twist is that we parameterize everything over some finite set L of locks that may be used. Furthermore, another parameter is a function $\mathcal{I}$ that maps locks to invariants, which have the same type as preconditions. The idea is this: when no one holds a lock, *the lock owns a chunk of memory that satisfies its invariant.* When a thread holds the lock, the lock doesn't own any memory; it is waiting for the thread to unlock it and *donate back* a chunk of memory satisfying the invariant. We now think of the precondition of a Hoare triple as only describing the *local memory* of a thread, which no other thread may access; while locks and their invariants coordinate the *shared memory* regions of an application. The proof rules will coordinate dynamic motion of memory regions between the shared regions

and local regions. This motion is only part of a proof technique; it has no runtime content reflected in the operational semantics!

With all of that set-up, the final two rules may seem surprisingly simple.

$$\frac{a \in L}{\{\mathsf{emp}\}\mathsf{Lock}\ a\{\lambda\_.\ \mathcal{I}(a)\}} \qquad \frac{a \in L}{\{\mathcal{I}(a)\}\mathsf{Unlock}\ a\{\lambda\_.\ \mathsf{emp}\}}$$

When a thread takes a lock, it appears as if *a memory chunk satisfying that lock's invariant materializes in the local memory space.* Conversely, when a thread releases a lock, it appears as if *the lock grabs a memory chunk satisfying the invariant out of the local memory space.* The rules are coordinating conceptual ownership transfers between local memory and the global lock memory.

The accompanying Coq code shows a few example verifications of interesting programs.

18.3. Soundness Proof

We can adapt the separation-logic soundness proof to concurrency, with just a few new ideas. First, we will appreciate some new connectives for writing assertions. One simple one is a guarded predicate, defined like so, for pure proposition ϕ (the guard) and separation-logic assertion P.

$$\phi \longrightarrow P \quad = \quad \mathsf{if}\ \phi\ \mathsf{then}\ P\ \mathsf{else}\ \mathsf{emp}$$

The other key addition will be the "big star," *iterated separating conjunction*, with quantification over finite sets, written like $\ast_{x \in S} P(x)$. The definition is:

$$\ast_{x \in \{v_1, \ldots, v_n\}} P(x) \quad = \quad P(v_1) * \ldots * P(v_n)$$

The reader may be worried about the inherently unordered nature of sets. For each ordering of a set, we get a syntactically distinct formula on the righthand side of the defining equation. Luckily, separating conjunction $*$ is associative and commutative, so all orders lead to logically equivalent formulas.

With those preliminaries out of the way, we can state the soundness theorem, referring again to the *not-about-to-fail* predicate natf from last chapter, extended appropriately to say that loops are not about to fail.

Invariants

Theorem 18.1 (Soundness). *If $\{P\}c\{Q\}$, and if a heap h satisfies the predicate $(P * \ast_{\ell \in L} \mathcal{I}(\ell))$, then* natf *is an invariant of the system starting at state $(h, \varnothing, c)$.*

The theorem lays out restrictions on the starting heap. It must have a segment to serve as the root thread's local heap, matching precondition P. Then, for each lock $\ell \in L$, there must be an associated memory region satisfying $\mathcal{I}(\ell)$. Our use of separating conjunction forces each of these regions to occupy disjoint memory from all the others.

Some key lemmas support the proof. Here are the highlights. The first is representative of a family of lemmas that we prove, one for each syntactic construct of the object language.

Lemma 18.2. *If $\{P\}$Read $a\{Q\}$, then there exists R such that $P \Rightarrow \exists v.\ a \mapsto v * R(v)$ and, for all r, $a \mapsto r * R(r) \Rightarrow Q(r)$.*

Proof. By induction on the derivation of $\{P\}$Read $a\{Q\}$. □

As another example incorporating more of the complexities of concurrency, we have this lemma.

LEMMA 18.3. *If $\{P\}c_1||c_2\{Q\}$, then there exist P_1, P_2, Q_1, and Q_2 such that $\{P_1\}c_1\{Q_1\}$, $\{P_2\}c_2\{Q_2\}$, and $P \Rightarrow P_1 * P_2$.*

PROOF. By induction on the derivation of $\{P\}c_1||c_2\{Q\}$. One somewhat surprising case is when the frame rule begins the derivation. We have some predicate R that is added to both the precondition and postcondition. In picking P_1, P_2, Q_1, and Q_2, we have a choice as to where we incorporate R. The two threads together leave R alone, so clearly either thread individually does, too. Therefore, we arbitrarily incorporate R in P_1 and Q_1. □

Two lemmas express crucial techniques to isolate elements within iterated conjunction.

LEMMA 18.4. *If $v \in S$, then $\circledast_{x\in S}P(x) \Rightarrow P(v) * \circledast_{x\in S\setminus\{v\}}P(x)$.*

PROOF. By induction on the cardinality of S. □

LEMMA 18.5. *If $v \notin S$, then $P(v) * \circledast_{x\in S}P(x) \Rightarrow \circledast_{x\in S\cup\{v\}}P(x)$.*

PROOF. By induction on the cardinality of S. □

LEMMA 18.6 (Preservation). *If $(h, l, c) \to (h', l', c')$, $\{P\}c\{Q\}$, and h satisfies $(P * R * \circledast_{\ell\in L}(\ell \notin l \longrightarrow \mathcal{I}(\ell)))$, then there exists P' such that $\{P'\}c'\{Q\}$, where h' satisfies $(P' * R * \circledast_{\ell\in L}(\ell \notin l' \longrightarrow \mathcal{I}(\ell)))$.*

PROOF. By induction on the derivation of $(h, l, c) \to (h', l', c')$. The cases for lock and unlock respectively use Lemmas 18.4 and 18.5. Note that we include the parameter R solely to get a strong enough induction hypothesis for steps of commands $c_1||c_2$. We need to know that a step by one thread does not change the private heap of the other thread. To draw that conclusion, in appealing to the induction hypothesis, we extend R with precisely that private state. □

LEMMA 18.7. $\circledast_{\ell\in L}\mathcal{I}(\ell) \Rightarrow \circledast_{\ell\in L}(\ell \notin \varnothing \longrightarrow \mathcal{I}(\ell))$.

PROOF. By induction on the cardinality of L. □

LEMMA 18.8. *If $\{P\}c\{Q\}$, and if a heap h satisfies the predicate $(P * \circledast_{\ell\in L}\mathcal{I}(\ell))$, then an invariant of the system starting at state $(h, \varnothing, c)$ is: for reachable state (h', l', c'), there exists P' where $\{P'\}c'\{Q\}$, such that h' satisfies $(P' * \circledast_{\ell\in L}(\ell \notin l' \longrightarrow \mathcal{I}(\ell)))$.*

PROOF. By invariant induction, using Lemma 18.7 for the base case and Lemma 18.6 for the induction step, the latter with $R = \mathsf{emp}$. □

LEMMA 18.9 (Progress). *If $\{P\}c\{Q\}$ and c is about to fail, then P is unsatisfiable.*

PROOF. By induction on the derivation of $\{P\}c\{Q\}$. □

The overall soundness proof proceeds by invariant weakening with the invariant established by Lemma 18.8. We prove the inclusion of new invariant in old by Lemma 18.9.

CHAPTER 19

Process Algebra and Refinement

The last two chapters dealt with the most popular sort of concurrent programming, the threads-and-locks shared-memory style. It's a fundamentally imperative style, with side effects coordinating synchronization across threads. Another well-established (and increasingly popular) style is *message passing*, which is closer in spirit to functional programming. In that world, there is, in fact, no memory at all, let alone shared memory. Instead, state is incorporated into the text of thread code, and information passes from thread to thread by sending *messages* over *channels*. There are two main kinds of message passing. In the *asynchronous* or *mailbox* style, a thread can deposit a message in a channel, even when no one is ready to receive the message immediately. Later, a thread can come along and effectively dequeue the message from the channel. In the *synchronous* or *rendezvous* style, a message send only executes when a matching receive, on the same channel, is available immediately. The threads of the two complementary operations *rendezvous* and pass the message in one atomic step.

Packages of semantics and proof techniques for such languages are often called *process algebras*, as they support an algebraic style of reasoning about the source code of message-passing programs. That is, we prove laws very similar to the familiar equations of algebra, and use those laws to "rewrite" inside larger processes, by replacing their subprocesses with others we have shown suitably equivalent. It's a powerful technique for highly modular proofs, which we develop in the rest of this chapter for one concrete synchronous language. Well-known process algebras include the π-calculus and the Calculus of Communicating Systems; the one we focus on is idiosyncratic and designed partly to make the Coq proofs manageable.

19.1. An Object Language with Synchronous Message Passing

$$\begin{array}{lllll} \text{Channels} & c & & & \\ \text{Processes} & p & ::= & \nu[\vec{c}](x); p(x) \mid \mathsf{block}(c); p \mid !c(v); p \mid ?c(x); p(x) \mid p||p \mid \mathsf{dup}(p) \mid \mathsf{done} \end{array}$$

Here's the intuitive explanation of each syntax construction.

- **Fresh channel generation** $\nu[\vec{c}](x); p(x)$ creates a new *private* channel to be used by the body process $p(x)$, where we replace x with the channel that is chosen. Following tradition, we use the Greek letter ν (nu) for this purpose. Each generation operation takes a parameter $\vec{c}$, which we call the *support* of the operation. It gives a list of channels already in use for other purposes, so that the fresh channel must not equal any of them. (We assume an infinite domain of channels, so that, for any specific list, it is always possible to find a channel not in that list.)

Abstraction

- **Abstraction boundaries** $\mathsf{block}(c); p$ prevent "the outside world" from sending p any messages on channel c or receiving any messages from p via c. That is, c is treated as a local channel for p.
- **Sends** $!c(v); p$ and **receives** $?c(x); p(x)$, where we use an exclamation mark to suggest "telling something" and a question mark to suggest "asking something." Processes of these kinds can rendezvous when they agree on the channel. When $!c(v); p_1$ and $?c(x); p_2(x)$ rendezvous, they respectively evolve to p_1 and $p_2(v)$.
- **Parallel compositions** $p_1||p_2$ work as we're used to by now.
- **Duplications** $\mathsf{dup}(p)$ act just like infinitely many copies of p composed in parallel. We use them to implement nonterminating "server" processes that are prepared to respond to many requests over particular channels. In traditional process algebra, duplication fills the role that loops and recursion fill in conventional programming.
- **The inert process** done is incapable of doing anything at all. It stands for a finished program.

We give an operational semantics in the form of a *labeled transition system*, as we did to formalize output instructions for compiler correctness in Chapter 9. That is, we not only express how a step takes us from one state to another, but we also associate each step with a *label* that summarizes what happened. Our labels will include the *silent* label ϵ, read labels $?c(v)$, and write labels $!c(v)$. The latter two indicate that a thread has read a value from or written a value to channel c, respectively, and the parameter v indicates which value was read or written. We write $p_1 \xrightarrow{l} p_2$ to say that process p_1 steps to p_2 by performing label l. We use $p_1 \longrightarrow p_2$ as an abbreviation for $p_1 \xrightarrow{\epsilon} p_2$.

We start with the rules for sends and receives.

$$\frac{}{!c(v); p \xrightarrow{!c(v)} p} \qquad \frac{}{?c(x); p(x) \xrightarrow{?c(v)} p(v)}$$

They record the action in the obvious way, but there is already an interesting wrinkle: the rule for receives *picks a value v nondeterministically*. This nondeterminism is resolved by the next two rules, the rendezvous rules, which force a read label to match a write label precisely.

$$\frac{p_1 \xrightarrow{!c(v)} p_1' \quad p_2 \xrightarrow{?c(v)} p_2'}{p_1||p_2 \longrightarrow p_1'||p_2'} \qquad \frac{p_1 \xrightarrow{?c(v)} p_1' \quad p_2 \xrightarrow{!c(v)} p_2'}{p_1||p_2 \longrightarrow p_1'||p_2'}$$

A fresh channel generation can step according to any valid choice of channel.

$$\frac{c \notin \vec{c}}{\nu[\vec{c}](x); p(x) \longrightarrow \mathsf{block}(c); p(c)}$$

An abstraction boundary prevents steps with labels that mention the protected channel. (We overload notation $c \in l$ to indicate that channel c appears in the send/receive position of label l.)

$$\frac{p \xrightarrow{l} p' \quad c \notin l}{\mathsf{block}(c); p \xrightarrow{l} \mathsf{block}(c); p'}$$

Any step can be lifted up out of a parallel composition.

$$\frac{p_1 \xrightarrow{l} p_1'}{p_1||p_2 \xrightarrow{l} p_1'||p_2} \qquad \frac{p_2 \xrightarrow{l} p_2'}{p_1||p_2 \xrightarrow{l} p_1||p_2'}$$

Finally, a duplication can spawn a new copy ("thread") at any time.

$$\frac{}{\mathsf{dup}(p) \longrightarrow \mathsf{dup}(p)||p}$$

The labeled-transition-system approach may seem a bit unwieldy for just explaining the behavior of programs. Where it really pays off is in supporting a modular, algebraic reasoning style about processes, which we turn to next.

19.2. Refinement Between Processes

What sorts of correctness theorems should we prove about processes? The classic choice is to show that a more complex *implementation* process is a *safe substitute* for a simpler *specification* process. We will say that the implementation p *refines* the specification p'. Intuitively, such a claim means that any trace of labels that p could generate may also be generated by p', so that p has *no more behaviors* than p' has, though it may have fewer behaviors. (There is a formal connection lurking here to the notion of refinement from Chapter 16, where method calls there are analogous to channel operations here.) Crucially, in building traces of process executions, we ignore silent labels, only collecting the send and receive labels.

This condition is called *trace inclusion*, and, though it is intuitive, it is not strong enough to support all of the composition properties that we will want. Instead, we formalize refinement via *simulation*, very similarly to how we formalized compiler correctness in Chapter 9 and data abstraction in Chapter 16.

Abstraction

Definition 19.1. Binary relation R between processes is a *simulation* when these two conditions hold.

- **Silent steps match up**: when $p_1\ R\ p_2$ and $p_1 \longrightarrow p_1'$, there always exists p_2' such that $p_2 \longrightarrow^* p_2'$ and $p_1'\ R\ p_2'$.
- **Communication steps match up**: when $p_1\ R\ p_2$ and $p_1 \xrightarrow{l} p_1'$ for $l \neq \epsilon$, there always exist p_2'' and p_2' such that $p_2 \longrightarrow^* p_2''$, $p_2'' \xrightarrow{l} p_2'$, and $p_1'\ R\ p_2'$.

Intuitively, R is a simulation when, starting in a pair of related processes, any step on the left can be matched by a step on the right, taking us back into R. The conditions are naturally illustrated with commuting diagrams.

$$\begin{array}{ccc} p_1 & \xrightarrow{R} & p_2 \\ \downarrow{\scriptstyle\forall} & & \downarrow{\scriptstyle\exists} \\ p_1' & \xleftarrow[R^{-1}]{} & p_2' \end{array} \qquad \begin{array}{ccc} p_1 & \xrightarrow{R} & p_2 \\ \downarrow{\scriptstyle\forall}\xrightarrow{l} & & \downarrow{\scriptstyle\exists}\longrightarrow^*\xrightarrow{l} \\ p_1' & \xleftarrow[R^{-1}]{} & p_2' \end{array}$$

Invariants

Simulations have quite a lot in common with our well-worn concept of invariants of transition systems. Simulation can be seen as a kind of natural generalization of invariants, which are predicates over single states, into relations that apply to states of two different transition systems that need to evolve in (approximate) lock-step.

We define *refinement* $p_1 \preccurlyeq p_2$ to indicate that there exists a simulation R such that $p_1\ R\ p_2$. Luckily, this somewhat involved definition is easily related back to our intuitions.

THEOREM 19.2. *If $p_1 \preccurlyeq p_2$, then every trace generated by p_1 is also generated by p_2.*

PROOF. By induction on executions of p_1. □

Refinement is also a preorder.

THEOREM 19.3 (Reflexivity). *For all p, $p \preccurlyeq p$.*

PROOF. Choose equality as the simulation relation. □

THEOREM 19.4 (Transitivity). *If $p_1 \preccurlyeq p_2$ and $p_2 \preccurlyeq p_3$, then $p_1 \preccurlyeq p_3$.*

PROOF. The two premises respectively imply the existence of simulations R_1 and R_2. Set the new simulation relation as $R_1 \circ R_2$, defined to contain a pair (p, q) iff there exists r with $p\ R_1\ r$ and $r\ R_2\ q$. □

The accompanying Coq code includes several examples of verifying moderately complex processes, by manual tailoring of simulation relations. We leave those details to the code, turning now instead to further algebraic properties that allow us to *compose* laborious manual proofs about components, in a black-box way.

19.3. The Algebra of Refinement

We finish the chapter with a tour through some algebraic properties of refinement that are proved in the Coq source. We usually omit proof details here, though we work out one interesting example in more detail.

Perhaps the greatest pay-off from the refinement approach is that *refinement is a congruence for parallel composition*.

Modularity

THEOREM 19.5. *If $p_1 \preccurlyeq p_1'$ and $p_2 \preccurlyeq p_2'$, then $p_1||p_2 \preccurlyeq p_1'||p_2'$.*

This deceptively simple theorem statement packs a strong modularity punch! We can verify a component in isolation and then connect to an arbitrary additional component, immediately concluding that the composition behaves properly. The secret sauce, implicit in our formulation of the object language and refinement, is the labeled-transition-system style, where processes may generate receive labels nondeterministically. In this way, we can reason about a process implicitly in terms of *every value that some other process might send to it when they are composed*, without needing to quantify explicitly over all other eligible processes.

A similar congruence property holds for duplication, and we'll take this opportunity to explain a bit of the proof, in the form of choosing a good simulation relation.

THEOREM 19.6. *If $p \preccurlyeq p'$, then $\mathsf{dup}(p) \preccurlyeq \mathsf{dup}(p')$.*

PROOF. The premise implies the existence of a simulation R. We define a derived relation R^D with these inference rules.

$$\frac{p\ R\ p'}{p\ R^D\ p'} \qquad \frac{p\ R\ p'}{\mathsf{dup}(p)\ R^D\ \mathsf{dup}(p')} \qquad \frac{p_1\ R^D\ p_1' \quad p_2\ R^D\ p_2'}{p_1||p_2\ R^D\ p_1'||p_2'}$$

R^D is precisely the relation we need to finish the current proof. Intuitively, the challenge is that $\mathsf{dup}(p)$ includes infinitely many copies of p, each of which may evolve in a different way. It is even possible for different copies to interact with each other through shared channels. However, comparing intermediate states of $\mathsf{dup}(p)$ and $\mathsf{dup}(p')$, we expect to see a shared backbone, where corresponding threads are related by the original simulation R. The definition of R^D formalizes that intuition of a shared backbone with R connecting corresponding leaves. □

We wrap up the chapter with a few more algebraic properties, which the Coq code puts to good use in larger examples. We sometimes rely on a predicate $\mathsf{neverUses}(c, p)$, to express that, no matter how other threads interact with it, process p will never perform a send or receive operation on channel c.

THEOREM 19.7. *If* $p \leqslant p'$, *then* $\mathsf{block}(c); p \leqslant \mathsf{block}(c); p'$.

THEOREM 19.8. $\mathsf{block}(c_1); \mathsf{block}(c_2); p \leqslant \mathsf{block}(c_2); \mathsf{block}(c_1); p$

THEOREM 19.9. *If* $\mathsf{neverUses}(c, p_2)$, *then* $(\mathsf{block}(c); p_1 || p_2) \leqslant (\mathsf{block}(c); p_1) || p_2$.

THEOREM 19.10 (Handoff). *If* $\mathsf{neverUses}(c, p(v))$, *then* $(\mathsf{block}(c); (!c(v); \mathsf{done}) || \mathsf{dup}(?c(x); p(x))) \leqslant p(v)$.

That last theorem is notable for how it prunes down the space of possibilities given an infinitely duplicated server, where each thread is trying to receive from a channel. If server threads never touch that channel after their initial receives, then most server threads will remain inert. The one send $!c(v); \mathsf{done}$ is the only possible source of interaction with server threads, thanks to the abstraction barrier on c, and that one send can only awaken one server thread. Thus, the whole composition behaves just like a single server thread, instantiated with the right input value.

A concrete example of the Handoff theorem in action is a refinement like this one, applying to a kind of forwarding chain between channels:

$$p = \mathsf{block}(c_1); \mathsf{block}(c_2); !c_1(v); \mathsf{done} || \mathsf{dup}(?c_1(x); !c_2(x); \mathsf{done}) || \mathsf{dup}(?c_2(y); !c_3(y); \mathsf{done})$$
$$p \leqslant !c_3(v); \mathsf{done}$$

Note that, without the abstraction boundaries at the start, this fact would not be derivable. We would need to worry about meddlesome threads in our environment interacting directly with c_1 or c_2, spoiling the protocol and forcing us to add extra cases to the righthand side of the refinement.

CHAPTER 20

Session Types

Process algebra, as we met it last chapter, can be helpful for modeling network protocols. Here, multiple *parties* step through a script of exchanging messages and making decisions based on message contents. A buggy party might introduce a *deadlock*, where, say, party A is blocked waiting for a message from party B, while B is also waiting for A. *Session types* are a style of static type system that rule out deadlock while allowing convenient separate checking of each party, given a shared protocol type.

There is an almost unlimited variation of different versions of session types. We still step through a progression of three variants here, and even by the end there will be obvious protocols that don't fit the framework. Still, we aim to convey the core ideas of the approach.

20.1. Basic Two-Party Session Types

Each of our type systems will apply to the object language from the prior chapter. Assume for now that a protocol involves exactly two parties. Here is a simple type system, explaining a protocol's "script" from the perspective of one party.

$$\begin{array}{rrcl} \text{Base types} & \sigma & & \\ \text{Session types} & \tau & ::= & !c(\sigma); \tau \mid ?c(\sigma); \tau \mid \mathsf{done} \end{array}$$

Abstraction

We model simple parties with no internal duplication or parallelism. A session type looks like an abstracted version of a process, remembering only the *types* of messages exchanged on channels, rather than their *values*. A simple set of typing rules makes the connection.

$$\frac{v : \sigma \quad p : \tau}{!c(v); p : !c(\sigma); \tau} \qquad \frac{\forall v : \sigma.\ p(v) : \tau}{?c(x); p(x) : ?c(\sigma); \tau} \qquad \frac{}{\mathsf{done} : \mathsf{done}}$$

The only wrinkle in these rules is the use of universal quantification for the receive rule, to force the body to type-check under any well-typed value read from the channel. Actually, such proof obligations may be nontrivial when we encode this object language in the mixed-embedding style of Section 13.1, where the body p in the rule could include arbitrary metalanguage computation, to choose a body based on the value v read from the channel.

The associated Coq code demonstrates tactics to deal with that complication, for automatic type-checking of concrete programs. That code is also where we keep all of our concrete examples of object-language programs.

For the rest of this chapter, we will interpret last chapter's object language as a transition system with one small change: we only allow silent steps. That is, we only model whole programs, with no communication with "the environment." As a result, we consider self-contained protocols.

A satisfying soundness theorem applies to our type system. To state it, we first need the crucial operation of *complementing* a session type.

$$\begin{aligned}\overline{!c(\sigma);\tau} &= ?c(\sigma);\overline{\tau}\\ \overline{?c(\sigma);\tau} &= !c(\sigma);\overline{\tau}\\ \overline{\mathsf{done}} &= \mathsf{done}\end{aligned}$$

Modularity

It is apparent that complementation just swaps the sends and receives. When the original session type tells one party what to do, the complement type tells the other party what to do. The power of this approach is that we can write one global protocol description (the session type) and then check two parties' code against it separately. A new version of one party can be dropped in without rechecking the other party's code.

Using complementation, we can give succinct conditions for deadlock freedom of a pair of parties.

THEOREM 20.1. *If $p_1 : \tau$ and $p_2 : \overline{\tau}$, then it is an invariant of $p_1||p_2$ that an intermediate process is either* done||done *or can take a step.*

PROOF. By invariant induction, after strengthening the invariant to say that any intermediate process takes the form $p_1'||p_2'$, where, for some type τ', we have $p_1' : \tau'$ and $p_2' : \overline{\tau'}$. The inductive case of the proof proceeds by simple inversion on the derivation of $p_1' : \tau'$, where by the definition of complement it is apparent that any communication p_1' performs has a matching action at the start of p_2'. The choice of τ' changes during such a step, to the "tail" of the old τ'. □

20.2. Dependent Two-Party Session Types

It is a boring protocol that follows such a regular communication pattern as our first type system accepts. Rather, it tends to be crucial to change up the expected protocol steps, based on *values* sent over channels. It is natural to switch to a *dependent* type system to strengthen our expressiveness. That is, a communication type will allow its body type to depend on the value sent or received.

$$\text{Session types} \quad \tau \quad ::= \quad !c(x : \sigma);\tau(x) \mid ?c(x : \sigma);\tau(x) \mid \mathsf{done}$$

Each nontrivial construct does more than give the base type that should be sent or received on or from a channel. We also bind a variable x, to stand for the value sent or received. It may be unintuitive that we must introduce a binder even for sends, when the sender is in control of which value will be sent. The reason is that we must allow the sender to then continue with different subprotocols for different values that might be sent. We should not force the sender's hand by fixing a value in advance, when that value might depend on arbitrary program logic.

Very little change is needed in the typing rules.

$$\frac{v : \sigma \quad p : \tau(v)}{!c(v); p : !c(x : \sigma);\tau(x)} \qquad \frac{\forall v : \sigma.\ p(v) : \tau(v)}{?c(x); p(x) : ?c(x : \sigma);\tau(x)} \qquad \frac{}{\mathsf{done} : \mathsf{done}}$$

Our deadlock-freedom property is easy to reestablish.

THEOREM 20.2. *If $p_1 : \tau$ and $p_2 : \overline{\tau}$, then it is an invariant of $p_1||p_2$ that an intermediate process is either* done||done *or can take a step.*

PROOF. Literally the same Coq proof script as for Theorem 20.1! □

20.3. Multiparty Session Types

New complications arise when more than two parties are communicating in a protocol. The Coq code demonstrates a case of an online merchant, a customer sending it orders, and a warehouse being queried by the merchant to be sure a product is in stock. Many other such examples appear in the real world.

Now it is no longer possible to start from one party's view of a protocol and compute any other party's view. The reason is that each message only involves two parties. Any other party will not see that message in its own session type, making it impossible to preserve that message in a complement-like operation.

Instead, we define one global session type that includes only "send" operations. However, we name the parties and parameterize on a mapping $\mathcal{C}$ from channels to unique parties that own their send and receive ends. That is, for any given channel and operation on it (send and receive), precisely one party is given permission to perform the operation – and indeed, when the time comes, that party is *obligated* to perform the operation, to avoid deadlock.

With that view in mind, our type language gets even simpler.

$$\text{Session types} \quad \tau \quad ::= \quad !c(x : \sigma); \tau(x) \mid \mathsf{done}$$

We redefine the typing judgment as $p :_{\alpha,b} \tau$. Here α is the identifier of the party running p, and b is a Boolean that, when set, enforces that p's next action (if any) is a receive.

$$\frac{v : \sigma \quad \mathcal{C}(c) = (\alpha, \beta) \quad \beta \neq \alpha \quad p :_{\alpha,\bot} \tau(v)}{!c(v); p :_{\alpha,\bot} !c(x : \sigma); \tau(x)}$$

$$\frac{\mathcal{C}(c) = (\beta, \alpha) \quad \beta \neq \alpha \quad \forall v : \sigma.\, p(v) :_{\alpha,\bot} \tau(v)}{?c(x); p(x) :_{\alpha,b} !c(x : \sigma); \tau(x)}$$

$$\frac{\mathcal{C}(c) = (\beta, \gamma) \quad \beta \neq \alpha \quad \gamma \neq \alpha \quad \forall v : \sigma.\, p :_{\alpha,\top} \tau(v)}{p :_{\alpha,b} !c(x : \sigma); \tau(x)}$$

$$\frac{}{\mathsf{done} :_{\alpha,b} \mathsf{done}}$$

The first two rules encode the simple cases where the current party α is one of the two designated to step next in the protocol, as we verify by looking up the channel in $\mathcal{C}$. It is important that the send and receive ends of the channel are owned by different parties, or we would clearly have a deadlock, as that party would either wait forever for a message from itself or try futilely to send itself a message! The $\neq$ premises enforce that condition. Also, the Boolean subscript enforces that we cannot be running a send operation if we have been instructed to run a receive next. That flag is reset to false in the recursive premises, since we only use the flag to express an obligation for the very next command.

The third rule is crucial: it applies to a process that is not participating in the next step of the protocol. That is, we look up the owners of the channel that comes next, and we verify that neither owner is α. In this case, we merely proceed to the next protocol step, leaving the process unchanged. Crucially, we must be prepared for any value that might be exchanged in this skipped step, even though we do not see it ourselves.

Why does the last premise of the third rule set the Boolean flag, forcing the next action to be a receive? Otherwise, at some point in the protocol, we could have multiple parties trying to send messages. In such a scenario, there might not be a unique step that the composed parties can take. The proofs are easier if we can assume deterministic execution within a protocol, which is why we introduced this static restriction.

To amend our theorem statement, we need to characterize when a process implements a set of parties correctly. We use the judgment $p :_{\vec{\alpha}} \tau$ to that end, where p is the process, $\vec{\alpha}$ is a list of all the involved parties, and τ is the type they must follow collectively.

$$\frac{}{\mathsf{done} :_{[]} \tau} \qquad \frac{p_1 :_{\alpha,\bot} \tau \quad p_2 :_{\vec{\beta}} \tau}{p_1 || p_2 :_{\alpha \bowtie \vec{\beta}} \tau}$$

The heart of the proof is demonstrating the existence of a unique sequence of steps to a point where all parties are done. Here is a sketch of the key lemmas.

LEMMA 20.3. *If $p :_{\vec{\alpha}} \mathsf{done}$, then p can't take any silent step.*

PROOF. By induction on any derivation of a silent step, followed by inversion on $p :_{\vec{\alpha}} \mathsf{done}$. □

LEMMA 20.4. *If $p :_{\vec{\alpha}} !c(x : \sigma); \tau(x)$ and at least one of sender or receiver of channel c is missing from $\vec{\alpha}$, then p can't take any silent step.*

PROOF. By induction on any derivation of a silent step, followed by inversion on $p :_{\vec{\alpha}} !c(x : \sigma); \tau(x)$. □

LEMMA 20.5. *Assume that $\vec{\alpha}$ is a duplicate-free list of parties excluding both sender and receiver of channel c. If $p :_{\vec{\alpha}} !c(x : \sigma); \tau(x)$, then for any $v : \sigma$, we have $p :_{\vec{\alpha}} \tau(v)$. In other words, when we have well-typed code for a set of parties that do not participate in the first step of a protocol, that code remains well-typed when we advance to the next protocol step.*

PROOF. By induction on the derivation of $p :_{\vec{\alpha}} !c(x : \sigma); \tau(x)$. □

LEMMA 20.6. *Assume that $\vec{\alpha}$ is a duplicate-free list of parties, at least comprehensive enough to include the sender of channel c. However, $\vec{\alpha}$ should* exclude *the receiver of c. If $p :_{\vec{\alpha}} !c(x : \sigma); \tau(x)$ and $p \xrightarrow{!c(v)} p'$, then $p' :_{\vec{\alpha}} \tau(v)$.*

PROOF. By induction on steps followed by inversion on multiparty typing. As we step through elements of $\vec{\alpha}$, we expect to "pass" parties that do not participate in the current protocol step. Lemma 20.5 lets us justify those passings. □

THEOREM 20.7. *Assume that $\vec{\alpha}$ is a duplicate-free list of* all *parties for a protocol. If $p :_{\vec{\alpha}} \tau$, then it is an invariant of p that an intermediate process is either inert (made up only of* done*s and parallel compositions) or can take a step.*

PROOF. By invariant induction, after strengthening the invariant to say that any intermediate process p' satisfies $p' :_{\vec{\alpha}} \tau'$ for some τ'. The inductive case uses Lemma 20.3 to rule out steps by finished protocols, and it uses Lemma 20.4 to rule out cases that are impossible because parties that are scheduled to go next are not present in $\vec{\alpha}$. Interesting cases are where we find that one of the active parties is at the head of $\vec{\alpha}$. That party either sends or receives. In the first case, we appeal

to Lemma 20.6 to find a receiver among the remaining parties. In the second case, we appeal to an analogous lemma (not stated here) to find a sender.

The other crucial case of the proof is showing that existence of a multiparty typing implies that, if a process is not inert, it can take a step. The reasoning is quite similar to in the inductive case, but where instead of showing that any possible step preserves typing, we demonstrate that a particular step exists. The head of the session type telegraphs what step it is: for the communication at the head of the type, the assigned sending party sends to the assigned receiving party. □

APPENDIX A

The Coq Proof Assistant

Coq is a proof-assistant software package developed as open source, primarily by Inria, the French national computer-science lab.

A.1. Installation and Basic Use

The project home page is:

`https://coq.inria.fr/`

The code associated with this book is designed to work with Coq versions 8.9 and higher. The project Web site makes a number of versions available, and versions are also available in popular OS package distributions, along with binaries for platforms where open-source package systems are less common. We assume that readers have installed Coq by one of those means or another. It will also be almost essential to use some graphical interface for Coq editing. The author prefers Proof General, an Emacs mode:

`http://proofgeneral.inf.ed.ac.uk/`

It should be possible to follow along using CoqIDE, a standalone tool distributed with Coq itself, but we will not give any CoqIDE-specific instructions.

The Proof General instructions are simple: after installing, within a regular Emacs session, open a file with the Coq extension `.v`. Move the point (cursor) to a position where you would like to examine the current state of a proof, etc. Then press C-C C-RET ("control-C, control-enter") to run Coq up to that point. Several display panes will open, showing different aspects of Coq's state, any error messages it wants to report, etc. This feature is the main workhorse of Proof General. It can be used both to move *forward*, checking that Coq accepts a command; and to move *backward*, to undo commands processed previously.

Proof General has plenty of other bells and whistles, but we won't go into them here.

A.2. Tactic Reference

Tactics are the commands run in Coq to advance the state of a proof, corresponding to deduction steps at different granularities. Here we collect all of the short explanations of tactics that appear in Coq source files associated with the chapters included in this document. Note that many of these are specific to the `Frap` library distributed with this book, where built-in tactics often do quite similar things, but in a way that the author judges to be more of a hassle for beginners.

`apply` H`:` For H a hypothesis or previously proved theorem, establishing some fact that matches the structure of the current conclusion, switch to proving H's own hypotheses. This is *backwards reasoning* via a known fact.

`apply` H `with (`x_1 `:=` e_1`) ... (`x_n `:=` e_n`):` Like the last one, supplying values for quantified variables in H's statement, especially for those variables whose values aren't immediately implied by the current goal.

`apply` H_1 `in` H_2`:` Like `apply` H_1, but used in a *forward* direction rather than *backward*. For instance, if H_1 proves $P \Rightarrow Q$ and H_2 proves P, then the effect is to change H_2 to Q.

`assert` P`:` First prove proposition P, then continue with it as a new hypothesis.

`assumption:` Prove a conclusion that matches a hypothesis exactly.

`cases` e`:` Break the proof into one case for each constructor that might have been used to build the value of expression e. In the special case where e essentially has a Boolean type, we consider whether e is true or false.

`constructor:` When proving an instance of an inductive predicate, `apply` the first matching rule of that predicate.

`eapply` H`:` Like `apply` but will work even when some quantified variables from H do not have their values determined immediately by the form of the goal. Instead, *existential variables* (with names starting with question marks) are introduced for those values.

`eassumption:` Like `assumption` but will figure out values of existential variables.

`econstructor:` When proving an instance of an inductive predicate, `eapply` the first matching rule of that predicate.

`eexists:` To prove $\exists x.\ P(x)$, switch to proving $P(?y)$, for a new existential variable $?y$.

`equality:` A complete decision procedure for the theory of equality and uninterpreted functions. That is, the goal must follow from only reflexivity, symmetry, transitivity, and congruence of equality, including that functions really do behave as functions. See Section 2.4.

`exfalso:` From any proof state, switch to proving `False`. In other words, indicate a switch to a proof by contradiction.

`exists` e`:` Prove $\exists x.\ P(x)$ by proving $P(e)$.

`first_order:` Simplify a goal into zero or more new goals, based on the rules of first-order logic alone. *Warning:* this tactic is especially likely to run forever, on complex enough goals! (While entailment for propositional logic is decidable, entailment for first-order logic isn't.)

`f_equal:` When the goal is an equality between two applications of the same function, switch to proving that the function arguments are pairwise equal.

`induct` x`:` Where x is a variable in the theorem statement, structure the proof by induction on the structure of x. You will get one generated subgoal per constructor in the inductive definition of x. (Indeed, it is required that x's type was introduced with `Inductive`.)

`invert` H`:` Replace hypothesis H with other facts that can be deduced from the structure of H's statement. More detail to be added here soon!

`linear_arithmetic:` A complete decision procedure for linear arithmetic. Relevant formulas are essentially those built up from variables and constant natural numbers and integers using only addition and subtraction, with equality and inequality comparisons on top. (Multiplication by constants is supported, as a shorthand for repeated addition.) See Section

2.4. Also note that this tactic goes a bit beyond that theory, by (1) converting multivariable terms into a standard polynomial form and then (2) treating each different product of powers of variables as one variable in a linear-arithmetic problem. So, for instance, `linear_arithmetic` can prove $x \times y = y \times x$ simply by deciding that a new variable $z = x \times y$, rewriting the goal to $z = z$ after putting polynomials in canonical form (in this case, commuting argument order in products to make it consistent).

`left`: Prove a disjunction by proving its left side.

`maps_equal`: Prove that two finite maps are equal by considering all the relevant cases for mappings of different keys.

`propositional`: Simplify a goal into zero or more new goals, based on the rules of propositional logic alone.

`replace` e_1 `with` e_2 `by tac`: Replace occurrences of e_1 with e_2, proving $e_2 = e_1$ with tactic `tac`.

`rewrite` H: Where H is a hypothesis or previously proved theorem, establishing `forall x1 .. xN, e1 = e2`, find a subterm of the goal that equals `e1`, given the right choices of `xi` values, and replace that subterm with `e2`.

`rewrite` H_1 `in` H_2: Like `rewrite` H_1 but performs the rewrite in hypothesis H_2 instead of in the conclusion.

`right`: Prove a disjunction by proving its right side.

`ring`: Prove goals that are equalities over some registered ring or semiring, in the sense of algebra, where the goal follows solely from the axioms of that algebraic structure. See Section 2.4.

`simplify`: Simplify throughout the goal, applying the definitions of recursive functions directly. That is, when a subterm matches one of the `match` cases in a defining `Fixpoint`, replace with the body of that case, then repeat.

`subst`: Remove all hypotheses like $x = e$ for variables x, simply replacing all uses of x by e.

`symmetry`: When proving $X = Y$, switch to proving $Y = X$.

`transitivity` X: When proving $Y = Z$, switch to proving $Y = X$ and $X = Z$.

`trivial`: Coq maintains a database of simple proof steps, such as proving a fact by direct appeal to a matching hypothesis. `trivial` asks to try all such simple steps.

`unfold` X: Replace X by its definition.

`unfold` X `in *`: Like the last one, but unfolds in hypotheses as well as conclusion.

A.3. Further Reading

For more Coq information, we recommend a few books (beyond the Coq reference manual). Some focus purely on introducing Coq:

- Adam Chlipala, *Certified Programming with Dependent Types*, MIT Press, `http://adam.chlipala.net/cpdt/`
- Yves Bertot and Pierre Castéran, *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*, Springer, `https://www.labri.fr/perso/casteran/CoqArt/`

The first of these two, especially, goes in-depth on the automated proof-scripting principles showcased from time to time in the Coq example code associated with the present book.

There are also other sources that introduce program-reasoning principles at the same time, including:

- Benjamin C. Pierce et al., *Software Foundations*, `http://www.cis.upenn.edu/~bcpierce/sf/`

Software Foundations generally proceeds at a slower pace than this book does.

Index