Algebraic Approaches to Program Verification with Isabelle/HOL

—Lecture Notes—

Georg Struth

g.struth@sheffield.ac.uk

Contents

Preface

These are lecture notes for a course on *Algebraic Approaches to Program Verification with Isabelle/HOL* with Kleene algebras and similar formalisms and via simple denotational semantics. I am using them for teaching program verification at the University of Sheffield, and have used part of this material for similar courses at École polytechnique and ENS de Lyon. The notes are work in progress and may contain typos and errors. They will be updated and corrected during the course—I am very grateful for comments and suggestions. Please email them to my University of Sheffield address.

Other sources are not yet adequately acknowledged or cited. The underlying algebraic approach has been inspired by John Conway's book on regular algebras, applications of regular algebra to program construction and verification by Roland Backhouse, Dexter Kozen's work on Hoare logic and Kleene algebra with tests, and my own work on modal Kleene algebras with Jules Desharnais, Bernhard Möller and others. The approach to designing verification components is my own. It has been developed and refined in collaboration with Alasdair Armstrong, Victor Gomes and Jonathan Julián Huerta y Munive. Yet it is strongly influenced by Mike Gordon's pioneering work on program verification with interactive theorem provers and his beautiful Cambridge lecture notes on Hoare Logic.

Last changes: 10.10.2022

© Georg Struth, 2020; all rights reserved.

Georg Struth
Sheffield in Spring 2020
Paris in Spring 2021
Lyon in Autumn 2022

CHAPTER 1

Introduction

Seek simplicity and distrust it.
— A. N. Whitehead

1.1. What is Program Correctness?

The following classical algorithm takes the natural numbers x and y, a dividend and a divisor, as its input, and it computes the natural numbers q and r, their quotient and remainder, as its output:

$$
\begin{array}{l}
q := 0; \\
r := x; \\
\textbf{while } y \leq r \textbf{ do} \\
\quad q := q + 1; \\
\quad r := r - y
\end{array}
$$

Understanding this program is easy. It subtracts y from x as long as possible, counts the number of subtraction to determine q and makes r the result of the repeated subtraction just before it would become negative. Think about cutting a piece of rope of length x into pieces of length y, counting the number q of y-pieces obtained and measuring the length r of the leftover piece, which could be zero. So who would doubt that this program is correct? But who could explain precisely why? Or what program correctness even means in general terms?

As a first step towards answers we specify more precisely what our program is meant to compute. Obviously, the relation $x = q \cdot y + r$ must hold after the program has terminated. But as it holds already after q and r have been initialised, we need to be more specific: q must be the *largest* quotient and r the *smallest* non-negative remainder for which $x = q \cdot y + r$ after termination. We thus require that $r < y$ holds as well.

Then of course we need to specify which natural numbers x and y should be allowed as inputs. While it seems reasonable to allow any natural number as a dividend, the divisor should certainly not be 0, because otherwise the program would not terminate: the value of r in the body of the loop r would always be x and the test $y \leq r$ of the loop would always remain true. It goes without saying that termination is important for program correctness as well.

Hence we deem our program correct if it terminates and its output values satisfy $x = q \cdot y + r$ and $r < y$ whenever it is executed from initial values x and $y \neq 0$. For obvious reasons, $y \neq 0$ is called the *precondition* and $x = q \cdot y + r \wedge r < y$ the *postcondition* of our program. More generally, a program is correct if it terminates

and does so in states that satisfy its postcondition whenever it starts from states satisfying its precondition.

Now that we know what program correctness means, and that we have a correctness specification for our particular program, we turn to the question how we can prove program correctness.

Verifying that our particular program terminates is easy: each iteration of the while-loop decreases r strictly if $y \neq 0$. It must therefore become strictly smaller than y after finitely many iterations. The test of the loop then becomes false and the loop stops executing.

Verifying that its postcondition $x = q \cdot y + r$ and $r < y$ holds after termination, whenever the precondition $y \neq 0$ holds, is less straightforward. Simply testing the program using some admissible input values is of course not good enough: we might miss precisely those for which the postcondition fails. Verification requires considering *all* admissible inputs, which is often more than a finite enumeration, and therefore requires more abstract ways of reasoning.

A natural way of reasoning about loops such as

$$\textbf{while } y \leq r \textbf{ do } q := q + 1 \ ; \ r := r - y$$

is by induction on the number of repetitions. This requires a property that holds before its execution (the base case) as well as before and after the execution of its body in each repetition (the induction step). Hence we need to check this property after initialising $q := 0$ and $r := x$ whenever $y \neq 0$ and verify that it continues to hold each time $q := q+1$ and then $r := r-y$ are executed, so long as the test $y \leq r$ of the loop remains true. Finally, when the loop terminates because $r < y$, we need to show that our property implies the postcondition. A property that continues to hold while the state of a system changes is an invariant. Here, more specifically, we need a *loop invariant*.

For our particular program, $x = q \cdot y + r$ is a natural loop invariant. We already know that it holds after initialisation, immediately before the loop is executed. Further, if it holds before an execution of the body of the loop, then it must also hold afterwards because $(q + 1) \cdot y + (r - y) = q \cdot y + r$, when we first increment q and then subtract y from r. Finally, it implies the postcondition $x = q \cdot y + r$ and $r < y$ because $x = q \cdot y + r$ *is* the loop invariant and $r < y$ *is* the negation of the test of the loop.

In sum, we have now given a correctness specification for our program and explained its correctness in a semiformal way. We have argued that the loop in our program terminates and that it does so in states in which the postcondition holds, whenever it is executed from states in which the precondition holds. Reasoning about loops required finding a loop invariant.

The calculations involved in our correctness proofs are certainly a big step towards precision. It remains to turn them into a method that works for any imperative program—or at least for many imperative programs. Ultimately, we would even like to execute this method on a machine to rule out human error in complex verification tasks that require reasoning about complex data structures or data domains or complicated corner cases (how does our division algorithm and calculation handle the case $x < y$, by the way?).

The formal verification of our simple division algorithm is also of mathematical interest. It is part of the proof of the classical theorem that for all natural numbers x and $y \neq 0$ there exist two unique natural numbers q and r such that $x = q \cdot y + r$ and

$r < y$.[1] The proof of existence is simply the pair (q, r) computed by the division algorithm for every input (x, y) together with its correctness proof. Uniqueness of q and r, however, is not guaranteed by the algorithm. So suppose that some other pair (q', r') satisfies $x = q' \cdot y + r'$ and $r' < y$. But then $|r' - r| < y$ and $|q - q'| \cdot y = |r' - r|$, which can only be the case if $q = q'$ and $r = r'$.

1.2. Overview of Content

Chapter 2 introduces an abstract algebraic semantics for programs based on a notion of program equivalence that considers programs as equal if they have the same input-output behaviour. Starting from intuitive algebraic laws, it introduces a series of algebraic structures—monoids, semilattices, Kleene algebras and Kleene algebras with tests—which encapsulate them. Kleene algebras with tests, in particular, yield an algebraic semantics for simple while-programs with sequential composition, conditionals and while loops, but disregarding variable assignments or structured tests. A first verification application of Kleene algebra with tests considers the correctness of simple program transformations.

The entire content of Chapter 2 can be formalised with proof assistants. We use the Isabelle/HOL proof assistant as an example. Chapter 3 explains how the algebraic hierarchy from Chapter 2 can be engineered using Isabelle's axiomatic type classes, and how simple models for these algebras can be integrated. This chapter also serves as a first introduction to formalised mathematics with Isabelle. It complements more traditional approaches that start from a functional programming perspective.

In Chapter 4, two more concrete semantics of program executions are introduced. The first one models programs as binary relations between input and output states, the second one as non-deterministic functions, or state transformers, that map states to sets of states. These form isomorphic models of Kleene algebra with tests. The content of this chapter is formalised in Chapter 5, except for an additional trace semantics, which we outline at the end of Chapter 4.

After this mathematical groundwork, we develop an algebraic variant of Hoare logic in Chapter 6. We show how partial program correctness specifications (assuming program termination instead of asking to prove it) can be expressed in Kleene algebras with tests and we derive algebraic variants of the classical rules of propositional Hoare logic—disregarding assignment laws—by simple equational reasoning. We also derive variants in the relational and state transformer semantics that are more suitable for automated verification condition generation with Isabelle and summarise the Isabelle formalisation of the material in this chapter.

A full Hoare logic is introduced in Chapter 7. It is developed in the concrete relational and state transformer semantics of the program store. We model program stores simply as functions from program variables to values that range over arbitrary data domains. We use a store update function to assign relational and state transformer semantics to variable assignment commands. Assignment rules in the style of Hoare logic are then derivable in these concrete program semantics. With a full Hoare logic in place, we can start verifying programs. We present a number of verification examples using proof outlines on paper as well as Isabelle proofs on a machine.

[1] The theorem is usually stated for integers, yet we stated our algorithm for positive numbers only.

Chapter 8 briefly introduces an alternative approach to program verification: the construction of programs from specifications using program refinement laws. To this end we introduce a refinement Kleene algebra with tests and derive simple refinement laws inspired by Carroll Morgan's refinement calculus, as explained in his book *Programming from Specifications*, in this algebra. After setting up the relational and state transformer semantics for refinement Kleene algebra with tests and deriving refinement laws for assignments in the concrete program store semantics, we show how programs can be developed from specifications that are correct by construction. Once again we formalise this approach with Isabelle.

Finally, we study a more powerful approach to program verification based on modal Kleene algebras in Chapter 9. This leads not only to more computational verification laws and verification conditions. It also increases flexibility, allowing for instance the integration of symbolic execution approaches and those computing weakest liberal preconditions. As an application of this formalism, we present an algebraic relative completeness proof of Hoare logic. As always, we discuss the Isabelle formalisation of this approach and present verification examples.

These lecture notes are complemented by an Isabelle formalisation of the entire mathematical content, up to very minor omissions. These Isabelle theories should be studied alongside this text. In the handouts of the theories provided, many theorems are shown without proofs in order to provide exercises for reasoning with the algebras formalised, their models, and finally for verifying simple programs. This formalisation shows in particular how program verification components can be built in a simple and principled way with proof assistants.

Using this approach, the entire formalisation, from the algebras and their models to the verification and refinement components, and the program verification examples, is provably correct relative to the core inference engine of the proof assistant. As the approach formalises a semantics of programs instead of starting syntactically from a programming language, it is based on mathematical objects such as functions, relations, predicates and various algebras, many of which are already well supported in proof assistants. This is far less involved than formalising grammars and interpretation functions for programming languages, let alone implementing program verification tools from scratch.

Instead, our approach brings us to the mathematical foundations of program correctness rather quickly: the algebras tell us quite generally when programs are equal and when one is a refinement of another; they also describe the laws that determine the behaviour of program constructs like sequential compositions, conditionals and loops, and allow us to express correctness specifications. Their relational or state transformer models over program stores capture the basic commands of programs in terms of updates to the program store. And those who like program syntax and interpretation maps can easily add them if they wish.

1.3. Background on Interactive Theorem Proving

Almost the entire mathematical development in these lecture notes, and in particular all verification examples, have been formalised with the Isabelle proof assistant, more precisely with Isabelle/HOL.

HOL stands for *Higher-Order Logic*, a powerful logic in which much of mathematics can be formalised, including the algebraic structures from monoids to KAT and the concrete semantics based on relations, state and predicate transformers.

Beyond first-order logic, it allows quantifications over functions and predicates. The logic is also typed like a functional programming language. Apart from formalising mathematics, interactive theorem provers have been used very successfully for verifying programs and software systems—in academic case studies and increasingly in industrial applications. Isabelle can be downloaded here:

https://isabelle.in.tum.de

A direct link to tutorials and reference manuals is

https://isabelle.in.tum.de/documentation.html

A tutorial to programming and proving with Isabelle can be found here:

https://isabelle.in.tum.de/dist/Isabelle2021-1/doc/prog-prove.pdf

Additional material can be found in Nipkow and Klein's book *Concrete Semantics*. We will learn to work with Isabelle as we go along. By contrast to traditional approaches to teaching Isabelle, which start from a (functional) programmer's point of view, we begin by declaring mathematical specifications and performing proofs of increasing complexity. Nevertheless, this introduction is *not* fully self-contained and the Isabelle documentation should be consulted alongside.

The Isabelle theory files complementing these lecture notes can be found at

https://github.com/gstruth/verisa

They contain holes that can be filled in as exercises.

CHAPTER 2

An Algebra of Programs

2.1. Intuitive Semantics for a Simple while-Language

Before introducing methods for verifying programs we need to explain what kind of mathematical objects programs are. We start with an abstract algebra for programs and refine it step by step to concrete semantics of program executions over a simple program store model. Algebra manipulates objects and expresses their relationships through equations. The equations $x+y = y+x$ or $x\cdot(y+z) = x\cdot y+x\cdot z$, for instance, can be used for reasoning with numbers, matrices or elements of a boolean algebra. Here we introduce such laws to reasons about programs, starting from intuition and then at increasing levels of formality and precision.

First we need to be precise about the programs considered. These are simple while-programs, as defined by the grammar

$$C \;::=\; x := e \mid C \,;\, C \mid \textbf{if } P \textbf{ then } C \textbf{ else } C \mid \textbf{while } P \textbf{ do } C\,.$$

The letter C indicates that we define the *commands* of this programming language. Assignment commands $x := e$ are basic commands, sequential compositions, conditionals and while loops are composite commands. The letters x, e and P do not represent commands: x represents a program variable, e an expression and P a test. Variables are elements of some suitable set; expressions and tests can be defined using other grammars. We return to them in Chapter 7. A common name for a simple while language with arithmetic expressions, interpreted in the natural numbers, and a simple boolean algebra of tests based comparisons for equality and inequality of arithmetic expressions is Imp. For now we consider assignments and tests as unstructured commands, and we simply identify programs and commands. Moreover, we do not consider this programming language directly, but start with an algebraic semantics that interprets commands, which we now simply call "programs", as abstract mathematical objects.

The most fundamental notion of algebra is equality or equivalence of objects. Our algebra of programs therefore requires a notion of *program equivalence*, which describes how we can identify and distinguish programs. Imperative programs act on state spaces by transforming input states into output states. We simply assume that they are completely determined by this input/output behaviour and ignore internal implementation details. Hence we deem two programs equivalent if they compute the same outputs from the same inputs; if their action on the state space has the same effect. There are of course more fine-grained notions of program equivalence—ours could not even distinguish a correct versions of quicksort from a correct version of merge sort. But for program verification, where we wish to guarantee that a program that satisfies a certain precondition in its input states satisfies a certain postcondition in its output states (whenever it terminates), our notion is good enough.

We abstract further and allow that programs may compute several outputs from a given input, or no output at all. Programs can thus be nondeterministic. This has several benefits. One is that we may represent the output of a program that fails to execute from some state by the empty set. Another one is that it yields a uniform view on programs and specifications, which need not be deterministic and not executable. A final abstraction is that we consider tests as special programs that compute or observe certain properties of states without changing them.

2.2. Algebraic Laws for Structural Commands

Based on these semantic intuitions, we now introduce an algebra of programs as an abstract program semantics. First we simply postulate its equational laws, then we organise them into an algebra—a Kleene algebra with tests. Concrete semantics of programs that transform a program store are developed as models of this algebra in later chapters. Yet most of the work for building program verification components can be done in algebra, by simple equational reasoning.

First we even ignore tests and generalise from conditionals to nondeterministic choices between programs and from while loops to finite unbounded iterations of programs. In addition, we consider two special programs: 1 or *skip*, which can be executed from any state without changing it, and 0 or *abort*, which can never be executed and thus computes no output at all.

We henceforth write $x, y, z, \ldots$ for programs and $=$ for program equivalence.

2.2.1. Sequential Composition. In a sequential composition $x\,;y$ of programs x and y, outputs of x serve as inputs for y, if possible. The combined program then executes from input states of x into output states of y—or not at all. In the tradition of algebra we often write $x \cdot y$ or even xy instead of $x\,;y$. Which programs do we consider equivalent? We postulate that

$$1x = x\,, \qquad x1 = x\,, \qquad 0x = 0\,, \qquad x0 = 0\,, \qquad (xy)z = x(yz)\,.$$

We expect the first two equations or identities because if skip, which does nothing and just lets any value pass, executes before or after any program x, it has the same effect as executing x alone. Abort, by contrast, never executes. The effect of aborting before or after a program execution should therefore be abort. In particular, therefore, if a program aborts at the end of an execution, all information is lost. This explains the third and fourth identity. Finally, we postulate the fifth identity because the effect of executing x, y and z in sequence should not depend on whether we consider xy or yz as composite programs internally—it only depends on the order in which values are passed from x to y and then to z. We may therefore just write xyz, as usual in algebra.

Unlike the multiplication of numbers, sequential compositions of programs need not commute. The programs $x := 5$; $y := x{+}1$ and $y := x{+}1$; $x := 5$, for instance, are not equivalent: the first one terminates in a state where $y = 6$ when executed from an input state where $y = 0$; the second one in a state where $y = 1$.

2.2.2. Nondeterministic Choice. Whenever a conditional **if** p **then** x **else** y executes, the test p determines which of its branches x or y executes. We generalise to programs $x + y$ that choose to execute x or y nondeterministically, so that the effect of $x + y$ can be any output of x or y, depending on this internal choice. We

postulate

$$(x+y)+z = x+(y+z)\,, \qquad x+y = y+x\,, \qquad x+x = x, \qquad x+0 = x\,.$$

First, we expect that the order of binary choices made by the program in the associativity law does not affect the global choice made; the internal choice between z and the result of the choice beween x and y, or between x and the result of the choice between y and z should be the same. Hence once again we may simply write $x+y+z$. Second, $x+y$ and $y+x$ should merely present the choice between x and y differently, reflecting the semantics of "and". Third, $x+x$ should present no choice but x. Finally, as 0 does not allow any execution, we postulate that the effect of choosing $x+0$ should simply be the outputs of x.

2.2.3. Finite Iteration. When program **while** p **do** x executes, it first tests p. If successful, x executes and then the loop executes again from the output states of x. Otherwise the loop skips. We generalise from such loops to the unbounded finite iteration x^* of program x, which chooses nondeterministically to repeat x a finite number of time, including zero times when it just skips. If x^i stands for an i-fold iteration of x, then x^* equals x^i for a nondeterministically chosen $i \in \mathbb{N}$. We could generalise $+$ to an unbounded nondeterministic choice operator to model x^*. Yet we have no further use for it. Instead, using the operations we have used so far, we postulate

$$1 + xx^* = x^* \qquad \text{and} \qquad 1 + x^*x = x^*\,.$$

By these identities, a finite iteration x^* of x chooses to either do nothing or execute x and then continue the iteration (alternatively, to continue the iteration and execute another x afterwards). Yet this alone is not enough: $1 + xy = y$ also holds of an iteration y that chooses to either execute x^* or repeat x infinitely often. Additional laws are therefore needed, but they require some preparation. We leave a more precise description of finite iteration to Section 2.3.4 and 4.1.2.

2.2.4. Distributivity Laws. It remains to consider the effect of the interactions between the operations on programs. We postulate that sequential composition distributes over nondeterministic choice in its first and second argument:

$$x(y+z) = xy + xz \qquad \text{and} \qquad (x+y)z = xz + yz\,.$$

The second law expresses that choosing between x and y and then executing z simply means choosing to execute z after x or after y. By the first law, the time when an internal choice between y and z takes place—before or after the execution of x—should not affect the input/output behaviour of a program.

Ultimately, the validity of these laws of programming depends on the behaviour of real-world programs or at least a more concrete semantics of idealised program executions; a more detailed model of programs as mathematical objects. We introduce such concrete semantics in Chapters 4 and 7. Before that, in the following two sections, we develop an algebra of programs simply by assembling the equations from this section—with some minor additions.

2.3. From Monoids to Kleene Algebras

First we wish to define an algebra $(K, \cdot, +, 0, 1, ^*)$, in which K is the set of all programs, $0, 1 \in K$ are special programs, the operations $\cdot : K \times K \to K$,

$+ : K \times K \to K$ and $(-)^* : K \to K$ model the sequential compositions, nondeterministic choices and finite iterations of programs, and they satisfy the equational laws outlined in the previous section. We build this algebra step-by-step from simpler algebras for sequential compositions, nondeterministic choices and their interactions.

2.3.1. Monoids. We start from algebras with a composition $\cdot$ and an identity 1 that model the sequential composition of programs and the program *skip*.

Definition 2.1. A *monoid* $(M, \cdot, 1)$ consists of a set M, a binary operation $\cdot : M \times M \to M$ and an element $1 \in M$ such that, for all $x, y, z \in M$,

$$x(yz) = (xy)z\,, \qquad 1x = x\,, \qquad x1 = x\,.$$

The monoid M is *commutative* if, for all $x, y \in M$,

$$xy = yx\,.$$

We call M the *carrier set* of the monoid and write M both for this set and as a name of the monoid. Monoids are everywhere in computer science and mathematics.

Example 2.2.

(1) Programs (in the language Imp and beyond) with sequential composition and *skip* form a monoid; they form a commutative monoid with respect to nondeterministic choice and *abort*.
(2) The set X^X of all (endo)functions of type $X \to X$ with function composition $\circ$ and identity function id_X forms a noncommutative monoid.
(3) The set of all $n \times n$ (real or complex) matrices with matrix multiplication and the $n \times n$ identity matrix forms another noncommutative monoid.
(4) Classical examples of commutative monoids are $(\mathbb{N}, +, 0)$, $(\mathbb{N}, \cdot, 1)$ and other kinds of numbers with respect to addition or multiplication. □

2.3.2. Semilattices. Next we consider algebras that model nondeterministic choices of programs and abort.

Definition 2.3. A *semilattice* is a commutative monoid $(S, +, 0)$ in which addition is *idempotent*: for all $x \in S$,

$$x + x = x\,.$$

Semilattices are often defined without 0, but we have no use for such variants.

Example 2.4.

(1) Programs with nondeterministic choice and *abort* form a semilattice.
(2) The power set $\mathcal{P}X$ of a set X, the set of all subsets of X, is a semilattice with respect to $\cup$ and $\emptyset$, and another one with respect to $\cap$ and X. □

Obviously, $X \subseteq Y$ holds in the semilattice $\mathcal{P}X$ if and only if $X \cup Y = Y$ or, equivalently, if and only if $X \cap Y = X$. More generally, we define the relation $\leq$ on the semilattice S, for all $x, y \in S$, as

$$x \leq y \Leftrightarrow x + y = y\,,$$

Because inequalities in semilattices are still based on equations, we refer to them as identities, too.

Recall that a partially ordered set $(P, \leq)$ is a set P with a partial order relation $\leq \subseteq X \times X$, a relation that is reflexive, transitive and antisymmetric.

LEMMA 2.5. *Relation $\leq$ is a partial order on the semilattice S.*

PROOF. We need to check that $\leq$ is reflexive, transitive and antisymmetric; for all $x, y, z \in S$,

$$x \leq x, \qquad x \leq y \wedge y \leq z \Rightarrow x \leq z, \qquad x \leq y \wedge y \leq x \Rightarrow x = y.$$

- Reflexivity: $x \leq x \Leftrightarrow x + x = x$ by idempotency of addition.
- Transitivity: If $x \leq y$ and $y \leq z$, that is, $x + y = y$ and $y + z = z$, then $x + z = x + y + z = y + z = z$ and therefore $x \leq z$.
- Antisymmetry: If $x \leq y$ and $y \leq x$, then $y = x + y = x$ follows from commutativity of $+$. □

REMARK 2.6. Our programming intuition for $x \leq y$ is that program x computes less than program y, that is, the input/output behaviour of x is included in that of y. This can happen because x executes from fewer inputs or is less nondeterministic than y.

REMARK 2.7. We might want to write $x \leq y \Leftrightarrow \exists z.\ x + z = y$ do indicate that some program z needs to be added to program x to obtain program y, but this makes no difference. If $x + y = y$, then obviously there is a z (namely y) such that $x + z = y$. Conversely, if there exists a z such that $x + z = y$, then there exists a z such that $x + y = x + (x + z) = (x + x) + z = x + z = y$. Hence both conditions—$x + y = y$ and $\exists z.\ x + z = y$—define the same partial order.

The partial order on a semilattice has further interesting properties.

LEMMA 2.8. *Let S be a semilattice. Then, for all $x, y, z \in S$,*

(1) $0 \leq x$,
(2) $x \leq y \Rightarrow z + x \leq z + y$,
(3) $x \leq x + y$ *and* $y \leq x + y$,
(4) $x \leq z \wedge y \leq z \Rightarrow x + y \leq z$,
(5) $x + y \leq z \Leftrightarrow x \leq z \wedge y \leq z$.

PROOF. Exercise. □

By (1), 0 is the least element of $\leq$; by (2), addition is *order-preserving*, which is also called *monotone* or *isotone* with respect to $\leq$. By (3), $x + y$ is an *upper bound* of x and y, and in fact the *least upper bound* or *supremum* of x and y by (4). Property (5) combines these two conditions into one.

REMARK 2.9. Our definition of semilattices is *algebraic* because it adds the algebraic operation $+$ to a set and imposes algebraic laws (associativity, commutativity, idempotence, unit laws) on its elements. Alternatively, in light of Lemma 2.5 and 2.8, one can define semilattices *order-theoretically* as partially ordered sets $(S, \leq)$ with least elements and in which each pair of elements has a supremum. This means of course that properties (1) and (5) of Lemma 2.8 are required to hold. One then often writes $\bot$ for 0 and $x \sqcup y$ (*join*) for the binary supremum of $x, y \in S$ and calls $(S, \leq)$ a a *sup-* or *join-semilattice*. It is routine to show that $\sqcup$ is associative, commutative and idempotent, and thus recover the algebraic definition.

Alternatively, one can require that $(S, \leq)$ is a partially ordered set with a greatest element $\top$ in which each pair of element has an infimum, indicated by $\sqcap$ (*meet*). Properties (1) and (5) of Lemma 2.8 are then need to hold with respect to $\geq$ instead of $\leq$. Such semilattices are known as *inf-* or *meet-semilattices*.

2.3.3. Semirings and Dioids. Next we provide an algebra that captures the interaction of sequential compositions and nondeterministic choices of programs.

DEFINITION 2.10. A *semiring* is a structure $(S, \cdot, +, 0, 1)$ such that $(S, \cdot, 1)$ is a monoid, $(S, +, 0)$ a commutative monoid, and, for all $x, y, z \in S$,

$$x(y+z) = xy + xz, \qquad (x+y)z = xz + yz, \qquad 0x = 0, \qquad x0 = 0.$$

A *dioid* is a semiring in which addition is idempotent.

In any dioid S, $(S, +, 1)$ is thus a semilattice and the partial order $\leq$ can be defined. Composition then preserves the order in both arguments.

LEMMA 2.11. *Let S be a dioid. Then, for all $x, y, z \in S$,*

(1) $x \leq y \Rightarrow zx \leq zy$,
(2) $x \leq y \Rightarrow xz \leq yz$.

PROOF. Exercise. □

Dioids therefore have both algebraic and order-theoretic structure.

EXAMPLE 2.12.

(1) Programs with sequential composition, nondeterministic choice, *skip* and *abort* form dioids.
(2) The set of all formal languages over a finite alphabet Σ forms a dioid (see Example 2.19 below for details).
(3) The $n \times n$ real or complex matrices with the obvious operations form non-idempotent non-commutative semirings.
(4) $n \times n$ matrices form dioids if the coefficients range over the booleans $\mathbb{B} = \{0, 1\}$, addition of booleans is max and multiplication is min.
(5) the booleans form themselves a dioid with the operations just mentioned. Mathematicians know this dioid as $\mathbb{Z}/2\mathbb{Z}$.
(6) $(\mathbb{N}, \cdot, +, 0, 1)$ and other algebras of numbers form of course non-idempotent semirings in which multiplication is commutative. □

There are many other computationally interesting examples of semirings and dioids that cannot be explained in these lecture notes.

REMARK 2.13. An important symmetry or duality of semirings and dioids is *opposition*. It swaps the order of composition: $x \cdot^{op} y = y \cdot x$. Intuitively, the opposite of a program executes backwards in time. It is easy to check that the opposite of every dioid axiom is a dioid axiom, so that opposites of dioids are dioids and the opposite of every property of dioids is again a property of dioids. Opposition duality often saves us half of the work in proofs: If we have established a property of a structure, a dual property, in which the order of multiplication is swapped, holds automatically in its opposite whenever the class of structures is closed under opposition. A proof of a dual statement can then be obtained simply by swapping all multiplications in the proof of the original one.

2.3.4. Kleene Algebras. Finally, we include finite iteration.

DEFINITION 2.14. A *Kleene algebra* is a structure $(K, \cdot, +, 0, 1, ^*)$ such that $(K, \cdot, +, 0, 1)$ is a dioid and the Kleene star operation $(-)^* : K \to K$ satisfies, for

all $x, y, z \in K$,

$$1 + xx^* \leq x^*\,, \qquad z + xy \leq y \Rightarrow x^*z \leq y\,,$$
$$1 + x^*x \leq x^*\,, \qquad z + yx \leq y \Rightarrow zx^* \leq y\,.$$

The axioms in the second line are the opposites of those in the first one. The axiom $1+xx^* \leq x^*$ and its opposite are called *unfold* axioms; $z+xy \leq y \Rightarrow x^*z \leq y$ and its opposite are called *induction* axioms.

Lemma 2.16 below shows that the left unfold axiom $1 + xx^* \leq x^*$ can be strengthened to $1+xx^* = x^*$ and the left induction axiom to $z+xy = y \Rightarrow x^*z \leq y$.

On one hand, $1+xx^* = x^*$ tells us that iterating x means either doing nothing or executing x and then continuing to iterate, as explained in Section 2.2.

On the other hand, x^* is the fixpoint of the function $\lambda y.\ 1 + xy$, where λy indicates that this function varies in y. Moreover the left induction axiom becomes $1 + xy = y \Rightarrow x^* \leq y$ for $z = 1$. This instance axiomatises x^* as the *least* fixpoint of $\lambda y.\ 1 + xy$. This excludes any proper infinite iteration of x.

The induction axioms of Kleene algebra are more general than that for reasons that are rather subtle. A practical explanation is that this generality is needed for proofs about programs; a more detailed mathematical one can be found in Remarks 4.7 and 4.8 in Section 4.1.2 below.

Remark 2.15. We use λ-notation throughout these lecture notes. It comes from the λ-calculus, which provides a foundation of functional programming, type theory and program semantics. We cannot explain it in detail. It suffices to know that we can write $\lambda x.\ x + y$ in place of $x \mapsto x + y$ for an "anonymous" function, which might otherwise be called $add_y\, x = x+y$. We can then apply this function to an argument, 5 say, so that $(\lambda x.\ x+y)\,5 = 5+y$, in the same way $add_y\, 5$ evaluates to $5 + y$. This function application is called *β-reduction*. We may equally see it as a substitution, $(\lambda x.\ x+y)\,5 = (x+y)[5/x]$, where the right-hand side indicates the result of substituting 5 for x in $x + y$.

The following facts are useful for reasoning about the star.

Lemma 2.16. *Let K be a Kleene algebra. Then, for all $x, y, z \in K$,*

(1) $1 \leq x^*$,
(2) $xx^* \leq x^*$ *and* $x^*x \leq x^*$,
(3) $x^i \leq x^*$ *for all* $i \in \mathbb{N}$, $x^0 = 1$ *and* $x^{i+1} = xx^i$,
(4) $x^*x^* = x^*$,
(5) $x^{**} = x^*$,
(6) $1 + xx^* = x^*$ *and* $1 + x^*x = x^*$,
(7) $x \leq y \Rightarrow x^* \leq y^*$,
(8) $(xy)^*x = x(yx)^*$,
(9) $(x + y)^* = x^*(yx^*)^*$,
(10) $(x + y)^* = (x^*y^*)^*$,
(11) $x \leq 1 \Leftrightarrow x^* = 1$,
(12) $zx \leq yz \Rightarrow zx^* \leq y^*z$ *and* $xz \leq zy \Rightarrow x^*z \leq zy^*$,
(13) $xy \leq y \Rightarrow x^*y \leq y$ *and* $yx \leq y \Rightarrow yx^* \leq y$.

Proof. We prove (4) to illustrate the typical style of reasoning with stars in Kleene algebra. The other properties are left as exercises.

First, we prove $x^*x^* \leq x^*$. By left star induction, it suffices to show that $x^* + xx^* \leq x^*$ and therefore $x^* \leq x^*$ and $xx^* \leq x^*$ by properties of suprema. The first inequality is trivial, the second follows from (2).

Next we prove $x^* \leq x^*x^*$. We calculate $x^* = x^*1 \leq x^*x^*$, using (1) and order-preservation of $\cdot$ in the second step.

Claim (4) then follows from antisymmetry of $\leq$. □

REMARK 2.17. Using antisymmetry of $\leq$ to prove $x \leq y$ and $y \leq x$ separately in order to establish $x = y$ is generally a good proof strategy for Kleene stars.

Properties (1), (2) and (3) say that doing nothing (1), unfolding xx^* and all finite iterations x^i are part of an iteration x^*. Properties (4) and (5) say that iterating x twice or iterating the iteration of x are as good as iterating x once. Property (6) says that x^* is a fixpoint of $\lambda y.\ 1 + xy$ and $\lambda y.\ 1 + yx$, as already mentioned. By property (7), the star is order-preserving. Property (8) says that iterating xy and then executing another x is the same as executing x and then iterating yx. Property (9) reduces the iteration of choices between x and y to sequences of iterations of x and y. By commutativity of addition, we immediately get that $(x+y)^* = y^*(xy^*)^*$. By property (11), iterating an element $x \leq 1$ amounts to doing nothing. The properties in (11) are called *simulation laws*. We can read $zx \leq yz$ as saying that whenever we can execute z and then x, then we can execute y and then z, so we can imagine that z allows us to relate or simulate any execution of x by an execution of y. The first simulation law then says that this relationship extends to iterations of x and iterations of y. The second simulation law is obtained by opposition. Finally, (13) presents two alternative simpler star induction laws. They are equivalent to those in Definition 2.14.

REMARK 2.18. Properties (3), in the special case $x \leq x^*$, (5) and (7) are interesting in their own right. A function $f : P \to P$ on a partially ordered set $(P, \leq)$ that is *extensive* (or *inflationary*), *transitive* and *order preserving*,

$$x \leq f\,x, \qquad f\,(f\,x) \leq f\,x \qquad \text{and} \qquad x \leq y \Rightarrow f\,x \leq f\,y,$$

is a *closure operator*. Hence $(-)^*$ is a closure operator.

EXAMPLE 2.19. Two models of Kleene algebra that serve as program semantics, binary relations and non-deterministic functions (state transformers), are discussed in detail in Chapter 4. Here we only mention another model relevant to computer science, which comes from language theory.

It is no coincidence that $+$, $\cdot$ and * are precisely the operations on regular expressions over a finite alphabet Σ. In formal language theory, regular expressions are interpreted as regular languages, as sets of finite words whose letters are elements of Σ. For languages X and Y one defines $X + Y$ as the union of X and Y, the composition $X \cdot Y$ as the concatenation of all words in X with all words in Y, the language product

$$X \cdot Y = \{x \cdot y \mid x \in X, y \in X\},$$

and X^* as the union of all i-fold compositions of language X with itself, including the empty-word language $\{\varepsilon\}$, the Kleene star

$$X^* = \bigcup X^i, \qquad \text{for } X^0 = \{\varepsilon\} \text{ and } X^{n+1} = X \cdot X^n.$$

The set of all languages over Σ then forms a Kleene algebra under these operations and with $\emptyset$ as 0 and $\{\varepsilon\}$ as 1.

In particular, the set of all regular languages generated by Σ forms a sub-Kleene algebra of the language Kleene algebra. These form the smallest set of languages that contains $\emptyset$, $\{\varepsilon\}$ and all singleton languages $\{a\}$ with $a \in \Sigma$, and that is closed with respect to union, language product and the Kleene star. □

EXAMPLE 2.20. More generally, for any monoid $(M, \cdot, 1)$ we obtain a Kleene algebra on $\mathcal{P}M$ as follows. For $X, Y \subseteq M$, define the composition or *complex product* $\cdot : \mathcal{P}M \times \mathcal{P}M \to \mathcal{P}M$ as $X \cdot Y = \{x \cdot y \mid x \in X, y \in Y\}$. Define addition as $\cup$, the unit as $\{1\}$, the zero as $\emptyset$ and X^* as the union of all i-fold compositions of language X with itself, including $\{1\}$, as for languages above. It is then straightforward to check the Kleene algebra axioms. □

REMARK 2.21. Historically, Kleene algebra did not originate in program verification. Instead, such algebras were meant to axiomatise the equivalence of regular expressions (two regular expressions are deemed equivalent if they are interpreted as the same regular language). The relationship to language theory has interesting consequences for program analysis with Kleene algebra. Regular expression equivalence is decidable using tools and techniques from automata theory. It can be shown that it is therefore decidable whether an identity in the language of Kleene algebra holds, too. Consequently, restricted forms of program equivalence are also decidable. Yet we do not make use of this in these lecture notes and refer to the literature for further information.

2.4. Kleene Algebra with Tests

We now add tests to Kleene algebra to capture conditionals and while-loops more faithfully. We model them as elements of a boolean algebra, so that we can express their conjunction, disjunction and negation. As explained in Section 2.1, we view tests as special programs that do not alter the program state. But before introducing Kleene algebras with tests, we recall the basics of boolean algebras.

2.4.1. Boolean Algebras. We base the definition of boolean algebras on lattices and distributive lattices, because we need these structures in Chapter 9.

DEFINITION 2.22.

(1) A *lattice* is a structure $(L, \sqcap, \sqcup, \bot, \top)$ such that $(L, \sqcap, \top)$ and $(L, \sqcup, \bot)$ are semilattices and the following *absorption laws* hold for all $x, y \in L$:

$$x \sqcup (x \sqcap y) = x \qquad \text{and} \qquad x \sqcap (x \sqcup y) = x.$$

(2) A lattice L is *distributive* if one of the following *distributivity laws* holds (and therefore both):

$$x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z), \qquad x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z).$$

(3) A *boolean algebra* is a structure $(B, \sqcap, \sqcup, \overline{\ }, \bot, \top)$ such that $(B, \sqcap, \sqcup, \bot, \top)$ is a distributive lattice, $\overline{(-)} : B \to B$ is a unary operation of *complementation* and, for all $x, y, z \in B$,

$$x \sqcup \overline{x} = \top \qquad \text{and} \qquad x \sqcap \overline{x} = \bot.$$

As for semilattices, lattices and distributive lattices are usually defined without $\bot$ and $\top$ (lattices which have these least and greatest elements are then called *bounded*). Yet we have need no need for lattices without $\bot$ and $\top$.

First we list some useful equational properties.

LEMMA 2.23. *Let B be a boolean algebra. Then, for all $x, y, z \in B$,*

(1) $\overline{\overline{x}} = x$,
(2) $x \sqcap \bot = \bot$ *and* $x \sqcup \top = \top$,
(3) $\overline{x \sqcup y} = \overline{x} \sqcap \overline{y}$ *and* $\overline{x \sqcap y} = \overline{x} \sqcup \overline{y}$.

PROOF. Exercise. □

The properties in (3) are known as *De Morgan laws*.

Boolean algebras are ordered by the partial orders on their underlying semilattices, that is,

$$x \leq y \Leftrightarrow x \sqcup y = y \Leftrightarrow x \sqcap y = x\,.$$

Thus in particular Lemma 2.5 and 2.8 apply. Boolean algebras show a form of duality as well. It is given by swapping, $\bot$ with $\top$, joins with meets and reversing the partial order. As for semirings and Kleene algebras, the dual of each boolean algebra axioms is another boolean algebra axiom. The class of boolean algebras is therefore closed with respect to duality, and for each property of boolean algebra a dual property holds automatically.

This duality is visible in the following lemma, which rephrases the properties for join of Lemma 2.8 in the context of boolean algebras, and adds the dual properties for meet—with one exception.

LEMMA 2.24. *Let B be a boolean algebra. Then, for all $x, y, z \in B$,*

(1) $\bot \leq x$ *and* $x \leq \top$,
(2) $x \leq y \Rightarrow z \sqcup x \leq z \sqcup y$ *and* $x \leq y \Rightarrow z \sqcap x \leq z \sqcap y$
(3) $x \leq x \sqcup y$ *and* $y \leq x \sqcup y$,
(4) $x \leq z \wedge y \leq z \Rightarrow x \sqcup y \leq z$,
(5) $x \sqcup y \leq z \Leftrightarrow x \leq z \wedge y \leq z$,
(6) $x \sqcap y \leq x$ *and* $x \sqcap y \leq y$,
(7) $z \leq x \wedge z \leq y \Rightarrow z \leq x \sqcap y$,
(8) $z \leq x \sqcap y \Leftrightarrow z \leq y \wedge z \leq y$,
(9) $x \leq y \Leftrightarrow \overline{y} \leq \overline{x}$,
(10) $x \sqcap \overline{y} \leq z \Leftrightarrow x \leq y \sqcup z$ *and* $x \leq \overline{y} \sqcup z \Leftrightarrow x \sqcap y \leq z$.

PROOF. Exercise. □

The exception is of course property (9), which shows that complementation is *antitone* with respect to the order or *order reversing*. Properties (3)-(5) state once again that $\sqcup$ is a binary supremum operation. Dually, by properties (6)-(8), $\sqcap$ is a binary *infimum* or *greatest lower bound* operation.

Properties (1)-(8) hold already in lattices, distributivity or complementation properties are not needed in their proofs. Order-theoretically, lattices can therefore be defined as partially ordered sets that are both sup- and inf-semilattices, that is, they have a least and and greatest element and all binary suprema and infima exist. The lattice axioms, notably the two absorption laws are then derivable.

Here are two boolean algebras we care about.

EXAMPLE 2.25.

(1) The powerset $\mathcal{P}X$ of set X forms a boolean algebra with $\cup$ as join, $\cap$ as meet, $\emptyset$ as the least element, X as the greatest element and set complementation $-$ as complementation.
(2) The booleans $\mathbb{B}$ form a boolean algebra with max as join, min as meet, 0 as least element, 1 as greatest element and $\lambda x.\ 1 - x$ as complementation.

Next we present a distributive lattice that is of interest to us.

EXAMPLE 2.26. An ordered set $(P, \leq)$ is *totally ordered* if any pair of elements is comparable by $\leq$, that is, $x \leq y$ or $y \leq x$ holds for all $x, y \in P$. Totally ordered sets (or totally ordered subsets of partially ordered sets) are also known as *chains*. Every finite chain is a distributive lattice with respect to suprema and infima. □

EXAMPLE 2.27. Every distributive lattice is a dioid with maximal element $\top = 1$. Thus, in particular, every finite chain is such a dioid. □

REMARK 2.28. Finally, another way of looking at boolean algebras is as follows. A pair of elements x, y in a lattice are *complements* if $x \sqcup y = \top$ and $x \sqcap y = \bot$. More specifically, we then call each of these elements a *complement* of the other. In a distributive lattice, each element can have at most one complement. For a proof, first note that, in any distributive lattice, $x \sqcap y = x \sqcap z$ and $x \sqcup y = x \sqcup z$ implies $y = z$:

$$\begin{aligned} y &= y \sqcap (x \sqcup y) \\ &= y \sqcup (x \sqcup z) \\ &= (y \sqcup x) \sqcap (y \sqcap z) \\ &= (x \sqcup z) \sqcap (y \sqcap z) \\ &= (x \sqcup y) \sqcap z \\ &= (x \sqcup z) \sqcap z \\ &= z. \end{aligned}$$

Now suppose element x has complements y and y'. Then $x \sqcup y = 1 = x \sqcup y'$ and $x \sqcap y = 0 = x \sqcap y'$ and therefore $y = y'$.

A lattice is *complemented* if each element has a complement. Boolean algebras and complemented distributive lattices are therefore the same structures.

2.4.2. Kleene Algebras with Tests.

DEFINITION 2.29. A *Kleene algebra with tests* is a structure $(K, B, \cdot, +, \overline{\ }, 0, 1, ^*)$ such that $(K, +, \cdot, 0, 1, ^*)$ is a Kleene algebra and $(B, \cdot, +, \overline{\ }, 0, 1)$ a boolean algebra that is a subalgebra of K.

In this definition, the operation $\overline{(-)}$ of complementation is only defined on B. The operations $\cdot$ and $+$ have two purposes: they model sequential composition and nondeterministic choice of general programs in K, and at the same time meet and join in of tests in B. As the boolean algebra of tests B is a subalgebra of K, the operations $+$ and $\cdot$ must be closed with respect to B: adding and multiplying two elements of B yields another element of B. Similarly, 0 and 1 model abort and skip in K and at the same time theleast and greatest element—true and false—in B.

We henceforth write $p, q, r, \ldots$ for tests in B and continue writing $x, y, z, \ldots$ for arbitrary programs in K. We also write KAT for Kleene algebra with tests.

REMARK 2.30. Alternatively, we can define a KAT as a structure (K, B, ι), where $\iota : B \to K$ is an *order-embedding* (it satisfies $\iota p \leq \iota q \Rightarrow p \leq q$) that preserves the boolean operations:

$$\iota(p \sqcup q) = \iota p + \iota q, \qquad \iota(p \sqcap q) = \iota p \cdot \iota q, \qquad \iota 0 = 0, \qquad \iota 1 = 1.$$

It then follows that $p \leq q \Leftrightarrow \iota p \leq \iota q$ and further that ι is injective. The image $\iota(B)$ of B under ι is thus a copy of B in K and ι is an *order isomorphism* between B and $\iota(B)$. The order structure of B and $\iota(B)$ is therefore the same, $\iota 0$ is the least and $\iota 1$ the greatest element in $\iota(B)$. By preservation of the boolean operations, sums and products of elements in $\iota(B)$ are again in $\iota(B)$. This shows that $\iota(B)$ is indeed a boolean subalgebra of K.

KAT models tests or assertions as programs that allow some states as inputs and have the same states as output, like in our intuitive semantics in Section 2.1. As tests are elements below 1, they are *partial identity* or *subidentity* elements that may not be executable from some states, but execute like 1 whenever they are. We henceforth freely identify tests with predicates and with the sets of states on which they hold without making a clear distinction.

It then follows that program px corresponds to the restriction of program x to input states where predicate p holds, whereas, by opposition, xp corresponds to the restriction of x to output states where predicate q holds. In KAT, of course, $px \leq x$ and $xq \leq x$ hold simply because $p, q \leq 1$ and multiplication is order preserving.

The following facts describe the interaction between tests and programs. They are useful for program verification in Chapter 6 and 8.

LEMMA 2.31. *Let K be a* KAT*. For all $x \in K$ and $p, q \in B$, the following identities are equivalent,*

(1) $px \leq xq$,
(2) $px\overline{q} = 0$,
(3) $pxq = px$.

By opposition, the following identities are equivalent,

(4) $xq \leq px$,
(5) $\overline{p}xq = 0$,
(6) $pxq = xq$.

Finally,

(7) $px \leq xq \Leftrightarrow x\overline{q} \leq \overline{p}x$.

PROOF. Exercise. □

PROPOSITION 2.32. *Every Kleene algebra can be extended to a Kleene algebra with tests.*

PROOF. In any Kleene algebra K we define $B = \{0, 1\} \subseteq K$ and complementationby $\overline{0} = 1$ and $\overline{1} = 0$. Checking that B forms a boolean subalgebra of K with $+$ as join, $\cdot$ as meet is then routine. □

REMARK 2.33. Could we not simply require that all subidentities form tests, that is, that $B = \{x \mid x \leq 1\}$? Though this would work for the semantics of programs we have in mind, other models of interest would be excluded.

(1) The so-called min-*plus semiring* or min-*tropical semiring* is a dioid that appears in combinatorial optimisation, the theory of algorithms or speech recognition. Its carrier set is $\mathbb{R}_{\geq 0} \cup \{\infty\}$, addition is min, multiplication is $+$, ∞ is 0 and 0 is 1. It becomes a Kleene algebra with a star assigning $0 \in \mathbb{R}_{\geq 0}$ to every element. The order of this min-plus Kleene algebra is $\geq_{\mathbb{R}}$. All elements are therefore subidentities, but they do not form a boolean algebra, simply because multiplication $+$ is not idempotent. Nevertheless $\{\infty, 1\}$ is a suitable boolean subalgebra of tests by Proposition 2.32.
(2) The chain $0 < a < 1$ on the set $K = \{0, a, 1\}$ is a distributive lattice according to Example 2.26 and a dioid with join as $+$ and meet as $\cdot$ according to Example 2.27. It forms a Kleene algebra with $x^* = 1$ for all $x \in K$ (there is no other choice for this map). Again, $\{x \in K \mid x \leq 1\} = K$ is not a boolean algebra (a is not complemented), but $B = \{0, 1\}$ yields a boolean subalgebra of tests by Proposition 2.32.

2.4.3. Algebraic Semantics of while-Programs. We can now define an algebraic semantics of conditionals and while-loops in KAT:

$$\textbf{if } p \textbf{ then } x \textbf{ else } y = px + \overline{p}y\,,$$
$$\textbf{while } p \textbf{ do } x = (px)^*\overline{p}\,.$$

That of conditionals is straightforward: from states where test p evaluates to true, program x executes, and otherwise, from states where p evaluates to false, program y. The semantics of the while-loop is slightly more complicated: so long as test p evaluates to true, program x is executed; after the loop terminates, the program is in a state where p evaluates to false.

As a sanity check, we show that our semantics of while-loops satisfies a least fixpoint property, that models the unfolding of a loop: If test p is true, then the body x of the loop executed and then the loop **while** p **do** x is executed again. Otherwise, if p is false, the loop skips.

LEMMA 2.34. *Let K be a* KAT*. Then, for all $x \in K$ and $p \in B$, the element* **while** p **do** x *is the least fixpoint of the function*

$$\varphi\, y = \textbf{if } p \textbf{ then } xy \textbf{ else } \mathsf{skip}\,.$$

PROOF. First we show that **while** p **do** x is a fixpoint of φ, that is,

$$\textbf{while } p \textbf{ do } x = \textbf{if } p \textbf{ then } (x \cdot \textbf{while } p \textbf{ do } x) \textbf{ else } \mathsf{skip}\,.$$

Folding and unfolding definitions and using laws of Kleene algebra,

$$\begin{aligned}\textbf{if } p \textbf{ then } (x \cdot \textbf{while } p \textbf{ do } x) \textbf{ else } \mathsf{skip} &= px(px)^*\overline{p} + \overline{p}1\\ &= (px(px)^* + 1)\overline{p}\\ &= (px)^*\overline{p}\\ &= \textbf{while } p \textbf{ do } x\,.\end{aligned}$$

The third step, in particular, uses the left star unfold law in equational form.

Next we show that **while** p **do** x is the least fixpoint of φ. Suppose y is another fixpoint of φ, that is, $\varphi\, y = y$ and therefore $pxy + \overline{p} = y$. Then $(px)^*\overline{p} \leq y$ by left star induction and therefore **while** p **do** $x \leq y$, as required. □

The programming intuitions about conditionals and loops given are shown to be consistent with concrete semantics of program executions in Chapter 4 and 7.

REMARK 2.35. It seems tempting to model an infinite loop of x as

$$\text{infinite loop of } x = \textbf{while } 1 \textbf{ do } x\,.$$

Yet expanding the loop semantics in KAT shows that $\textbf{while } 1 \textbf{ do } x = (1x)^*0 = 0$, so that this infinite loop is actually equal to *abort*. This should not surprise us. Our failed attempt is based on the Kleene star, which models finite iteration. In fact, for the result y of an infinite iteration, even the right annihilation equation $y0 = 0$ should raise questions—how could we possibly execute another element after y? We should rather expect that $y0 = y$, and more generally $yx = y$ holds for a non-terminating element y, if we are prepared to allow such sequential compositions at all. KAT therefore yields an algebraic semantics for *partial program correctness*, where all loops are assumed to terminate, but not for *total correctness*, where loops may run forever. We exploit this further in Chapter 6 and 7

2.5. Examples: Program Transformations

The verification of the fixpoint equation for the while loop in KAT in the proof of Lemma 2.34 is an example of a *program transformation*. Applied from right to left it transforms a more complex program with a nested conditional and loop into an equivalent simple while-loop—without changing the input/output behaviour of the program. Such transformations are performed, for instance, when compilers optimise programs. One reason for the widespread attention of KAT is that it supports the verification of some non-trivial program transformations and compiler optimisations by equational reasoning. We present two further examples.

EXAMPLE 2.36. We wish to verify that

$$\textbf{if } p \textbf{ then } xy \textbf{ else } xz \;=\; x \cdot \textbf{if } p \textbf{ then } y \textbf{ else } z$$

holds whenever $px = xp$, which means that program x does not affect the outcome of test p. This commutativity condition is obviously useful for pushing p into the xy-branch of the conditional, but proceeding likewise in its xz-branch would require $\overline{p}x = x\overline{p}$. Fortunately, this identity is derivable in KAT by Lemma 2.31:

$$\begin{aligned} px = xp &\Leftrightarrow px\overline{p} + \overline{p}xp = 0 \\ &\Leftrightarrow \overline{\overline{p}}x\overline{p} + \overline{p}x\overline{\overline{p}} = 0 \\ &\Leftrightarrow \overline{p}x = x\overline{p}\,. \end{aligned}$$

We can then finish the proof by calculating

$$\begin{aligned} \textbf{if } p \textbf{ then } xy \textbf{ else } xz &= pxy + \overline{p}xz \\ &= xpy + x\overline{p}z \\ &= x(py + \overline{p}z) \\ &= x \cdot \textbf{if } p \textbf{ then } y \textbf{ else } z\,. \end{aligned}$$

□

The next example shows that one can denest while-loops with KAT, which is generally important for program optimisation.

EXAMPLE 2.37. We wish to verify that

$$\begin{aligned} &\textbf{while } p \textbf{ do } (x \cdot \textbf{while } q \textbf{ do } y) = \\ &\qquad \textbf{if } p \textbf{ then } (x \cdot \textbf{while } (p+q) \textbf{ do } (\textbf{if } q \textbf{ then } y \textbf{ else } x)) \textbf{ else } \mathsf{skip}\,. \end{aligned}$$

First we simplify the while loop in the second program using KAT:

$$\begin{aligned}\textbf{while } (p+q) \textbf{ do } (\textbf{if } q \textbf{ then } y \textbf{ else } x) &= ((p+q)(qy+\overline{q}x))^* \,\overline{qp}\\ &= (\overline{q}px+qy)^* \,\overline{qp}\\ &= (qy)^* \,(\overline{q}px(qy)^*)^* \,\overline{qp}\\ &= (qy)^*\overline{q}\,(px(qy)^*\overline{q})^* \,\overline{p}\,.\end{aligned}$$

The proof uses Lemma 2.16 (9) and then (8) in the last two steps and mainly boolean algebra in the other ones. Rewriting the second program using this result and unfolding the definition of conditional yields $p(qy)^*\overline{q}\,(px(qy)^*\overline{q})^* \,\overline{p} + \overline{p}$. The equivalence proof, from right to left, is then straightforward:

$$\begin{aligned}\textbf{if } p \textbf{ then } (x \cdot \textbf{while } (p+q) \textbf{ do } (\textbf{if } q \textbf{ then } y \textbf{ else } x)) \textbf{ else } \mathsf{skip}&\\ = px(qy)^*\overline{q}\,(px(qy)^*\overline{q})^* \,\overline{p} + \overline{p}&\\ = \left(1 + px(qy)^*\overline{q}\,(px(qy)^*\overline{q})^*\right)\overline{p}&\\ = (px(qy)^*\overline{q})^* \,\overline{p}&\\ = (px \cdot \textbf{while } q \textbf{ do } y)^* \,\overline{p}&\\ = \textbf{while } p \textbf{ do } (x \cdot \textbf{while } q \textbf{ do } y)\,.&\end{aligned}$$

The star unfold axiom of Kleene algebra is used in the third step. □

The literature contains many other examples of verifications of program equivalences and transformations based on KAT. Such tasks were traditionally considered tedious; proofs in concrete program semantics could fill many pages. The first breakthrough, about 30 years ago, was the realisation that such proofs could be performed much more concisely by equational reasoning in algebra. KAT, in particular, was conceived about 25 years ago. The second breakthrough, about 15 years, was the insight that such calculations could even be performed by machine, and often fully automatically.

CHAPTER 3

Formalising the Algebra of Programs

We formalise the hierarchy of algebras from Section 2.3 and 2.4 with the Isabelle/HOL proof assistant. This is a first step towards building program verification components with Isabelle. We also show how the program transformation examples in Section 2.5 can be formalised with KAT and Isabelle. This section can be read as an brief introduction to formalising mathematics, in particular algebra, with Isabelle. Yet it is not fully self-contained and requires reading the Isabelle documentation in parallel. The entire material in this section, and many additional definitions and proofs, can be found in the Isabelle theories for this course at my git repository

https://github.com/gstruth/verisa

3.1. Engineering Algebraic Hierarchies with Isabelle

Isabelle offers two mechanisms for engineering mathematical hierarchies—type classes and locales—with specific tutorials for both:

https://isabelle.in.tum.de/dist/Isabelle2021-1/doc/classes.pdf

https://isabelle.in.tum.de/dist/Isabelle2021-1/doc/locales.pdf

Type classes are available in many proof assistants. They are inspired by those in functional programming languages like Haskell. A type class is roughly a collection of types that are parametrically polymorphic. The types in a type class thus share properties that are independent of their particular type, such as a set of axioms, which hold in many models or situations. Operationally, type classes allow us to axiomatise algebraic structures, extend and combine them, and instantiate them to models using the polymorphism mentioned. They also provide contexts for proving facts that hold in a particular class. All this will become clear by example.

Locales offer similar features, but outside of Isabelle's type system within set theory. The concrete relationships and differences between classes and locales and the consequences of choosing one or the other are quite subtle and not very well documented. We generally prefer type classes unless an algebra requires several type parameters—simply because type classes do not support that. In fact, our entire mathematical development can be restricted to this setting, and we show only one formalisation of a locale for KAT as an example. Nevertheless we use tools from Isabelle's locale package for relating type classes and constructing models for them. We explain these features as we go along.

3.1.1. Monoids. We start with formalising monoids as type classes in Isabelle, translating the definitions of Section 2.3.1. Isabelle's main libraries contain type classes for monoids already; we switch to them when we build more complex algebras. We begin with our own classes simply to explain the typical set-up.

But before that we take a quick look at the header of our Isabelle theory file *KA.thy*. It starts by declaring the name *KA* of the Isabelle theory, which must be identical with the file name, and by listing the Isabelle libraries imported. We start from scratch and import only Isabelle's main libraries. Then we open the theory context with the keyword **begin**.

theory *KA*
imports *Main*

begin

Isabelle requires formalising multiplicative and additive monoids separately.

notation *times* (**infixl** · *70*)

class *mult-monoid* = *times* + *one* +
assumes *mult-assoc*: $x \cdot (y \cdot z) = (x \cdot y) \cdot z$
and *mult-unitl*: $1 \cdot x = x$
and *mult-unitr*: $x \cdot 1 = x$

class *add-monoid* = *plus* + *zero* +
assumes *add-assoc*: $x + (y + z) = (x + y) + z$
and *add-unitl*: $0 + x = x$
and *add-unitr*: $x + 0 = x$

class *abelian-add-monoid* = *add-monoid* +
assumes *add-comm*: $x + y = y + x$

Type class *mult-monoid* extends the type classes *times* and *one* from Isabelle's main libraries. This extension is indicated by writing +. The classes *times* and *one* simply introduce notation for the binary operation $*$ and the constant or nullary operation 1. We have changed Isabelle's notation for composition in class *times* from $*$ to $\cdot$ using the **notation** command. Hovering over the class name in Isabelle's editor *jEdit* and then clicking leads to the class definitions in Isabelle's main libraries. This shows that the identity and composition operations are parametrised by type *'a*, usually written α in text, and therefore polymorphic, so that their type can be instantiated.

Class *mult-monoids* also declares the axioms of multiplicative monoids. Each one carries a name so that users can tell Isabelle to use them as hypotheses in proofs. In the mathematical chapters of these lecture notes, we have given such names only sporadically. Instead we often show detailed textbook style Isabelle proofs in our theory files that expose the Isabelle names of the hypotheses used (and the hypotheses themselves by hovering over the names and then clicking in jEdit). In the theory files, the three monoid axioms are put in quotes. These are usually not displayed in the Isabelle documentation. We follow this tradition.

Class *add-monoid* is built similarly from the classes *plus* and *zero*. Additive monoids are extended to abelian additive monoids by adding the commutativity axiom.

By contrast to Definition 2.1, we do not specify the carrier sets of monoids explicitly. Instead, Isabelle assigns an implicit carrier set *UNIV*. Working without carrier sets has limitations when formalising advanced mathematical concepts, but is sufficient and more convenient for us as it leads to simpler classes and proofs.

3.1.2. Semilattices. It is straightforward to extend the class for commutative additive monoids to the class *sup-semilattice* for semilattices. Now we start using Isabelle's built-in type class *comm-monoid-add* instead of *add-monoid* to have all facts about commutative monoids from Isabelle's main libraries in scope.

class *sup-semilattice* = *comm-monoid-add* + *ord* +
  assumes *add-idem*: $x + x = x$
  and *order-def*: $x \leq y \longleftrightarrow x + y = y$
  and *strict-order-def*: $x < y \longleftrightarrow x \leq y \wedge x \neq y$

begin

Writing **begin** after the type class declaration opens a context in which facts about semilattices can be proved, including those from Section 2.3.2. First we show that semilattices form partial orders (Lemma 2.5).

Formally, with Isabelle, we show that the class *sup-semilattice* of semilattices forms a subclass of Isabelle's class *order* for partial orders.

subclass *order*
proof *unfold-locales*
  fix x y z
  show $(x < y) = (x \leq y \wedge \neg\, y \leq x)$
    using *add-commute order-def strict-order-def* **by** *auto*
  show $x \leq x$
    by (*simp add*: *add-idem order-def*)
  show $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$
    by (*metis add-assoc order-def*)
  show $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
    by (*simp add*: *add-commute order-def*)
qed

Declaring **subclass** *order* in the context of class *sup-semilattice* makes Isabelle generate the proof obligations for deriving the partial order axioms from the semilattice axioms, unfolding the definitions of $\leq$ and $<$ automatically. Typing **proof** *unfold-locales* opens a proof context in which these proof obligations are accessible. We can copy them from the proof window in jEdit into the proof context shown above. The declaration **fix** x y z translates the universal quantifiers, which are implicit in the equational laws, into parameters. The proof context is closed by typing **qed**. After each line starting with **show**, a proof is required (in the first line, Isabelle uses = instead of $\longleftrightarrow$, and we use this notation henceforth). We can search for a proof by invoking Isabelle's Sledgehammer tool from jEdit. When proofs are simple, as in this case, Sledgehammer finds them quickly and fully automatically. It can call other methods like *auto*, *simp* or *metis*, which are part of

Isabelle's theorem proving infrastructure. Isabelle lists the (main) facts it uses in proofs explicitly. Axiom *add-commute* from type class *comm-monoid-add* is used, for instance, to discharge the first proof obligation.

After this **subclass** proof, everything Isabelle knows about partial orders—all theorems proved within the context of class *order*—become available in the context of class *sup-semilattice*. We can therefore use them for proving additional facts about semilattices. Here are some of the proofs for Lemma 2.8 as examples.

lemma *zero-least*: $0 \leq x$
proof−
have $0 + x = x$
by *simp*
thus $0 \leq x$
by (*simp add*: *order-def*)
qed

lemma *add-isor*: $x \leq y \Longrightarrow x + z \leq y + z$
proof−
assume $x \leq y$
hence *a*: $x + y = y$
by (*simp add*: *order-def*)
have $x + z + y + z = x + y + z$
by (*metis add-commute local.add-assoc local.add-idem*)
also have $\ldots = y + z$
by (*simp add*: *a*)
finally have $x + z + y + z = y + z$**.**
thus $x + z \leq y + z$
by (*simp add*: *add-assoc order-def*)
qed

They illustrate the basic features of Isabelle's proof scripting language *Isar*, where we type textbook-style proofs step-by-step and use Sledgehammer to discharge the resulting proof obligations. We can decompose an implication by isolating its assumption. The keyword **hence** indicates that the fact which follows it can be derived by the line immediately above (**have** can be used otherwise).

lemma *add-ubl*: $x \leq x + y$
proof−
have $x + y = x + x + y$
by (*simp add*: *add-idem*)
thus *?thesis*
by (*metis add-assoc order-def*)

lemma *add-ubr*: $y \leq x + y$
using *add-commute add-ubl* **by** *fastforce*

The proofs of *add-ubl* and *add-ubr* are very similar. In that of *add-ubl*, *?thesis* abbreviates the proof goal. Instead of performing that of *add-ubr* step by step with Isar, we can call Sledgehammer directly and obtain a one-line automatic proof.

Beyond Sledgehammer, we can also use proof tools and simplifiers like *auto*, *simp*, *fastforce* or *blast* directly and in particular by listing the relevant assumptions explicitly. Details are described in Isabelle's documentation.

https://isabelle.in.tum.de/dist/Isabelle2021-1/doc/sledgehammer.pdf

The *metis* tool, however, is typically used by Sledgehammer. Sledgehammer calls untrusted external automated theorem provers. Metis is an internally verified automated theorem prover that reconstructs their proofs to make them acceptable for Isabelle.

3.1.3. Semirings and Dioids. Building type classes for semirings and dioids (Section 2.3.3) is now straightforward.

```
class semiring = comm-monoid-add + monoid-mult +
  assumes distl: x · (y + z) = x · y + x · z
  and distr: (x + y) · z = x · z + y · z
  and annil [simp]: 0 · x = 0
  and annir [simp]: x · 0 = 0

class dioid = semiring + sup-semilattice
```

Formalising Lemma 2.11 offers no new insight and can be left as an exercise.

3.1.4. Kleene Algebras. The type class for Kleene algebras extends that for dioids as expected. We star with a class *star* in which the type and notation for the Kleene star is declared. We write $(-)^\star$ for the Kleene star.

```
class star = fixes star :: 'a ⇒ 'a (-⋆ [101] 100)

class kleene-algebra = dioid + star +
  assumes star-unfoldl: 1 + x · x⋆ ≤ x⋆
  and star-unfoldr: 1 + x⋆ · x ≤ x⋆
  and star-inductl: z + x · y ≤ y ⟹ x⋆ · z ≤ y
  and star-inductr: z + y · x ≤ y ⟹ z · x⋆ ≤ y
```

The type declaration *star* :: $'a \Rightarrow 'a$ in class *star* shows for the first time the type parameter $'a$ of Kleene algebras explicitly.

After opening a context, we can start proving facts like those in Lemma 2.16. We leave most of them as exercises and discuss only a few to highlight other Isabelle features.

```
lemma star-infl: x ≤ x⋆
proof−
  have x = x · 1
    by simp
  also have ... ≤ x · x⋆
    using mult-isol one-le-star by force
  also have ... ≤ x⋆
    by (simp add: star-unfoldlr)
  finally show x ≤ x⋆.
```

qed

This proof shows how equational proof steps can be chained together in textbook style using keyword **also have**. Such steps are combined (by transitivity) by **finally have**, or else by **finally show** if this proves the overall goal.

Isabelle supports proofs by induction. In the one below, powers are defined recursively in class *monoid-mult*. Isabelle supplies an induction principle and generates proof obligations for the base case and the induction step. We have slightly rewritten the proof template suggested by jEdit to make the proof more readable.

lemma *star-power*: $x \hat{} i \leq x^\star$
proof (*induct i*)
  case *0*
  show $x \hat{} 0 \leq x^\star$
    by (*simp add*: *one-le-star*)
next
  case (*Suc i*)
  assume $x \hat{} i \leq x^\star$
  have $x \hat{} Suc\ i = x \cdot x \hat{} i$
    by *simp*
  also have $\ldots \leq x \cdot x^\star$
    by (*simp add*: *Suc.hyps mult-isol*)
  also have $\ldots \leq x^\star$
    by (*simp add*: *star-unfoldlr*)
  finally show $x \hat{} Suc\ i \leq x^\star$**.**
qed

We have already mentioned that reasoning about the star is mainly inequational. The following set-up, using the antisymmetry axiom of partial orders, splits equational proof goals automatically into two inequalities.

lemma *star-unfoldl-eq* [*simp*]: $1 + x \cdot x^\star = x^\star$
proof (*rule antisym*)
  show *le*: $1 + x \cdot x^\star \leq x^\star$
    by (*simp add*: *star-unfoldl*)
  have $1 + x \cdot (1 + x \cdot x^\star) = 1 + x + x \cdot x \cdot x^\star$
    by (*simp add*: *add-assoc distl mult-assoc*)
  also have $\ldots \leq 1 + x \cdot x^\star$
    by (*smt calculation le add-assoc distl add-idem order-def*)
  finally have $1 + x \cdot (1 + x \cdot x^\star) \leq 1 + x \cdot x^\star$**.**
  thus $x^\star \leq 1 + x \cdot x^\star$
    using *star-inductl* **by** *fastforce*
qed

The keyword [*simp*] adds this fact as a rewrite rule (from left to right) to Isabelle's simplifiers so that tools like *simp* can apply it under the hood. Simplification rules can greatly enhance proof performance, but must be added with care as it is easy to make the simplifier loop. As a rule of thumb, an equation is a candidate for simplification if its right-hand side is obviously simpler than its left-hand side.

The first line of the proof introduces the label *le* as a local name, which is used in the proof of the third line. Labels are generally useful when hypotheses or intermediate results cannot be linked immediately with **hence**.

3.1.5. Kleene Algebras with Tests. KAT, as defined in Section 2.4, requires two type parameters—one for the Kleene algebra of programs and one for the boolean algebra of tests. It is therefore impossible to formalise it in the style of Definition 2.29 as a type class. So we use a trick to encode tests via an endofunction $\tau : K \to K$. The idea is to axiomatise τ in such a way that the image $\tau(K)$ of K under τ forms the boolean subalgebra of test elements, that is, τ maps elements of K to the boolean subalgebra $B = \tau(K)$ of K.

Suitable axioms need to ensure that addition and multiplication is closed on the subalgebra and that multiplication on the subalgebra corresponds to meet. In particular, τ should be an identity on tests, that is, $\tau\,(\tau\,x) = \tau\,x$. It then follows that $\tau\,x = x \Leftrightarrow x \in \tau(K)$, so that the tests are precisely the fixpoints of τ. We can then write $\tau\,x$ to express that x is a test.

The only problem is that τ does not allow us to express boolean complementation on the subalgebra. We therefore start from an *antitest function* $\alpha : K \to K$ that maps each element of K to the boolean complement of a test, in the sense that $\tau = \alpha \circ \alpha$, $\alpha\,x + \tau\,x = 1$ and $\alpha\,x \cdot \tau\,x = 0$. The precise choice of the axioms for antitests that meet these requirements are not important for us. What matters is that they construct the boolean subalgebra B of tests as described.

```
class kat = kleene-algebra +
  fixes atest :: 'a ⇒ 'a (α)
  assumes test-one [simp]: α (α 1) = 1
  and test-mult [simp]: α (α (α (α x) · α (α y))) = α (α y) · α (α x)
  and test-mult-comp [simp]: α x · α (α x) = 0
  and test-de-morgan: α x + α y = α (α (α x) · α (α y))
```

The test function τ can then be defined in class *kat* as described.

```
definition test :: 'a ⇒ 'a  (τ) where
  τ x = α (α x)
```

Proofs in KAT can now be performed as usual, see the theory files for examples and exercises. In particular, we have proved that the algebra of test elements forms indeed a boolean subalgebra. To express that an element x is a test we can simply write $\tau\,x$. Further, we can indicate test complementation by writing $\alpha\,x$, which is equal to $\alpha\,(\tau\,x)$. More importantly, we can now formalise the algebraic semantics of conditionals and while loops in Isabelle.

```
definition cond :: 'a ⇒ 'a ⇒ 'a ⇒ 'a (if - then - else - fi [64,64,64] 63) where
  if p then x else y fi = τ p · x + α p · y

definition while :: 'a ⇒ 'a ⇒ 'a (while - do - od [64,64] 63) where
  while p do x od = (τ p · x)⋆ · α p
```

3.1.6. Finding Models and Counterexamples. Remark 2.33 in Chapter 2 shows a 3-element KAT which refutes the proposition that all subidentities in a

KAT must be tests. Such counterexamples can often be found with Isabelle, using its *Nitpick* tool. Nitpick is essentially at SAT-solver that enumerates small finite structures that falsify a given proof goal. In our case, it would try to find a KAT in which the proposition $x \leq 1 \Rightarrow \tau x = x$ is false. With Isabelle, we write

lemma (**in** *kat*) $x \leq 1 \Longrightarrow \tau\ x = x$
  nitpick
oops

to invoke Nitpick. The command **oops** allows Isabelle to ignore this lemma and proceed. Indeed, after a short search, the 3-element KAT from Remark 2.33 appears in the output window, represented as a list of elements and tables for α, $<$, $\leq$, $+$ and $\cdot$. Bringing Isabelle's output into more standard mathematical form yields the carrier set $\{0, a, 1\}$ (using simpler names for elements), the chain $0 < a < 1$ and the tables

$+$	0	a	1
0	0	a	1
a	a	a	1
1	1	1	1

$\cdot$	0	a	1
0	0	0	0
a	0	a	a
1	0	a	1

	α
0	1
a	0
1	0

The table for $\leq$ does not yield any additional information. In fact, even the table for $+$ is determined by the chain $0 < a < 1$. The table for $\cdot$ shows that multiplication coincides with meet. The table for α allows us to determine $\tau : 0 \mapsto 0, a \mapsto 1, 1 \mapsto 1$. The set of fixpoints of τ and hence the elements of the boolean algebra of tests is indeed $B = \{0, 1\}$, the antidomain operations complements these elements. Element a, by contrast, is not a fixpoint of α and not in B—but it is of course a subidentity.

Beyond this particular counterexample, Nitpick can be very useful for detecting typos and other errors in axiom systems, checking whether expected consequences fail to hold, or that a given set of axioms is irredundant in the sense that some of these axioms do not follow from the remaining ones, in particular in combination with Sledgehammer.

Example 3.1.

(1) To show that the left distributivity axiom $x \cdot (y + z) = x \cdot y + x \cdot z$ is irredundant as an axiom for semirings, we can comment it out in the corresponding type class and ask Nitpick to find a model in which this axiom is false, while the remaining semiring axioms are true. This implies that the left distributivity axiom is not derivable from the other axioms (by soundness of first-order logic) and therefore irredundant.
(2) The right unfold axiom $1 + x^* \cdot x \leq x^*$ of Kleene algebra, by contrast is redundant and hence not needed as an axiom. When we comment this axiom out and add it as a lemma, Sledgehammer can find an proof. Nevertheless it is usually kept as an axiom because it highlights an important duality.
(3) Is the right induction axiom $z + y \cdot x \leq y \Rightarrow z \cdot x^* \leq y$ of Kleene algebras redundant? In this case, Nitpick neither finds a counterexample, nor does Sledgehammer find a proof—at least not on my machine. In fact, the axiom is irredundant, but to my knowledge, all counterexamples that

separate Kleene algebras with and without this axiom are infinite, and hence beyond the scope of Nitpick.

For more information on Nitpick, see

https://isabelle.in.tum.de/dist/Isabelle2021-1/doc/nitpick.pdf

By contrast to Sledgehammer, which internally reconstructs proofs supplied by external automated theorem provers, the counterexamples provided by Nitpick are not internally verified. In the example related to Remark 2.33 above, for example, Isabelle has not checked whether the algebra displayed is indeed a KAT. In principle, therefore, the structures supplied by Nitpick could therefore been deemed less trustworthy than the proofs found by Sledgehammer. In practice, however, the SAT-solvers used by Isabelle have large user communities and should thus be reasonably safe.

Those who do not trust external tools can of course formalise and verify counterexamples directly with Isabelle. This can be quite complicated and is beyond the scope of these lecture notes.

3.1.7. Formalising KAT as a Locale. Alternatively, KAT can be formalised as a locale with two type parameters. The syntax is very similar to that of type classes, although locales are generally more difficult to set up. We model the order-embedding of the boolean algebra in the Kleene algebra explicitly.

```
locale katloc =
  fixes test :: 'a::boolean-algebra ⇒ 'b::kleene-algebra (ι)
  and not :: 'b::kleene-algebra ⇒ 'b::kleene-algebra (!)
  assumes test-sup: ι (sup p q) = ι p + ι q
  and test-inf: ι (inf p q) = ι p · ι q
  and test-top: ι top = 1
  and test-bot: ι bot = 0
  and test-not: ι (− p) = ! (ι p)
  and test-ord-emb: ι p ≤ ι q ⟹ p ≤ q
```

The **fixes** statement shows that the names of the classes *boolean-algebra*, which is from Isabelle's main libraries, and *kleene-algebra* can be used as *sort parameters* that restrict the types α and β. Function ι embeds the boolean algebra of tests into the Kleene algebra; function ! lifts complementation on the boolean algebra to the Kleene algebra. The following lemma shows that tests now need to be decorated with ι's.

```
lemma test-eq: p = q ⟷ ι p = ι q
  by (metis eq-iff test-iso-eq)
```

We do not use this locale-based formalisation of KAT any further, though there is little difference in proof performance relative to the class-based one.

3.2. Examples: Program Transformations with Isabelle

With type classes and definitions for KAT in place we can formalise the program transformations from Section 2.5 with Isabelle, translating the proofs in this section

line-by-line into Isabelle most of the time. We only show the simple transformation in the proof of Lemma 2.34 and leave the more advanced ones as exercises.

```
lemma while-rec: while p do x od = if p then x · (while p do x od) else 1 fi
proof−
  have if p then x · (while p do x od) else 1 fi = τ p · x · (τ p · x)⋆ · α p + α p · 1
    by (simp add: cond-def mult-assoc while-def)
  also have ... = (τ p · x · (τ p · x)⋆ + 1) · α p
    by (simp add: distr)
  also have ... = (τ p · x)⋆ · α p
    by (simp add: add-commute)
  also have ... = while p do x od
    by (simp add: while-def)
  finally show ?thesis..
qed
```

3.3. Integrating Models

Chapter 2 discusses briefly models of the algebras introduced. Isabelle's **instantiation** mechanism allows formalising the model relation between algebras modelled as type classes and algebraic structures given by more concrete types—recall that type classes support parametric polymorphism. This requires interpreting the constant and operation symbols of the algebra as elements and functions defined in the model and proving that the axioms of the algebra hold in the model.

I have formalised most of the models from Chapter 2 with Isabelle in the git repository. Three examples highlight important aspects of the approach. The first one considers the semilattice of sets with union and the empty set (Example 2.4).

```
instantiation set :: (type) sup-semilattice
begin

definition plus-set :: 'a set ⇒ 'a set ⇒ 'a set where
  plus-set x y = x ∪ y

definition zero-set :: 'a set where
  zero-set = {}

instance
proof
  fix x y z :: 'a set
  show x + y + z = x + (y + z)
    by (simp add: KA.plus-set-def sup-assoc)
  show 0 + x = x
    by (simp add: plus-set-def zero-set-def)
  show x + x = x
    by (simp add: KA.plus-set-def)
  show x + y = y + x
    by (simp add: KA.plus-set-def sup-commute)
  show (x ⊆ y) = (x + y = y)
    by (simp add: KA.plus-set-def subset-Un-eq)
  show (x ⊂ y) = (x ⊆ y ∧ x ≠ y)
```

```
  by force
qed

end
```

The **instantiation** statement asserts that sets—elements of Isabelle's type *set* of arbitrary type *type*—form semilattices and hence an instance of the type *sup-semilattice*. Isabelle asks us first to associate the operation with name *plus-set* with the function symbol + and the element with name *zero-set* with the constant symbol 0 in sets (names and are prescribed by Isabelle), in this case, the operation ∪ and the element ∅. An **instance** proof is then required to checks that the semilattice axioms hold in this model. Command *intro-classes* generates the proof obligations needed. Discharging them one by one is automatic. After typing **end**, all facts Isabelle knows about semilattices are available for sets.

Our second example (Example 2.2) shows that endofunctions form a monoid. The *instantiation* statement requires a type of endofunctions. We define it as a subtype of Isabelle's function type. It is inhabited by all functions of type $\alpha \Rightarrow \alpha$.

```
typedef 'a endo = {f::'a ⇒ 'a . True}
  by simp

setup-lifting type-definition-endo
```

A proof is required to show that this type is inhabited. Function composition and the identity function can then be lifted to it. Isabelle's lifting package is set up to supply the type coercion functions *Abs-endo*, which projects on the endofuctions among the functions, and *Rep-endo*, which injects endofunctions into functions. More information about type definitions and the lifting package can be found in the documentation.

The instantiation declaration below now requires lifting the operation of function composition and the identity function to type α *endo* before the *instance* proof.

```
instantiation endo :: (type) mult-monoid
begin

lift-definition one-endo :: 'a endo is
  Abs-endo id.

lift-definition times-endo :: 'a endo ⇒ 'a endo ⇒ 'a endo is
  λx y. Abs-endo (Rep-endo x ∘ Rep-endo y).

instance
proof
  fix x y z :: 'a endo
  show x · (y · z) = (x · y) · z
    by transfer (simp add: Abs-endo-inverse fun.map-comp)
  show 1 · x = x
    by transfer (simp add: Abs-endo-inverse Rep-endo-inverse)
  show x · 1 = x
    by transfer (simp add: Abs-endo-inverse Rep-endo-inverse)
qed
```

end

The lifting of the identity function *id* to type α *endo* uses function *Abs-endo*. The (trival) proof indicated by . checks that the object defined has the right type. The lifting of function composition injects the endofunctions x and y into the function type using *Rep-endo*, composes them using ordinary function composition and projects the resulting function to an endofunction using *Abs-endo*. The associated proof checks that endofunctions are closed under composition. Command *transfer* uses generic properties of *Abs-* and *Rep*-functions to simplify the **instance** proof.

Finally, we show that sets under intersection and the set *UNIV* of all sets (the implicit carrier set mentioned) forms another semilattice. Isabelle, however, allows only one instance per type in an **instantiation**. We therefore need to include sets with meets by an **interpretation** statement, which are part of Isabelle's locale infrastructure. We use this mechanism quite frequently in later chapters for building verification components, as it is not restricted to types such as *set* or *endo*.

interpretation *inter-sl*: *sup-semilattice* $(\cap)$ *UNIV* $(\supseteq)$ $(\supset)$
proof *unfold-locales*
  fix X Y Z :: $'a$ *set*
  show $X \cap Y \cap Z = X \cap (Y \cap Z)$
    by (*simp add*: *Int-assoc*)
  show $X \cap Y = Y \cap X$
    by (*simp add*: *inf-commute*)
  show *UNIV* $\cap X = X$
    by *simp*
  show $X \cap X = X$
    by *simp*
  show $(Y \subseteq X) = (X \cap Y = Y)$
    by *blast*
  show $(Y \subset X) = (Y \subseteq X \wedge X \neq Y)$
    by *auto*
qed

The list of operations and relations in the **interpretation** declaration associates the function and predicate symbols of the semiattice class with our model. Typing *unfold-locales* now exposes the proof obligations. This time we have chosen to discharge them step by step in an Isar proof. We could have given a similar **instance** proof for union and the empty set above. We could also have given an interpretation proof for endofunctions without defining a subtype, adding the type declaration for functions explicitly to the list of operations after the **interpretation** statement.

CHAPTER 4

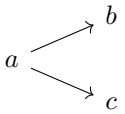
Two Semantics of Program Execution

KAT is an impressive tool for reasoning equational about while programs, but too abstract for describing in detail how programs act on state spaces when variable assignments update program stores. In this chapter we introduce two models of KAT as an intermediate step towards such concrete semantics. They describe how programs transform unstructured state spaces, but do not yet capture how assignments transform program stores. The first semantics models programs as binary relations between input and output states. The inherent nondeterminism of relations makes modelling while-loops and failure easier. Alternatively, we model programs as (nondeterministic) state transformers that map input states to sets of output states. This model is rooted in category theory and the theory of monads, but we leave these connections implicit. Both semantics are standard. We start with binary relations.

4.1. Relational Semantics

Suppose that set X is the state space of Imp, the set of program stores on which programs act. We consider concrete program stores in detail in Chapter 7. For now it suffices to assume that state spaces form sets. We model programs as binary relations $R \subseteq X \times X$, so that $(a, b) \in R$ means that $a \in X$ is an input state and $b \in X$ an output state that program R relates with a. In other words, if R is executed from a, then b is one of the states in which R may terminate. We write $\mathsf{Rel}\, X = \mathcal{P}(X \times X)$ for the set of all binary relations on X. For us, $\mathsf{Rel}\, X$ is the set of all programs on state space X.

Relations admit nondeterminism when an element is related to more than one element, for instance,



where we simply write $a \to b$ instead of $(a, b) \in R$. They also allow that an element is related to no other element. This distinguishes relations on X from functions, and even partial functions on X that need not be defined everywhere on X.

4.1.1. Basic Algebra of Binary Relations. Relations are sets and hence form (powerset) boolean algebras. The least element in the boolean algebra of relations is the *empty relation* $\emptyset_X$ on $X \times X$; the greatest element is the *universal relation* $U_X = X \times X$. One can take unions and intersections of relations as their joins and meets, and complements of any relation like in Example 2.25.

The union of relations is more important for modelling programs than intersection or complementation. The union of two programs as relations models their

nondeterministic choice. Intersecting two programs may seem less interesting, and few people may ever have attempted to take the complement of a program in applications.

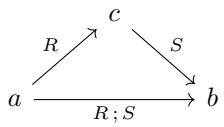
Building a relational relational model of KAT requires one additional element and two additional operations. The *identity relation* on X is

$$Id_X = \{(a,a) \mid a \in X\}.$$

The *relational composition* of $R, S \in \mathsf{Rel}\, X$ is the relation

$$R\,;S = \{(a,b) \mid \exists c.\ (a,c) \in R \wedge (c,b) \in S\}.$$

Hence $(a,b) \in R\,;S \Leftrightarrow \exists c.\ (a,c) \in R \wedge (c,b) \in S$.



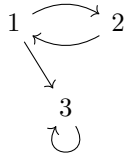
Finally, with $R^0 = Id_X$ and $R^{i+1} = R\,;R^i$ for all $i \in \mathbb{N}$, the *reflexive transitive closure* of $R \in \mathsf{Rel}\, X$ is the relation

$$R^* = \bigcup_{i \in \mathbb{N}} R^i\,.$$

Hence $(a,b) \in R^* \Leftrightarrow \exists i \in \mathbb{N}.\ (a,b) \in R^i$, which means that R^* is R^i for some nondeterministically chosen $i \in \mathbb{N}$.

Remark 4.1. Binary relations on $\mathsf{Rel}\, X$ correspond to directed graphs with vertices in X and arrows in R. Finite relations with $|X| = n$, for some $n \in \mathbb{N}$, can be modelled by $n \times n$ (adjacency) matrices over the booleans $\mathbb{B}$ with addition and multiplication defined as in Example 2.12. The union of relations then corresponds to matrix addition; relational composition to matrix multiplication. The empty relation corresponds to the zero matrix and the identity relation to the diagonal matrix. The reflexive transitive closure becomes a sum of matrix iterations. It becomes stationary after at most n^2 steps.

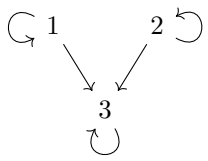
Example 4.2. Relation $R = \{(1,2),(1,3),(2,1),(3,3)\}$ over $X = \{1,2,3\}$ is represented by the digraph and matrix



$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

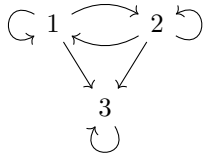
where rows are implicitly indexed by 1, 2 and 3 from left to right and columns by 1, 2 and 3 from top to bottom.

Relation $R\,;R = \{(1,1),(1,3),(2,2),(2,3),(3,3)\}$ captures two-step reachability with respect to R.



$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

The reflexive-transitive closure $R^* = \{(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,3)$ thus captures global reachability in R, including self-reachability.



$$R^* = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

□

REMARK 4.3. For proper reachability, the transitive closure of R can be used. It is defined as $R^+ = \bigcup_{i\in\mathbb{N}} R^{i+1}$. It is easy to check that $R^* = Id_X \cup R^+$ and $R^+ = R\,;R^*$.

4.1.2. Relation KAT. To show that binary relations form KATs we proceed via dioids and Kleene algebras.

PROPOSITION 4.4. *$(\mathsf{Rel}\,X, ;, \cup, \emptyset_X, Id_X)$ forms a dioid.*

PROOF. We need to show that $(\mathsf{Rel}\,X, ;, Id_X)$ is a monoid, that $(\mathsf{Rel}\,X, \cup, \emptyset_X)$ is a semilattice, that relational composition distributes over union in both arguments and that $\emptyset_X$ is a left and right zero of relational composition. We prove associativity of relational composition as an example. Let $R, S, T \in \mathsf{Rel}\,X$. Then

$$\begin{aligned}
(a,b) \in R\,;(S\,;T) &\Leftrightarrow \exists c.\ (a,c) \in R \wedge (c,b) \in S\,;T \\
&\Leftrightarrow \exists c.\ (a,c) \in R \wedge (\exists d.\ (c,d) \in S \wedge (d,b) \in T) \\
&\Leftrightarrow \exists c, d.\ (a,c) \in R \wedge (c,d) \in S \wedge (d,b) \in T \\
&\Leftrightarrow \exists d.\ (\exists c.\ (a,c) \in R \wedge (c,d) \in S) \wedge (d,b) \in T \\
&\Leftrightarrow \exists d.\ (a,d) \in R\,;S \wedge (d,b) \in T \\
&\Leftrightarrow (a,b) \in (R\,;S)\,;T.
\end{aligned}$$

Associativity of relational composition then follows because two sets are equal if and only if they have the same elements. □

The following generalised distributivity or continuity laws for relational composition are useful for showing that $\mathsf{Rel}\,X$ forms a Kleene algebra.

LEMMA 4.5. *Let X be a set, $R, S \in \mathsf{Rel}\,X$ and $\mathcal{R}, \mathcal{S} \subseteq \mathsf{Rel}\,X$. Then*

(1) $R\,;(\bigcup \mathcal{S}) = \bigcup_{S\in\mathcal{S}} R\,;S$,
(2) $(\bigcup \mathcal{R})\,;S = \bigcup_{R\in\mathcal{R}} R\,;S$.

PROOF. We only show (1); (2) follows by opposition.

$$\begin{aligned}
(a,b) \in R\,;\Big(\bigcup \mathcal{S}\Big) &\Leftrightarrow \exists c.\ (a,c) \in R \wedge (c,b) \in \bigcup \mathcal{S} \\
&\Leftrightarrow \exists c.\ (a,c) \in R \wedge \exists S \in \mathcal{S}.\ (c,b) \in S \\
&\Leftrightarrow \exists S \in \mathcal{S}. \exists c.\ (a,c) \in R \wedge (c,b) \in S \\
&\Leftrightarrow \exists S \in \mathcal{S}.\ (a,b) \in R\,;S \\
&\Leftrightarrow (a,b) \in \bigcup_{S\in\mathcal{S}} R\,;S.
\end{aligned}$$

□

PROPOSITION 4.6. $(\mathsf{Rel}\, X, ;, \cup, \emptyset_X, Id_X, ^*)$ *forms a Kleene algebra.*

PROOF. Relative to Proposition 4.4, it remains to verify the star axioms. This is left as an exercise. □

REMARK 4.7. The fact that continuity of relational composition is needed for proving the star induction laws in the relational model sheds further light on these laws. Strictly speaking, the function $\lambda x.\ z + x \cdot y$ has a binary least fixpoint $x^{*_2} z$ that depends on the parameters z and x and satisfies

$$z + x \cdot (x^{*_2} z) = x^{*_2} z, \qquad z + x \cdot y = y \Rightarrow x^{*_2} z \leq y.$$

In the proof of Proposition 4.6 we see how, in the relational model, continuity allows us do decompose this binary fixpoint into a unary one and a relational composition, that is, $x^{*_2} z = x^* \cdot z$, while relating the unary fixpoint with an iteration. The left star induction axiom $z + x \cdot y \leq y \Rightarrow x^* \cdot z \leq y$ of Kleene algebra therefore captures the least fixpoint property of the binary fixpoint $x^{*_2} z$ and its decomposition into the unary fixpoint $x^* \cdot z$ at the same time, using continuity implicitly. The left star unfold axiom $1 + x \cdot x^* \leq x^*$ becomes the fixpoint law for the binary fixpoint simply by multiplying both sides of the inequality with z and using $x^{*_2} z = x^* \cdot z$. In sum, the left star axioms of Kleene algebra thus do not axiomatise x^*, but $x^{*_2} y$ and its decomposition into $x^* \cdot z$ by continuity, without being able to express continuity in the language of Kleene algebra. A dual argument holds for the right star axioms.

REMARK 4.8. The relationship between fixpoints, continuity and iteration can be studied in algebra. By Knaster-Tarski's fixpoint theorem, every order-preserving function over a complete lattice has a least fixpoint. A complete lattice is a partial order in which each set has a supremum. Boolean algebras need only have binary suprema or joins, but powerset boolean algebras and therefore algebras of binary relations have arbitrary suprema, represented by $\bigcup$. The function $\lambda X.\ R \cup S ; X$ is order-preserving over the complete lattice of binary relations in $\mathsf{Rel}\, X$, hence the binary least fixpoint $R^{*_2} S$ exists.

By Kleene's fixpoint theorem, every continuous function (which means that the function distributes over suprema in the sense of Lemma 4.5) can be represented by an iteration. The function $\lambda X.\ R \cup S ; X$ is continuous over the complete lattice of binary relations in $\mathsf{Rel}\, X$, but continuity cannot be expressed in Kleene algebra because $+$ can only model finite suprema.

Finally, fixpoint fusion theorems provide conditions for decomposing fixpoints of functions over a complete lattice. Without mentioning details, it can be shown that the functions $f = \lambda X.\ X ; R$ and $g = \lambda X.\ Id \cup S ; X$ satisfy these conditions, in particular, f needs to be continuous, so that the fixpoint of $f \circ g = \lambda X.\ R \cup S ; X$ becomes function f applied to the fixpoint of g. Once again these conditions can be shown to hold in the relational model.

Further details can be found in the literature.

Our main theorem—that relations form KATs—requires additional definitions. First, we write

$$\mathsf{Id}{\downarrow_X} = \{R \in \mathsf{Rel}\, X \mid R \subseteq Id_X\}$$

for the set of all *subidentity relations*. All elements of a subidentity are therefore of the form (a, a) with $a \in X$; subidentities therefore relate elements of X either with themselves or with no other element. We write $P, Q, \ldots$ for relational subidentities. Subidentities are in one-to-one correspondence with predicates ranging over X and

with subsets of X. Hence we can identify $\mathsf{Id}{\downarrow_X}$ with $\mathcal{P}X$ and may write $P\,a$ instead of $(a,a) \in P$. Next, $\overline{P} = Id_X - P$ defines the complement of $P \in \mathsf{Id}{\downarrow_X}$ within $\mathsf{Id}{\downarrow_X}$.

THEOREM 4.9. $(\mathsf{Rel}\,X, \mathsf{Id}{\downarrow_X}, ;, \cup, \overline{(-)}, \emptyset_X, Id_X, ^*)$ *forms a* KAT.

PROOF. Relative to Proposition 4.6 it remains to show that $\mathsf{Id}{\downarrow_X}$ forms a subalgebra of $\mathsf{Rel}\,X$ that is a boolean algebra with ; as intersection and $\overline{(-)}$ as complementation restricted to this subalgebra. In particular, it must be shown that all unions, intersections, complements (with respect to $\overline{(-)}$) and stars of elements in $\mathsf{Id}{\downarrow_X}$ are again in $\mathsf{Id}{\downarrow_X}$. Details are left as an exercise. □

Because of this result, we call $\mathsf{Rel}\,X$ the *relation Kleene algebra with tests* over X. As we identify $\mathsf{Rel}\,X$ with the set of (abstract) programs with state space X, Theorem 4.9 puts our programming intuitions for KAT from Chapter 2 on solid semantic foundations. In particular, two programs are equivalent if they are equal as relations, and a program R is smaller than a program S if $R \subseteq S$. Theorem 4.9 yields a *soundness proof* for KAT and its abstract program semantics with respect to the more concrete relational semantics. It remains to link it with program stores.

REMARK 4.10. Every partial order is of course a binary relation. More generally, a relation $R \in \mathsf{Rel}\,X$ is called *reflexive* if $(a,a) \in R$ for all $a \in X$ and *transitive* if $(a,b) \in R$ and $(b,c) \in R$ imply $(a,c) \in R$ for all $a,b,c \in X$. It is easy to show that R is reflexive if and only if $Id_X \subseteq R$ and transitive if and only if $R\,;R \subseteq R$. It then follows from Theorem 4.9, Lemma 2.16 and remark 2.18 that R^* is indeed the reflexive-transitive closure of R.

REMARK 4.11. We have claimed in Chapter 2 that px and xq model the restriction of program x to input states where predicate p holds and output states where predicate q holds in KAT. We can now check this in the relational semantics. Let $R \in \mathsf{Rel}\,X$ and $P,Q \in \mathsf{Id}{\downarrow_X}$. Then, identifying subidentities and predicates,

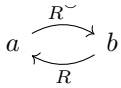
$$P\,;R = \{(a,b) \mid \exists c.\ (a,c) \in P \wedge (c,b) \in R\} = \{(a,b) \mid P\,a \wedge (a,b) \in R\}$$

and $R\,;Q = \{(a,b) \mid (a,b) \in R \wedge Q\,b\}$ by opposition.

4.1.3. Further Operations on Binary Relations. The algebra of binary relations admits further operations. We briefly mention three of them. The *converse* of relation $R \in \mathsf{Rel}\,X$ is the relation

$$R^\smile = \{(a,b) \mid (b,a) \in R\}.$$

Hence $(a,b) \in R^\smile \Leftrightarrow (b,a) \in R$.



In the relational program semantics (but outside of relation KAT), conversion swaps the execution order of programs. We can therefore express opposition explicitly in terms of conversion, using the following lemma.

LEMMA 4.12. *Let* $R \in \mathsf{Rel}\,X$. *Then*

(1) $\emptyset_X^\smile = \emptyset_X$, $Id_X^\smile = Id_X$ *and* $U_X^\smile = U_X$,
(2) $R^{\smile\smile} = R$,
(3) $(R \cup S)^\smile = R^\smile \cup S^\smile$ *and* $(R \cap S)^\smile = R^\smile \cap S^\smile$,
(4) $(R\,;S)^\smile = S^\smile\,;R^\smile$,

(5) $(-R)^\smile = -(R^\smile)$,
(6) $(R^*)^\smile = (R^\smile)^*$.

PROOF. We check (6) as an example. First, we prove by induction on i that $\left(R^i\right)^\smile = \left(R^\smile\right)^i$ for all $i \in \mathbb{N}$. In the base case,

$$\left(R^0\right)^\smile = Id_X^\smile = Id_X = \left(R^\smile\right)^0.$$

For the inductive step, suppose $(R^i)^\smile = \left(R^\smile\right)^i$. Then

$$\left(R^{i+1}\right)^\smile = \left(R\,;R^i\right)^\smile = \left(R^i\right)^\smile ;R^\smile = \left(R^\smile\right)^i ;R^\smile = \left(R^\smile\right)^{i+1}.$$

Finally,

$$\begin{aligned}(a,b) \in (R^*)^\smile &\Leftrightarrow (b,a) \in R^* \\ &\Leftrightarrow \exists i \in \mathbb{N}.\ (b,a) \in R^i \\ &\Leftrightarrow \exists i \in \mathbb{N}.\ (a,b) \in \left(R^i\right)^\smile \\ &\Leftrightarrow \exists i \in \mathbb{N}.\ (a,b) \in \left(R^\smile\right)^i \\ &\Leftrightarrow (a,b) \in \left(R^\smile\right)^*.\end{aligned}$$

□

The *domain* and *range* (*codomain*) of $R \in \mathsf{Rel}\, X$ are defined as

$$dom\, R = \{a \mid \exists b.\ (a,b) \in R\} \qquad \text{and} \qquad ran\, R = \{b \mid \exists a.\ (a,b) \in R\}.$$

Hence $x \in dom\, R \Leftrightarrow \exists b.\ (a,b) \in R$ and $b \in ran\, R \Leftrightarrow \exists a.\ (a,b) \in R$.

Relations satisfy even more complex laws such as $R^\smile ;(-(R\,;S)) \subseteq -S$, but these are irrelevant for our purposes.

4.2. State Transformer Semantics

Next, we present a state transformer semantics that considers programs as nondeterministic functions that transform program states. This is perhaps a more natural view of programs as transformations on a state space than binary relations, but the two models are isomorphic, as we will see.

4.2.1. Basic Algebra of State Transformers. A *state transformer* is a function of type $X \to \mathcal{P}X$. It maps elements of X to subsets of X, including the empty set. We write $\mathsf{Sta}\, X$ for the set of all state transformers on X. We now define the Kleene algebra operations on $\mathsf{Sta}\, X$.

To compose two state transformers $f, g : X \to \mathcal{P}X$, f after g, we first map an element a by $f : a \mapsto B$ to some $B \subseteq X$. Then we map every $b \in B$ by $g : b \mapsto C_b$ to some $C_b \subseteq X$. Combining both steps yields $c \in (f \circ_K g)\, a \Leftrightarrow \exists b.\ b \in f\, a \wedge c \in g\, b$, which defines the *Kleisli composition*

$$(f \circ_K g)\, a = \bigcup \{g\, b \mid b \in f\, a\}.$$

REMARK 4.13. In the standard definition of Kleisli composition, the functions are applied like in function composition $(f \circ g)\, x = f\, (g\, x)$, that is, our right-hand side would in fact define $(g \circ_K f)$. We have chosen the reverse order that works in the direction of relational composition. Otherwise we would have to reverse all algebraic definitions so that **if** p **then** x **else** y, for instance, would become $x \cdot p + y \cdot \overline{p}$.

The *identity state transformer* $\eta_X : X \to \mathcal{P}X$ maps every element $a \in X$ to the singleton set $\{a\}$ containing it, that is,

$$\eta_X = \{-\}.$$

And indeed, it is easy to check that, for all $a \in X$,

$$(f \circ_K \eta_X)\, a = \bigcup \{\{b\} \mid b \in f\, a\} = f\, a,$$
$$(\eta_X \circ_K f)\, a = \bigcup \{f\, b \mid b \in \{a\}\} = \bigcup \{f\, a\} = f\, a,$$

The *sum* $f + g : X \to \mathcal{P}X$ of state transformers $f, g : X \to \mathcal{P}X$ is defined by so-called *pointwise extension* as

$$(f + g)\, a = f\, a \cup g\, a.$$

It is immediate from this definition that $+$ is associative, commutative and idempotent. It also follows that

$$f \leq g \Leftrightarrow \forall a \in X.\ f\, a \subseteq g\, a.$$

The *star* of state transformer $f : X \to \mathcal{P}X$ is defined, with $f^{0_K} = \eta_X$ and $f^{(i+1)_K} = f \circ_K f^{i_K}$, as

$$f^{*_K}\, a = \bigcup_{i \in \mathbb{N}} f^{i_K}\, a,$$

for all $a \in X$. Thus $b \in f^{*_K}\, a \Leftrightarrow \exists i \in \mathbb{N}.\ b \in f^{i_K}$.

Finally, the *zero* state transformer is

$$0_X = \lambda x.\ \emptyset.$$

Next we consider the boolean subalgebra of KAT in the context of $\mathsf{Sta}\, X$.

A state transformer $f \in \mathsf{Sta}\, X$ is a *subidentity* if $f \leq \eta_X$. By definition, it maps elements $a \in X$ either to $\{a\}$ or to $\emptyset$:

$$b \in f\, a \ \Leftrightarrow\ b = a \wedge f\, a \neq \emptyset \ \Leftrightarrow\ b = a \wedge f\, a = \{a\}.$$

We write $\eta{\downarrow_X}$ for the set of all subidentity state transformers and $P, Q \ldots$ for subidentities, for notational coherence with respect to relations and predicates. Subidentities are once again in one-to-one correspondence with predicates and sets. Hence we can identify $\eta{\downarrow_X}$ with $\mathcal{P}X$ and write $P\, a$ instead of $P\, a \neq \emptyset$. We discuss the relationship between subidentity state transformers and relational subidentities in the following section.

It remains to define the *complementation* of a subidentity state transformer $P \in \eta{\downarrow_X}$ as $\overline{P} = \eta_X - P$, where $(f - g)\, a = f\, a - g\, a$ is defined by pointwise extension, as expected. Unfolding definitions,

$$\overline{P}\, a = \begin{cases} \{a\}, & \text{if } P\, a = \emptyset, \\ \emptyset, & \text{if } P\, a = \{a\}. \end{cases}$$

Remark 4.14. It is easy to check that $f \circ_K Q$ and $P \circ_K f$ restrict once again the inputs of program f to states where $P \in \eta{\downarrow_X}$ holds and its outputs to states where $Q \in \eta{\downarrow_X}$ holds. We leave this as an exercise.

We could now show that $\mathsf{Sta}\, X$ forms a KATs. However, we do not perform this proof directly and obtain if from that for binary relations in the next section. A direct proof can be found in the Isabelle theories.

4.2.2. Isomorphism Between Relations and State Transformers. It is often important to compare and formally relate different models and semantics of programs. One way is to use mappings between models that relate or even preserve their structure, which then implies that mappings preserve operations. In mathematics, structure-preserving functions between algebras are known as homomorphisms. Models that have the same structure are related by isomorphisms, which are bijections that preserve all operations. As an example, we define an isomorphism between $\mathsf{Rel}\,X$ and $\mathsf{Sta}\,X$ that preserves the KAT operations.

We associate the state transformer $f_R = \lambda x.\ \{b \mid (x,b) \in R\}$ in $\mathsf{Sta}\,X$ with each relation $R \in \mathsf{Rel}\,X$ and the relation $R_f = \{(a,b) \mid b \in f\,a\}$ in $\mathsf{Rel}\,X$ with each state transformer $f \in \mathsf{Sta}\,X$. This defines two *functors* $\mathcal{S} : \mathsf{Rel}\,X \to \mathsf{Sta}\,X$ and $\mathcal{R} : \mathsf{Sta}\,X \to \mathsf{Rel}\,X$ by

$$\mathcal{S}\,R\,a = \{b \in X \mid (a,b) \in R\} \qquad \text{and} \qquad \mathcal{R}\,f = \{(a,b) \mid b \in f\,a\}\,.$$

These are essentially the curry and uncurry functions known from functional programming. We wish to show that they are isomorphisms that preserve the KAT structure of $\mathsf{Rel}\,X$ and $\mathsf{Sta}\,X$. This requires proving that they are a pair of injective and surjective functions that map between ; and $\circ_K$, $\cup$ and $+_K$, $\emptyset$ and 0_X, Id_X and η_X, and $(-)^*$ and $(-)^{*_K}$ and preserve subidentities. This is the subject of the following lemma.

LEMMA 4.15. *The functions $\mathcal{S}$ and $\mathcal{R}$ are a bijective pair. They also satisfy*

(1) $\mathcal{S}\,(R\,;S) = \mathcal{S}\,R \circ_K \mathcal{S}\,S$ *and* $\mathcal{R}\,(f \circ_K g) = \mathcal{R}\,f\,;\mathcal{R}\,g$,
(2) $\mathcal{S}\,Id_X = \eta_X$ *and* $\mathcal{R}\,\eta_X = Id_X$,
(3) $\mathcal{S}\,\emptyset = 0_K$ *and* $\mathcal{R}\,0_K = \emptyset$,
(4) $\mathcal{S}\,(R \cup S) = \mathcal{S}\,R + \mathcal{S}\,S$ *and* $\mathcal{R}\,(f+g) = \mathcal{R}\,f \cup \mathcal{R}\,g$,
(5) $\mathcal{S}\,R^* = (\mathcal{S}\,R)^{*_K}$ *and* $\mathcal{R}\,f^{*_K} = (\mathcal{R}\,f)^*$.

PROOF. We first show that $\mathcal{S} \circ \mathcal{R} = id_{\mathsf{Sta}\,X}$ and $\mathcal{R} \circ \mathcal{S}\,R = id_{\mathsf{Rel}\,X}$:

$$\mathcal{S}\,(\mathcal{R}\,f)\,a = \{b \mid (a,b) \in \mathcal{R}\,f\} = \{b \mid b \in f\,a\} = f\,a,$$
$$\mathcal{R}\,(\mathcal{S}\,R) = \{(s,b) \mid b \in (\mathcal{S}\,R)\,a\} = \{(a,b) \mid (a,b) \in R\} = R.$$

It follows that $\mathcal{S}$ and $\mathcal{R}$ are injective,

$$\mathcal{S}\,R = \mathcal{S}\,S \Rightarrow \mathcal{R}\,(\mathcal{S}\,R) = \mathcal{R}\,(\mathcal{S}\,S) \Rightarrow R = S,$$

and likewise for $\mathcal{R}$. Beyond that, $\mathcal{S}$ and $\mathcal{R}$ are surjective by definition of R_f and f_R. Thus $\mathcal{S}$ and $\mathcal{R}$ form a bijective pair. Next we show (5) as an example. We show by induction on i that $\mathcal{S}\,R^i = (\mathcal{S}\,R)^{i_K}$ for all $i \in \mathbb{N}$. The base case is (2). For the induction step, suppose $\mathcal{S}\,R^i = (\mathcal{S}\,R)^{i_K}$. Then

$$\mathcal{S}\,R^{i+1} = \mathcal{S}\,(R\,;R^i) = \mathcal{S}\,R \circ_K (\mathcal{S}\,R)^{i_K} = (\mathcal{S}\,R)^{(i+1)_K},$$

using (1), and therefore

$$\begin{aligned}
\mathcal{S}\,R^*\,a &= \left\{b \,\middle|\, \exists i \in \mathbb{N}.\ (a,b) \in R^i\right\} \\
&= \bigcup_{i \in \mathbb{N}} \mathcal{S}\,R^i\,a \\
&= \bigcup_{i \in \mathbb{N}} (\mathcal{S}\,R)^{i_K}\,a \\
&= (\mathcal{S}\,R)^{*_K}\,a.
\end{aligned}$$

Next we show by induction on i that $\mathcal{R}\, f^{*_K} = (\mathcal{R}\, f)^*$. The base case is (2). For the induction, step suppose $\mathcal{R}\, f^{*_K} = (\mathcal{R}\, f)^*$. Then

$$\mathcal{R}\, f^{(i+1)_K} = \mathcal{R}\,(f \circ_K f^{i_K}) = \mathcal{R}\, f\,;(\mathcal{R}\, f)^i = (\mathcal{R}\, f)^{(i+1)},$$

using (1), and therefore

$$\begin{aligned}\mathcal{R}\, f^{*_K} &= \left\{(a,b) \,\middle|\, \exists i \in \mathbb{N}.\ b \in f^{i_K}\, a\right\}\\ &= \bigcup_{i\in\mathbb{N}} \mathcal{R}\, f^{i_K}\\ &= \bigcup_{i\in\mathbb{N}} (\mathcal{R}\, f)^i\\ &= (\mathcal{R}\, f)^*.\end{aligned}$$

The remaining proofs are left as exercises. □

LEMMA 4.16. *The functors $\mathcal{S}$ and $\mathcal{R}$ satisfy*

$$\mathcal{S}\,\overline{R} = \overline{\mathcal{S}\, R} \qquad \textit{and} \qquad \mathcal{R}\,\overline{f} = \overline{\mathcal{R}\, f}.$$

PROOF.

$$\mathcal{S}\,\overline{R}\, a = \{b \mid (a,b) \in Id_X \wedge (a,b) \notin R\} = \{b \mid b \in \mathcal{S}\,\eta_X\, a - \mathcal{S}\, R\, a\} = \overline{\mathcal{S}\, R}\, a,$$
$$\mathcal{R}\,\overline{f} = \{(a,b) \mid b \in \eta_X\, a \wedge y \notin f\, a\} = \mathcal{R}\,\eta_X - \mathcal{R}\, f = Id_X - \mathcal{R}\, f = \overline{\mathcal{R}\, f}.$$

□

REMARK 4.17. The functors $\mathcal{R}$ and $\mathcal{S}$ set up the isomorphisms between subidentity state transformers, subidentity relations, predicates and sets as well. For every $P \in \mathsf{Id}{\downarrow_X}$ we can use $\mathcal{S}$ to calculate the isomorphic $f_P \in \eta{\downarrow_X}$:

$$f_P\, a = \{b \in X \mid P\, a\} = \begin{cases}\{a\}, & \text{if } P\, a,\\ \emptyset, & \text{otherwise,}\end{cases}$$

identifying subidentity relations with predicates in this definition.

For every $P \in \eta{\downarrow_X}$ we can use $\mathcal{R}$ to calculate the isomorphic subidentity relation $R_P \in \mathsf{Id}{\downarrow_X}$ as $R_p = \mathcal{R}\, P = \{(a,a) \mid P\, a \neq \emptyset\}$. The isomorphic set is therefore given by $\{a \mid P\, a\}$, identifying subidentity state transformers with predicates.

4.2.3. State Transformer KAT. We can now use $\mathcal{R}$ and $\mathcal{S}$ to infer the KAT structure on $\mathsf{Sta}\, X$ from that in $\mathsf{Rel}\, X$.

PROPOSITION 4.18. $(\mathsf{Sta}\, X, \circ_K, +, 0_X, \eta_X, {}^{*_K})$ *forms a Kleene algebra.*

PROOF. This is an immediate consequence of the isomorphism with $\mathsf{Rel}\, X$. Concretely, we can calculate instances of axioms in $\mathsf{Sta}\, X$ from their counterparts in $\mathsf{Rel}\, X$ using $\mathcal{R}$ and $\mathcal{S}$. We derive associativity of $\circ_K$ from that of ; as an example.

$$\begin{aligned}f \circ_K (g \circ_K h) &= \mathcal{S}\,(\mathcal{R}\,(f \circ_K (g \circ_K h)))\\ &= \mathcal{S}\,(\mathcal{R}\, f\,;(\mathcal{R}\, g\,;\mathcal{R}\, h))\\ &= \mathcal{S}\,((\mathcal{R}\, f\,;\mathcal{R}\, g)\,;\mathcal{R}\, h)\\ &= (\mathcal{S}\,(\mathcal{R}\, f) \circ_K \mathcal{S}\,(\mathcal{R}\, g)) \circ_K \mathcal{S}\,(\mathcal{R}\, h)\\ &= (f \circ_K g) \circ_K h.\end{aligned}$$

□

THEOREM 4.19. $(\mathsf{Sta}\, X, \eta{\downarrow_X}, \circ_K, +, \overline{(-)}, 0_X, \eta_X, {}^{*_K})$ *forms a* KAT.

PROOF. Straightforward from Theorem 4.18 using the isomorphisms $\mathcal{S}$ and $\mathcal{R}$ on subidentity relations and subidentity state transformers. □

We have now constructed two computationally important models of KAT: the relational $\mathsf{KAT}\ \mathsf{Rel}\, X$ and the state transformer $\mathsf{KAT}\ \mathsf{Sta}\, X$. Our programming intuitions for KAT are therefore standing on solid ground. It remains to relate state space X with program stores. But before that, in Chapter 6, we use KAT to derive inference rules for reasoning about while programs abstractly, and we will use them for generating structural verification conditions for imperative programs.

The benefits of algebra in mathematics and computer science are well known. One of them is that one can reason equationally about objects, which is easy to automate on a machine. Another one is that algebraic axioms may hold uniformly in many models of interest. Here, we could still argue that $\mathsf{Sta}\, X$ and $\mathsf{Rel}\, X$ are simply two isomorphic incarnations of the same underlying structure, namely nondeterministic functions. Yet there are nonisomorphic models of KAT that offer more detailed views of program execution in terms of program traces. These are particularly interesting for concurrent program executions, yet beyond the scope of these lecture notes. We only present a model of program traces briefly in the following and final section of this chapter.

4.3. Path Semantics

In more refined semantics of programs than $\mathsf{Rel}\, X$ or $\mathsf{Sta}\, X$ one may wish to consider execution sequences of programs in which information about the states a program visits and their state transitions alternates. This is in particular important for concurrent programs that interact and interfere with each other or for reactive systems, which are not even meant to terminate.

DEFINITION 4.20. A *directed graph* $G = (V, E, s, t)$ is formed by a set V of *vertices*, a set E of *edges* and a *source map* and a *target map* $s, t : E \to V$.

By definition, digraphs can have multiple edges between pairs of vertices as well as loops on vertices. A *path* in a digraph G is a sequence

$$\pi = (v_1, e_1, v_2, \ldots, v_{n-1}, e_{n-1}, v_n)$$

that starts and ends with an element of V and in which elements of V and E alternate. These must be compatible with source and target maps in the obvious way: $s(e_i) = v_i$ and $t(e_i) = v_{i+1}$ for $0 < i < n$ in the example above. We extend source and target maps from digraphs to paths in the obvious way. For instance, $s(\pi) = v_1$ and $t(\pi) = v_n$. In the context of programs, the set V represents the state space of a program and the set E the set of transitions between states. Path then correspond to possible execution traces of programs.

Composition of paths $\pi = (v_1, \ldots, e_{m-1}, v_m)$ and $\pi' = (v'_1, e'_1 \ldots, v'_n)$ is defined if $v_m = v'_1$, in which case it yields the path

$$\pi \cdot \pi' = (v_1, \ldots, v_{m-1}, e_{m-1}, v_m, e'_1, \ldots, v'_n).$$

It concatenates the two paths while identifying the vertices at the ends if possible. We write $\pi : v_1 \to v_2$ to indicate sources and targets of paths.

It is easy to check that path composition is associative whenever it is defined. Let $\pi_1 : v_1 \to v_2$, $\pi_2 : v_3 \to v_4$ and $\pi_3 : v_5 \to v_6$. Then $\pi_1 \cdot (\pi_2 \cdot \pi_3) = (\pi_1 \cdot \pi_2) \cdot \pi_3$

whenever $v_2 = v_3$ and $v_4 = v_5$. It is also easy to check that paths (v) of length 0 are identities of composition. For $\pi : v_1 \to v_2$, for example, $(v_1) \cdot \pi \cdot (v_2) = \pi$.

We write $\mathsf{Path}(G)$ for the set of all paths over digraph G and Id_G for the set of all zero-length paths.

PROPOSITION 4.21. *$(\mathcal{P}\,\mathsf{Path}(G), \cdot, \cup, \emptyset, Id_G)$ forms a Kleene algebra with, for all $X, Y \subseteq \mathsf{Path}(G)$,*

$$X \cdot Y = \{\pi \cdot \pi' \mid \pi \in X, \pi' \in Y \text{ and } \pi \cdot \pi' \text{ defined}\}, \qquad X^* = \bigcup_{i \geq 0} X^i.$$

Checking the axioms of Kleene algebra is left as an exercise.

Once again, the subidentities of paths, that is, the subsets of Id_G, form a boolean algebra. So let $Id_G{\downarrow} = \{X \subseteq \mathsf{Path}(G) \mid X \subseteq Id_G\}$. We can then state the main result of this section.

THEOREM 4.22. *$(\mathcal{P}\,\mathsf{Path}(G), Id_G{\downarrow})$ forms a KAT.*

REMARK 4.23. The path algebra over a digraph G can be seen as the free category generated by G, in which the objects are the vertices of G and the paths $\pi : v \to v'$ the morphisms. Details can be found in any textbook on category theory, but are beyond the scope of these lecture notes.

REMARK 4.24. We have already seen that languages form Kleene algebras. It is easy to check that very language model is isomorphic to a path model over a digraph with one vertex. For languages over alphabet Σ, take the digraph $(\{*\}, \Sigma, s, t)$, where $s, t : a \mapsto *$ for every $a \in \Sigma$. The paths over this digraph, which all $*$'s deleted, are nothing but Σ^*, the set of all words over Σ. Conversely, we can map words to path by injecting $*$ in the obvious way.

LEMMA 4.25. *Every path model $\mathcal{P}\,\mathsf{Path}(G)$ is isomorphic to a relational model on $\mathcal{P}\,(\mathsf{Path}(G) \times \mathsf{Path}(G))$ via the* Cayley map

$$c(X) = \{(\pi, \pi\pi') \mid \pi \in \mathsf{Path}(G) \text{ and } \pi' \in X\}, \quad \text{for } X \subseteq V.$$

PROOF. For $c(X \cdot Y) = c(X); c(Y)$, for instance,

$$\begin{aligned} c(X \cdot Y) &= \{(\pi_1, \pi_1\pi_2\pi_3) \mid \pi_1 \in \mathsf{Path}(G), \pi_2 \in X, \pi_3 \in Y\} \\ &= \{(\pi_1, \pi_1\pi_2) \mid \pi_1 \in \mathsf{Path}(G), \pi_2 \in X\}; \{(\pi_1\pi_2, \pi_1\pi_2\pi_3) \mid \pi_1 \in \mathsf{Path}(G), \pi_2 \in X, \pi_3 \in Y\} \\ &= \{(\pi_1, \pi_1\pi_2) \mid \pi_1 \in \mathsf{Path}(G), \pi_2 \in X\}; \{(\pi_4, \pi_4\pi_3) \mid \pi_4 \in \mathsf{Path}(G), \pi_3 \in Y\} \\ &= c(X); c(Y). \end{aligned}$$

For injectivity, if $X \neq Y$, suppose $\pi \in X - Y$, the other case being symmetric, and suppose e is a left unit of π, looping on the source vertex of π. Then $(e, x) \in h(X)$, but not in $h(Y)$. The relational model is then the image of $\mathcal{P}\,\mathsf{Path}(G)$ under c. □

Conversely, not every relational model is isomorphic to a path model. It is easy to see that $X^2 = Id_G \Rightarrow X = Id_G$ holds for all $X \in \mathsf{Path}(G)$, but the analogous property fails for the relation $R = \{(0, 1), (1, 0)\}$.

CHAPTER 5

Formalising the Two Semantics

Before formalising the relational and state transformer semantics with Isabelle and showing that they form models of KAT we prove two useful induction laws for powers in the context of class *dioid*. The second law follows from the first by opposition.

lemma *power-inductl*: $z + x \cdot y \leq y \Longrightarrow x \hat{} i \cdot z \leq y$
proof (*induct i*)
  case *0*
  have $x \hat{} 0 \cdot z = z$
    by *simp*
  also have $\ldots \leq y$
    using *0.prems local.add-lub* **by** *fastforce*
  finally show *?case*.
next
  case (*Suc i*)
  have $x \hat{} Suc\ i \cdot z = x \cdot x \hat{} i \cdot z$
    by *simp*
  also have $\ldots \leq x \cdot y$
    by (*simp add*: *Suc.hyps Suc.prems local.mult-assoc local.mult-isol*)
  also have $\ldots \leq y$
    using *Suc.prems local.add-lub* **by** *auto*
  finally show *?case*.
qed

lemma *power-inductr*: $z + y \cdot x \leq y \Longrightarrow z \cdot x \hat{} i \leq y$
  ⟨*Proof*⟩

5.1. Relational Semantics

We follow Section 4.1.2 and formalise the model relationship between relations and KAT with Isabelle in three steps, starting from dioids. First we add notation for relational composition.

notation *relcomp* (**infixl** ; *70*)

Showing that relations form dioids is straightforward and automatic because relations are well supported in Isabelle. We use an interpretation statement as Isabelle has no direct type for binary relations on a set X.

interpretation *rel-d*: *dioid* (∪) {} *Id* (;) (⊆) (⊂)
  by *unfold-locales auto*

Showing that relations form Kleene algebras requires a few technical lemmas, mainly because Isabelle's standard reflexive-transitive closure operation has not been defined as a union of powers. We need to relate it with the operation *powers* in the relation dioid *rel-d*, which comes from multiplicative monoids.

lemma *power-is-relpow*: *rel-d.power X i* = *X* ^^ *i*
⟨*Proof*⟩

Here, *rel-d.power* referes to the power operation for multiplicative monoids instantiated to relation dioids. The proof uses induction on i and poses no difficulty.

lemma *rel-star-def*: R^* = ($\bigcup i$. *rel-d.power R i*)
by (*simp add*: *power-is-relpow rtrancl-is-UN-relpow*)

In addition, the following distributivity or continuity laws are helpful for deriving the star induction laws in the relational model.

lemma *rel-star-contl*: *R* ; S^* = ($\bigcup i$. *R* ; *rel-d.power S i*)
proof−
have *R* ; S^* = *R* ; ($\bigcup i$. *rel-d.power S i*)
unfolding *rel-star-def* **by** *simp*
also have ... = ($\bigcup i$. *R* ; *rel-d.power S i*)
by (*simp add*: *relcomp-UNION-distrib*)
finally show *?thesis*.
qed

lemma *rel-star-contr*: R^* ; *S* = ($\bigcup i$. (*rel-d.power R i*) ; *S*)
by (*simp add*: *rel-star-def relcomp-UNION-distrib2*)

Their proofs use the more general continuity laws from Lemma 4.5. Next we show that relations form Kleene algebras.

interpretation *rel-ka*: *kleene-algebra* (∪) {} *Id* (;) (⊆) (⊂) *rtrancl*
proof *unfold-locales*
fix *x y z* :: 'a *rel*
show *Id* ∪ *x* ; x^* ⊆ x^*
by (*simp add*: *rel-star-unfoldl*)
show *Id* ∪ x^* ; *x* ⊆ x^*
by *fastforce*
show *z* ∪ *x* ; *y* ⊆ *y* ⟹ x^* ; *z* ⊆ *y*
by (*simp add*: *rel-star-inductl*)
show *z* ∪ *y* ; *x* ⊆ *y* ⟹ *z* ; x^* ⊆ *y*
by (*simp add*: *rel-star-inductr*)
qed

These depend on lemmas proving the unfold and induction axioms, which can be found in the Isabelle theories. Finally, it remains to define the antitest operation in the relational model.

definition *rel-atest* :: 'a *rel* ⇒ 'a *rel* (α_r) **where**

$\alpha_r\ R = Id \cap -R$

The interpretation statement showing that relations form KATs is then routine.

interpretation *rel-kat*: *kat* $(\cup)$ $\{\}$ *Id* $(;)$ $(\subseteq)$ $(\subset)$ *rtrancl* α_r
by *unfold-locales* (*auto simp*: *rel-atest-def*)

After this result, all laws proved about KAT within the context of class *kat* are available in the relational program semantics.

5.2. State Transformer Semantics

State transformers are not part of Isabelle's main libraries. We need to define their type and operations and do some background theory engineering. This development is not within the context of a type class or locale. First we introduce *'a sta* as an abbreviation for the function type *'a* $\Rightarrow$ *'a set*. Then we define the most important operations on state transformers.

type-synonym *'a sta* = *'a* $\Rightarrow$ *'a set*

abbreviation *eta* :: *'a sta* (η) **where**
$\eta\ x \equiv \{x\}$

abbreviation *nsta* :: *'a sta* (ν) **where**
$\nu\ x \equiv \{\}$

definition *kcomp* :: *'a sta* $\Rightarrow$ *'a sta* $\Rightarrow$ *'a sta* (**infixl** $\circ_K$ *75*) **where**
$(f \circ_K g)\ x = \bigcup \{g\ y\ |y.\ y \in f\ x\}$

definition *kadd* :: *'a sta* $\Rightarrow$ *'a sta* $\Rightarrow$ *'a sta* (**infixl** $+_K$ *65*) **where**
$(f +_K g)\ x = f\ x \cup g\ x$

definition *kleq* :: *'a sta* $\Rightarrow$ *'a sta* $\Rightarrow$ *bool* (**infix** $\sqsubseteq$ *50*) **where**
$f \sqsubseteq g = (\forall x.\ f\ x \subseteq g\ x)$

definition *kle* :: *'a sta* $\Rightarrow$ *'a sta* $\Rightarrow$ *bool* (**infix** $\sqsubset$ *50*) **where**
$f \sqsubset g = (f \sqsubseteq g \wedge f \neq g)$

Next we prove some helper lemmas for these operations.

lemma *sta-iff*: $((f::'a\ sta) = g) = (\forall x\ y.\ y \in f\ x \longleftrightarrow y \in g\ x)$
unfolding *fun-eq-iff* **by** *force*

lemma *kcomp-iff*: $y \in (f \circ_K g)\ x = (\exists z.\ y \in g\ z \wedge z \in f\ x)$
unfolding *kcomp-def* **by** *force*

lemma *kadd-iff*: $y \in (f +_K g)\ x = (y \in f\ x \vee y \in g\ x)$
unfolding *kadd-def* **by** *simp*

lemma *kleq-iff*: $f \sqsubseteq g = (\forall x\ y.\ y \in f\ x \longrightarrow y \in g\ x)$
unfolding *kleq-def* **by** *blast*

This makes the interpretation proof for dioids straightforward. But first we prove associativity of Kleisli composition as an example.

lemma *kcomp-assoc*: $(f \circ_K g) \circ_K h = f \circ_K (g \circ_K h)$
proof−
{fix x y
have $y \in ((f \circ_K g) \circ_K h)\ x = (\exists v.\ y \in h\ v \wedge (\exists w.\ v \in g\ w \wedge w \in f\ x))$
unfolding *kcomp-iff* **by** *simp*
also have $\ldots = (\exists w.\ (\exists v.\ y \in h\ v \wedge v \in g\ w) \wedge w \in f\ x)$
by *force*
also have $\ldots = (y \in (f \circ_K (g \circ_K h))\ x)$
unfolding *kcomp-iff* **by** *simp*
finally have $y \in ((f \circ_K g) \circ_K h)\ x = (y \in (f \circ_K (g \circ_K h))\ x)$**.}**
thus *?thesis*
by *force*
qed

The braces allow us to give a point-wise proof with element x first, and then generalise it to a point-free one.

interpretation *sta-monm*: *monoid-mult* η $(\circ_K)$
by *unfold-locales* (*transfer*, *force*)+

interpretation *sta-di*: *dioid* $(+_K)$ ν η $(\circ_K)$ $(\sqsubseteq)$ $(\sqsubset)$
by *unfold-locales* (*transfer*, *force*)+

The interpretation statement for multiplicative monoids brings once again powers in scope. It allows us to define the Kleene star for state transformers with respect to Kleisli composition.

abbreviation *kpow* ≡ *sta-monm.power*

definition *kstar* :: *'a sta* ⇒ *'a sta* **where**
kstar f x = $(\bigcup i.\ kpow\ f\ i\ x)$

Helper lemmas analogous to those of the relational model then allow us to show that state transformers form Kleene algebras. We do not show the details.

interpretation *sta-ka*: *kleene-algebra* $(+_K)$ ν η $(\circ_K)$ $(\sqsubseteq)$ $(\sqsubset)$ *kstar*
by *unfold-locales* (*transfer*, *auto simp*: *rel-star-inductl rel-star-inductr*)+

After defining the antitest operation and a helper lemma for it, the final interpretation proof for KAT is equally simple.

definition *sta-atest* :: *'a sta* ⇒ *'a sta* (α_s) **where**
$\alpha_s\ f\ x = \eta\ x - f\ x$

lemma *katest-iff*: $y \in \alpha_s\ f\ x \longleftrightarrow y \in \eta\ x \wedge \neg\ y \in f\ x$
unfolding *sta-atest-def* **by** *simp*

interpretation *sta-kat*: *kat* $(+_K)$ ν η $(\circ_K)$ $(\sqsubseteq)$ $(\sqsubset)$ *kstar* α_s
apply *unfold-locales*

unfolding *sta-iff katest-iff eta-def kcomp-iff kadd-iff nsta-def* **by** *auto*

It remains to mention that all interpretation proofs are incremental. In those for KAT, for instance, only the antitest axioms need to be checked relative those for Kleene algebra, if an interpretation proof for this class has already been given.

5.3. Isomorphism Between the Semantics

Setting up the isomorphisms from Section 4.2.2 with Isabelle is straightforward. Like the definitions for state transformers, this happens outside a type class or locale context. Manipulating functors, that is, functions that act on functions or relations poses no problem for Isabelle. However, Sledgehammer may struggles with such higher-order functions. Many proofs rely rather on *simp*, *auto*, *force* or *blast*; they require more user interaction and experience than previous ones. We only show the main definitions and a few example lemmas.

The two functors $\mathcal{S}$ and $\mathcal{R}$ are formalised as

definition *r2s* :: *'a rel* ⇒ *'a sta* ($\mathcal{S}$) **where**
$\mathcal{S}\ R = Image\ R \circ \eta$

definition *s2r* :: *'a sta* ⇒ *'a rel* ($\mathcal{R}$) **where**
$\mathcal{R}\ f = \{(x,y).\ y \in f\ x\}$

Next we show one of the star preservation laws in Lemma 4.15.

lemma *r2s-pow*: *rel-d.power* ($\mathcal{R}\ f$) $i = \mathcal{R}\ (kpow\ f\ i)$
by (*induct i*, *simp-all add*: *r2s-id r2s-comp*)

lemma *r2s-star*: $\mathcal{R}\ (kstar\ f) = (\mathcal{R}\ f)^*$
proof−
{fix *x y*
have $(x,y) \in \mathcal{R}\ (kstar\ f) = (\exists\, i.\ y \in kpow\ f\ i\ x)$
by (*simp add*: *kstar-def s2r-def*)
also have ... $= ((x,y) \in (\bigcup i.\ \mathcal{R}\ (kpow\ f\ i)))$
unfolding *s2r-def* **by** *simp*
also have ... $= ((x,y) \in (\bigcup i.$ *rel-d.power* ($\mathcal{R}\ f$) $i))$
using *r2s-pow* **by** *fastforce*
finally have $(x,y) \in \mathcal{R}\ (kstar\ f) = ((x,y) \in (\mathcal{R}\ f)^*)$
using *rel-star-def* **by** *blast***}**
thus *?thesis*
by *auto*
qed

The Isar proof has been translated more or less one-to-one from the stepwise proof on paper Chapter 4.

Taken together these Isabelle proofs illustrate the impressive capability of translating non-trivial mathematical proofs essentially one-to-one into an interactive theorem prover. And beyond that, many of these proofs can even be fully automated.

CHAPTER 6

Propositional Hoare Logic

After the mathematical ground work of the previous chapters we start to develop our first formalism for program verification—an algebraic variant of *Hoare logic*. First we derive the inference rules of an algebraic variant of *propositional Hoare logic*, which continues to disregard the internal structure of tests and basic commands, and therefore lacks a rule for variable assignments. The main use of propositional Hoare logic is the generation of algebraic verification conditions. This is achieved in a recursive way with respect to the program constructs modelled by KAT expressions. It can be fully automated using Isabelle tactics.

6.1. Partial Correctness Specifications

A program is *partially correct* with respect to a precondition and a postcondition if, whenever the program is assumed to start in states that satisfy the precondition and to terminate, then it terminates in states that satisfy the postcondition (a program is *totally correct* if it is partially correct and all loops terminate).

In KAT we assume that all assertions, including preconditions, postconditions and tests, are modelled within its boolean algebra of tests. Then px expresses that program x executes from states where precondition p holds. By opposition, xq expresses that program x terminates in states where postcondition q holds. The above partial correctness specification can thus be formalised in KAT as

$$px \leq xq.$$

It is equivalent to $px\overline{q} = 0$ by Lemma 2.31, which means that if program x executes from states where precondition p holds and is assumed to terminate, then it will never end up in states where postcondition q fails.

In the tradition of Hoare logic we use *Hoare triples* to express such partial correctness specifications for programms, defining

$$\mathsf{H}\, p\, x\, q \Leftrightarrow px \leq xq\,.$$

Alternatively, we may view this equivalence as an algebraic semantics to Hoare triples in KAT. The simple identity in its right-hand side allows us to reason equationally about program correctness. Its adequacy for partial correctness is can be confirmed in relation and state transformer KAT.

Lemma 6.1.

(1) *In* $\mathsf{Rel}\, X$*, for all* $R \in \mathsf{Rel}\, X$ *and* $P, Q \in \mathsf{Id}{\downarrow_x}$*,*

$$\mathsf{H}\, P\, R\, Q \;\Leftrightarrow\; (\forall a, b \in X.\; P\, a \wedge (a, b) \in R \Rightarrow Q\, b)\,,$$

(2) *In* $\mathsf{Sta}\, X$*, for all* $f \in \mathsf{Sta}\, X$ *and* $P, Q \in \eta{\downarrow_X}$*,*

$$\mathsf{H}\, P\, f\, Q \;\Leftrightarrow\; (\forall a, b \in X.\; P\, a \wedge b \in f\, a \Rightarrow Q\, b)\,.$$

Both formulas clearly express partial correctness.

6.2. Rules of Propositional Hoare Logic

We can now derive the following facts, which correspond to the standard structural rules of Hoare logic.

THEOREM 6.2. *Let* (K, B) *be a* KAT. *For all* $x, y \in K$ *and* $p, p', q, q', t \in B$, *the following rules of* propositional Hoare logic *(*PHL*) are derivable:*

(H-skip) $$\mathsf{H}\, p\, 1\, p\,,$$

(H-cons) $$p \leq p' \wedge \mathsf{H}\, p'\, x\, q' \wedge q' \leq q \Rightarrow \mathsf{H}\, p\, x\, q\,,$$

(H-seq) $$\mathsf{H}\, p\, x\, r \wedge \mathsf{H}\, r\, y\, q \Rightarrow \mathsf{H}\, p\, (xy)\, q\,,$$

(H-cond) $$\mathsf{H}\, (tp)\, x\, q \wedge \mathsf{H}\, (\bar{t}p)\, y\, q \Rightarrow \mathsf{H}\, p\, (\textbf{if } t \textbf{ then } x \textbf{ else } y)\, q\,,$$

(H-while) $$\mathsf{H}\, (tp)\, x\, p \Rightarrow \mathsf{H}\, p\, (\textbf{while } t \textbf{ do } x)\, (\bar{t}p)\,.$$

PROOF. We verify (H-while) as an example.

$$\begin{aligned} \mathsf{H}\, (tp)\, x\, p &\Rightarrow ptx \leq txp \\ &\Rightarrow p(tx)^* \leq (tx)^* p \\ &\Rightarrow p(tx)^* \bar{t} \leq (tx)^* p\bar{t} \\ &\Leftrightarrow \mathsf{H}\, p\, (\textbf{while } t \textbf{ do } x)\, (\bar{t}p)\,. \end{aligned}$$

The first step unfolds the definition of Hoare triples and applies monotonicity as well as some boolean laws for tests; the second one applies a simulation law from Lemma 2.16. The third one uses order-preservation of composition and the fourth one the definition of while loops and Hoare triples. □

We now explain these inference rules in detail.

- The *skip rule* (H-skip) states that any assertion continues to hold if a program does nothing.
- The *consequence rule* (H-cons) allows weakening preconditions and strengthening postconditions.

Rules (H-seq)-(H-while) are compositional with respect to the program structure. They derive partial correctness specifications of complex programs from those for simpler ones.

- The *sequential composition rule* (H-seq) states that for proving a sequential composition partially correct it suffices to prove its components partially correct, using an assertion r as a postcondition for the first component and a precondition for the second one.
- The *conditional rule* (H-cond) states that for proving a conditional partially correct it suffices to prove its two branches partially correct—the first when the test holds and the second when it doesn't.
- The *while rule* (H-while) states that for proving a loop partially correct, the postcondition must be the meet of the precondition p and the complement of test t of the loop. It then suffices to prove partial correctness of the body of the loop with precondition tp and postcondition p.

PHL offers one structural rule per program construct in (H-seq)-(H-while). This makes their application deterministic.

The rules (H-cons) and (H-seq) introduce new assertions when applied backwards. In fact, the whole point of (H-cons) is to rewrite pre- and postconditions in proofs so that parts of proofs can be composed. Finding suitable intermediate assertions in (H-seq) may be non-trivial. We return to this issue later. The rule (H-while) is more restrictive than the other rules because p must appear in all pre- and postconditions. Formally, an element $i \in B$ of a KAT (K, B) is an *invariant* of program $x \in K$ if

$$ix \leq xi.$$

Thus $\mathsf{H}\, i\, x\, i$ holds if i is an invariant for x. Reasoning with while loops in PHL therefore requires finding *loop invariants*. This can be hard in practice as well.

In addition, (H-cons) must usually be invoked in combination with (H-while) to link invariants with the pre- and postconditions of correctness specifications. This can be internalised by deriving a macro-rule for while loops annotated by invariants in KAT. First, a general purpose invariant law

$$p \leq i \wedge \mathsf{H}\, i\, x\, i \wedge i \leq q \Rightarrow \mathsf{H}\, p\, x\, q$$

is derivable in KAT using (2). Next we provide notation for annotating while loops with invariants:

$$\textbf{while } p \textbf{ inv } i \textbf{ do } x = \textbf{while } p \textbf{ do } x.$$

Operationally, adding this invariant does not change the semantics of the while rule, yet it triggers the following macro inference rule.

LEMMA 6.3. *Let (K, B) be a* KAT. *Then, for all $x \in K$ and $p, q, i, t \in B$,*

(H-while-inv) $$p \leq i \wedge \mathsf{H}\,(it)\, x\, i \wedge \bar{t}i \leq q \Rightarrow \mathsf{H}\, p\, (\textbf{while } t \textbf{ inv } i \textbf{ do } x)\, (q\bar{t}).$$

PROOF. Combine (H-while) with the rule for reasoning with invariants. □

Inference rule (H-while-inv) captures precisely the approach for reasoning with simple while-loops outlined in Chapter 1: Show that

- the precondition implies the invariant;
- the invariant is preserved by the body of the loop when its test is true;
- the invariant implies the postcondition when the test is false.

By Theorem 4.9 and 4.19, the rules of PHL hold a fortiori in the relational and state transformer semantics. By deriving the rules of PHL in KAT and proving that binary relations and state transformers form KATs in Theorem 4.9 and 4.19, we have thus proved PHL *sound* with respect to these two program semantics.

6.3. Formalising Propositional Hoare Logic

We now outline the Isabelle formalisation of PHL. We also show how more specific PHL rules can be derived in the relational and state transformer semantics, where subidentities are replaced by predicates ranging over unstructured state spaces. This enhances reasoning with Isabelle.

6.3.1. PHL in KAT. First we define Hoare triples and while loops decorated with invariants.

```
definition Ho :: 'a ⇒ 'a ⇒ 'a ⇒ bool where
  Ho p x q = (τ p · x ≤ x · τ q)

definition while-inv :: 'a ⇒ 'a ⇒ 'a ⇒ 'a (while - inv - do - od [64,64,64] 63) where
```

while p inv i do x od = while p do x od

The derivation of the inference rules of PHL within KAT in Isabelle can then be left as an exercise. For program verification with Isabelle, the following variant of the conditional rule is interesting.

lemma *H-cond-iff*:
Ho p (if r then x else y fi) q = (Ho (τ p · τ r) x q ∧ Ho (τ p · α r) y q)
⟨*Proof*⟩

6.3.2. Specialised PHL in Relation KAT. In relation and state transformer KAT we can derive more specialised inference rules that work with predicates instead of subidentity relations or subidentity state transformers. These are beneficial for Isabelle, as we cannot simply identify subidentity relations or state transformers with predicates in a proof assistant—we need to make the isomorphisms between subidentities and predicates explicit. We start with relation KAT. First we introduce a type for predicates that depend on one single parameter. This parameter will later be instantiated to program stores. Then we introduce an explicit type coercion function that formalises the isomorphism between predicates and subidentity relations. Predicates are boolean-valued functions in Isabelle.

type-synonym *'a pred = 'a ⇒ bool*

abbreviation *p2r :: 'a pred ⇒ 'a rel* (⌈-⌉$_r$) **where**
⌈*P*⌉$_r$ ≡ {(*s*,*s*) |*s*. *P s*}

After proving some helper lemmas that do the right kind of magic to translate subidentities to predicates in verification proofs, we can derive specialised structural rules of Hoare logic based on predicate logic.

abbreviation *rH :: 'a pred ⇒ 'a rel ⇒ 'a pred ⇒ bool* (H_r) **where**
H_r *P R Q* ≡ *rel-kat.H* ⌈*P*⌉$_r$ *R* ⌈*Q*⌉$_r$

abbreviation
rcond :: 'a pred ⇒ 'a rel ⇒ 'a rel ⇒ 'a rel (*rif - then - else - fi* [*64*,*64*,*64*] *63*) **where**
rif P then R else S fi ≡ *rel-kat.cond* ⌈*P*⌉$_r$ *R S*

abbreviation
rwhile-inv :: 'a pred ⇒ 'a pred ⇒ 'a rel ⇒ 'a rel
(*rwhile - inv - do - od* [*64*,*64*,*64*] *63*) **where**
rwhile P inv I do R od ≡ *rel-kat.while-inv* ⌈*P*⌉$_r$ ⌈*I*⌉$_r$ *R*

lemma *rH-unfold*: H_r *P R Q* = (∀ *x y*. *P x* ⟶ (*x*,*y*) ∈ *R* ⟶ *Q y*)
⟨*Proof*⟩

lemma *rH-skip*: H_r *P Id Q* = (∀ *x*. *P x* ⟶ *Q x*)
⟨*Proof*⟩

lemma *rH-cons1*:
assumes H_r *P' R Q*
and ∀ *x*. *P x* ⟶ *P' x*

shows $H_r\ P\ R\ Q$
⟨*Proof*⟩

lemma *rH-cons2*:
assumes $H_r\ P\ R\ Q'$
and $\forall\, x.\ Q'\ x \longrightarrow Q\ x$
shows $H_r\ P\ R\ Q$
⟨*Proof*⟩

lemma *rH-cons*:
assumes $H_r\ P'\ R\ Q'$
and $\forall\, x.\ P\ x \longrightarrow P'\ x$
and $\forall\, x.\ Q'\ x \longrightarrow Q\ x$
shows $H_r\ P\ R\ Q$
⟨*Proof*⟩

lemma *rH-cond* [*simp*]:
$(H_r\ P\ (\mathit{rif}\ T\ \mathit{then}\ R\ \mathit{else}\ S\ \mathit{fi})\ Q)$
$\quad = (H_r\ (\lambda s.\ P\ s \wedge T\ s)\ R\ Q \wedge H_r\ (\lambda\ s.\ P\ s \wedge \neg T\ s)\ S\ Q)$
⟨*Proof*⟩

lemma *rH-while-inv*:
assumes $H_r\ (\lambda s.\ I\ s \wedge T\ s)\ R\ I$
and $\forall\, x.\ P\ x \longrightarrow I\ x$
and $\forall\, x.\ I\ x \wedge \neg\ T\ x \longrightarrow Q\ x$
shows $H_r\ P\ (\mathit{rwhile}\ T\ \mathit{inv}\ I\ \mathit{do}\ R\ \mathit{od})\ Q$
⟨*Proof*⟩

In all these rules, subidentity relations have been replaced by predicates, which are more amenable to automated reasoning with Isabelle. Alternatively we could have replaced subidentities with sets. Under the hood, making such replacements smooth requires setting up some simplification rules for $\lceil - \rceil_r$. This can be quite subtle; details can be found in our Isabelle theories.

6.3.3. Specialiced PHL in State Transformer KAT. The development in state transformer KAT parallels that of relation KAT.

abbreviation *p2s* :: *'a pred* ⇒ *'a sta* ($\lceil\text{-}\rceil_s$) **where**
$\lceil P \rceil_s\ x \equiv \mathit{if}\ P\ x\ \mathit{then}\ \{x\}\ \mathit{else}\ \{\}$

After proving some helper lemmas, we specialise Hoare triples and the definitions of conditionals and while-loops with invariants.

abbreviation *Hs* :: *'a pred* ⇒ *'a sta* ⇒ *'a pred* ⇒ *bool* (H_s) **where**
$H_s\ P\ f\ Q \equiv \mathit{sta\text{-}kat.H}\ \lceil P \rceil_s\ f\ \lceil Q \rceil_s$

abbreviation
scond :: *'a pred* ⇒ *'a sta* ⇒ *'a sta* ⇒ *'a sta* (*sif - then - else - fi* [*64*,*64*,*64*] *63*) **where**
sif P then f else g fi ≡ *sta-kat.cond* $\lceil P \rceil_s\ f\ g$

abbreviation
swhile-inv :: *'a pred* ⇒ *'a pred* ⇒ *'a sta* ⇒ *'a sta*

(*swhile - inv - do - od* [*64*,*64*,*64*] *63*) **where**
swhile P inv I do f od ≡ *sta-kat.while-inv* ⌈*P*⌉$_s$ ⌈*I*⌉$_s$ *f*

We can then derive specialised structural rules for Hoare logic that have the same shape as those for relations. They differ only with respect to typing and syntax indicating state transformers instead of relations. It thus remains to model the program store, integrate it into the relational and state transformer semantics, define the relational and state transformer semantics of assignment commands and derive assignment rules for Hoare logic in these concrete program semantics.

CHAPTER 7

Hoare Logic

We now integrate a simple mathematical model of the program store into the relational and state transformer semantics, instantiating the unstructured state space X of relations or state transformers to a structured state space of program stores. We define the semantics of variable assignments in the relational and state transformer semantics over the program store and derive semantic variants of the standard assignment rules of Hoare logic. This completes the derivation of semantic variants of the rules of Hoare logic and allows us to verify simple while programs in languages like Imp.

Our semantic approach is not a Hoare logic in the strict sense. We do not work with a program syntax for command, tests and expressions and hence not with a symbolic logic. Instead we consider programs as actions on a program store like in a discrete dynamical system. Compared to classical Hoare logic, our approach is more detailed and precise, but also more verbose. At the end of this section, we present a more standard style of program analysis as a semi-formal notational simplification of the semantic approach.

7.1. Semantics of the Program Store

There are many ways of modelling the program store, from simple to complex and from abstract to concrete. We pick one of the simplest models.

7.1.1. Program Store. We model program stores as functions $s : V \to D$ from variables in the set V to values in the semantic domain D. Semantic domains could be the integers, strings, or any other kind of data. We write $S = D^V$ for the set of program stores, where D^V stands for the set of functions of type $V \to D$.

In simple while-languages like Imp, the store dynamics is generated by assignment commands $x := e$, which assign values of expressions e, taken in the program store, to program variables x. Expressions are part of the program syntax. They can be defined by a grammar and evaluated in the program store by a function of type $E \to S \to D$, which maps expressions of type E and stores of type S to values of type D. In our semantic approach, we use functions of type $S \to D$ instead. We explain the difference by example.

Example 7.1. In languages like Imp, simple arithmetic expressions are typically defined by a syntax similar to

$$E ::= v \mid n \mid E + E \mid E \cdot E \,,$$

where $v \in V$ and n stands for a *numeral*, a string of decimal digits $0, \ldots, 9$, denoting a natural number $n_{\mathbb{N}}$. Such numerals can be defined by another grammar. Examples of arithmetic expressions are v, 42, $2 + 6$ or $3 \cdot v + 9$.

Arithmetic expressions are evaluated in semantic domains such as the natural numbers, $D = \mathbb{N}$. Program stores then specialise to functions $s : V \to \mathbb{N}$; they parametrise the evaluation function $[\![-]\!]_{(-)} : E \to S \to \mathbb{N}$, which can be defined recursively as

$$\begin{aligned} [\![v]\!]_s &= s\, v\,, \\ [\![n]\!]_s &= n_{\mathbb{N}}\,, \\ [\![e_1 + e_2]\!]_s &= [\![e_1]\!]_s +_{\mathbb{N}} [\![e_2]\!]_s\,, \\ [\![e_1 \cdot e_2]\!]_s &= [\![e_1]\!]_s \cdot_{\mathbb{N}} [\![e_2]\!]_s\,. \end{aligned}$$

Here, we write somewhat tediously $+_{\mathbb{N}}$, $\cdot_{\mathbb{N}}$ and likewise to distinguish the semantic data of numbers and operations on them from the syntactic data of variables, constant symbols and function symbols such as $+$ and $\cdot$.

In a store s in which $s : v \mapsto 11_{\mathbb{N}}$, expression $3 \cdot v + 9$ thus evaluates to

$$\begin{aligned} [\![3 \cdot v + 9]\!]_s &= [\![3 \cdot v]\!]_s +_{\mathbb{N}} [\![9]\!]_s \\ &= [\![3]\!]_s \cdot_{\mathbb{N}} [\![v]\!]_s +_{\mathbb{N}} 9_{\mathbb{N}} \\ &= 3_{\mathbb{N}} \cdot_{\mathbb{N}} s\, v +_{\mathbb{N}} 9_{\mathbb{N}} \\ &= 3_{\mathbb{N}} \cdot_{\mathbb{N}} 11_{\mathbb{N}} +_{\mathbb{N}} 9_{\mathbb{N}} \\ &= 42_{\mathbb{N}}\,. \end{aligned}$$

In our semantic approach, we use functions $S \to \mathbb{N}$ instead. Obviously, for every expression e of type E and evaluation function $E \to S \to \mathbb{N}$, the function $\lambda s.\ [\![e]\!]_s$ has this type. Evaluating e in store s thus yields the same natural number as applying $\lambda s.\ [\![e]\!]_s$ to s. Instead of expression $3 \cdot v + 9$, for instance, we use the function $\lambda s.\ [\![3 \cdot v + 9]\!]_s = \lambda s.\ 3_{\mathbb{N}} \cdot_{\mathbb{N}} s\, v +_{\mathbb{N}} 9_{\mathbb{N}}$. It is already evaluated to the level of variables. Applying it to store $s : v \mapsto 11_{\mathbb{N}}$ yields

$$(\lambda s'.\ 3_{\mathbb{N}} \cdot_{\mathbb{N}} s'\, v +_{\mathbb{N}} 9_{\mathbb{N}})\, s = 3_{\mathbb{N}} \cdot_{\mathbb{N}} s\, v +_{\mathbb{N}} 9_{\mathbb{N}} = 3_{\mathbb{N}} \cdot_{\mathbb{N}} 11_{\mathbb{N}} +_{\mathbb{N}} 9_{\mathbb{N}} = 42_{\mathbb{N}}\,,$$

as expected. □

7.1.2. Store Updates. Variable assignments $v := e$ form the basic commands in simple while languages like Imp. In our semantic approach, a variable assignment $v := e$, with $v \in V$ and $e : S \to D$, acts on a program store $s \in S$ as follows: the value $e\, s \in D$ is computed and then used to construct a new store s', which is the same as s except that now $s' : v \mapsto e\, s$.

Store updates are therefore functions of type $V \to D \to S \to S$ that take a variable, a value (such as $e\, s$) and a store and yield another store. As stores are themselves functions, store update functions are higher-order functions that act on functions by updating one of their arguments. They can of course be defined for functions of arbitrary type $X \to Y$.

The *function update* function $\Delta : X \to Y \to Y^X \to Y^X$ that updates functions of type $X \to Y$ in an argument of type X by a value of type Y is defined as

$$(\Delta\, x\, y\, f)\, x' = \begin{cases} y & \text{if } x = x', \\ f\, x' & \text{if } x \neq x'. \end{cases}$$

It takes elements $x \in X$ and $y \in Y$ and a function $f : X \to Y$ and yields the function $f' : X \to Y$ that maps x to y and every other $x' \in X$ to $f\, x'$.

Program verification requires calculating with Δ. The following properties may simplify such proofs.

LEMMA 7.2. *For all* $x, y \in X$, $a, b \in Y$ *and* $f : X \to Y$,

(1) $(\Delta\, x\, a\, f)\, x = a$,
(2) $(\Delta\, x\, a\, f)\, y = f\, y$ *for* $x \neq y$,
(3) $\Delta\, x\, a \circ \Delta\, x\, b = \Delta\, x\, a$,
(4) $\Delta\, x\, a \circ \Delta\, y\, b = \Delta\, y\, b \circ \Delta\, x\, a$ *for* $x \neq y$,
(5) $\Delta\, x\, (f\, x)\, f = f$.

PROOF. Exercise. □

Items (1) and (2) correspond to the two cases in the definition of Δ. By (3), the last update in a sequence of consecutive updates to the same variable overwrites all previous updates. By (4), updates to different variables are independent; they can be performed in any order. Finally, by (5), an update of a variable to a value it already has does not change a function.

We define the following variant of Δ for convenience. Let $S = V \to D$ be a set of program stores. A *store update function* is a function $set : V \to (S \to D) \to S \to S$ such that, for all $v \in V$. $e : S \to V$ and $s \in S$,

$$set\, v\, e\, s = \Delta\, v\, (e\, s)\, s.$$

Store update functions thus act on the store by changing the value of a variable to a value that depends on the previous store.

EXAMPLE 7.3. For function $e : S \to \mathbb{N}$ defined by $e = (\lambda s'.\ 3_{\mathbb{N}} \cdot_{\mathbb{N}} s'\, v +_{\mathbb{N}} 9_{\mathbb{N}})$ and store s that maps v to $11_{\mathbb{N}}$, the updated store

$$s' = set\, v\, e\, s = \Delta\, v\, (e\, s)\, s = \Delta\, v\, 42_{\mathbb{N}}\, s$$

maps v to $42_{\mathbb{N}}$ and every other variable v' to $s\, v'$. □

7.2. Semantics of Assignment Commands

We can now define the semantics of assignment commands in the relational and state transformer semantics.

In $\mathsf{Sta}\, S$ we define, for $v \in V$, $e : S \to V$ and $s \in S$,

$$(v := e) = \eta \circ (set\, v\, e).$$

In other words, $(v := e)\, s = \{set\, v\, e\, s\}$. An assignment $(v := e) : S \to \mathcal{P}S$ is thus simply a store update function lifted to a state transformer by decorating it with braces.

In $\mathsf{Rel}\, S$, we calculate

$$\mathcal{R}\, (v := e) = \{(s, \Delta\, v\, (e\, s)\, s) \mid s \in S\}\,.$$

We henceforth write $v := e$ uniformly for both semantics.

7.3. Assignment Rules of Hoare Logic

With the semantics of assignment commands in place it is now straightforward to derive inference rules for them in the style of Hoare logic in the concrete relational and state transformer semantics of the program store. These are uniform and we can therefore drop indices in the following proposition.

PROPOSITION 7.4. *Let P and Q be in $\eta{\downarrow_S}$ or in $\mathsf{Id}{\downarrow_S}$, and let $v \in V$ and $e \in S \to D$. The following* assignment rules *are derivable in* $\mathsf{Sta}\, S$ *or* $\mathsf{Rel}\, S$.

(H-assign-iff) $\quad \mathsf{H}\, P\, (v := e)\, Q \Leftrightarrow (\forall s \in S.\ P\, s \Rightarrow Q\, (set\, v\, e\, s))$,

(H-assign) $\quad \mathsf{H}\, (Q \circ (set\, v\, e))\, (v := e)\, Q$,

(H-assign-floyd) $\quad \mathsf{H}\, P\, (v := e)\, (\lambda s.\ \exists w \in D.\ s\, v = e\, (\Delta\, v\, w\, s) \wedge P\, (\Delta\, v\, w\, s))$.

PROOF.

(1) In $\mathsf{Sta}\, S$,

$$\begin{aligned}\mathsf{H}\, P\, (v := e)\, Q &\Leftrightarrow (\forall s, s' \in S.\ P\, s \wedge s' \in \{set\, v\, e\, s\} \Rightarrow Q\, s')\\ &\Leftrightarrow (\forall s, s' \in S.\ P\, s \wedge s' = set\, v\, e\, s \Rightarrow Q\, s')\\ &\Leftrightarrow (\forall s \in S.\ P\, s \Rightarrow Q\, (set\, v\, e\, s)).\end{aligned}$$

The proof in $\mathsf{Rel}\, S$ is almost identical.

(2) This is a special case of (H-assign-iff) with $P = Q \circ (set\, v\, e)$.

(3) We calculate

$$\begin{aligned}P\, s &\Leftrightarrow e\, s = e\, s \wedge P\, s\\ &\Leftrightarrow e\, s = e\, (\Delta\, v\, (s\, v)\, s) \wedge P\, (\Delta\, v\, (s\, v)\, s)\\ &\Rightarrow \exists w.\ e\, s = e\, (\Delta\, v\, w\, s) \wedge P\, (\Delta\, v\, w\, s)\\ &\Leftrightarrow \exists w.\ (set\, v\, e\, s)\, v = e\, (\Delta\, v\, w\, (set\, v\, e\, s)) \wedge P\, (\Delta\, v\, w\, (set\, v\, e\, s))\\ &\Leftrightarrow (\lambda s'.\ \exists w.\ s'\, v = e\, (\Delta\, v\, w\, s') \wedge P\, (\Delta\, v\, w\, s'))(set\, v\, e\, s).\end{aligned}$$

The second step uses Lemma 7.2(2), the fourth one Lemma 7.2(3). The last step uses β-reduction. The claim then follows by (1). □

Rule (H-assign) is a semantic variant of *Hoare's assignment rule*, (H-assign-floyd) a semantic variant of *Floyd's assignment rule*. Hoare's rule and (H-assign-iff) are backward rules because assignments affect the preconditions of Hoare triples. The typical workflow with these rules is from the postcondition towards the precondition of a composite program. Floyd's rule, by contrast, is a forward assignment rule that affects the postcondition. It supports a workflow from preconditions towards postconditions and thus the symbolic execution of programs from initial values. In (H-assign-floyd), the variable w represents the value in D of variable v prior to execution. The postcondition then states that the value of v in store s after the execution of the assignment is equal to the value of e in the store prior to execution, and that the precondition P (in the prestate) remains true.

The derivation of the assignment rules finishes the derivation of the rules of Hoare logic, more precisely that of their semantic variants. The rules of PHL were derived in KAT, yet hold in the relational and state transformer semantics over the program store. The assignment rules have been derived in these concrete program semantics. Hence the verification of programs with this approach is ultimately performed in the concrete models of program execution.

EXAMPLE 7.5. Consider the predicate $P_{\mathbb{N}} = \lambda s.\ 3_{\mathbb{N}} \cdot_{\mathbb{N}} s\, v + 9_{\mathbb{N}} = 42_{\mathbb{N}}$ and let s be a store in which variable v has value $7_{\mathbb{N}}$. Then

$$P_{\mathbb{N}}\, (\Delta\, v\, 7_{\mathbb{N}}\, s) \Leftrightarrow 3_{\mathbb{N}} \cdot_{\mathbb{N}} (\Delta\, v\, 7_{\mathbb{N}}\, s)\, v) +_{\mathbb{N}} 9_{\mathbb{N}} = 42_{\mathbb{N}} \Leftrightarrow 3_{\mathbb{N}} \cdot_{\mathbb{N}} 7_{\mathbb{N}} +_{\mathbb{N}} 9_{\mathbb{N}} = 42_{\mathbb{N}},$$

which reduces further to false. □

7.4. Formalising the Program Store and Hoare Logic

The Isabelle formalisation of the program store and Hoare logic follows the mathematical development closely. We only show the main definitions and facts. We formalise Δ and *set* as

definition *fup* :: $'a \Rightarrow 'b \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$ **where**
fup x a f = $(\lambda y.$ *if x* = *y then a else f y*)

abbreviation *set* :: $'a \Rightarrow (('a \Rightarrow 'b) \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$ **where**
set v e s ≡ *fup v* (*e s*) *s*

We have also verified the identities in Lemma 7.2 with Isabelle. We introduce a type synonym as notation for the program store.

type-synonym $'a$ *store* = *string* $\Rightarrow$ $'a$

Using strings as a type for variables allows us to use Isabelle's built-in equality for strings (two strings are equal if they consist of the same letters). Otherwise we would have to declare explicitly for each pair of variables denoted by different symbols that they are different.

While this is important when verifying individual programs, we can formalise the semantics of assignment commands and derive the rules of Hoare logic for arbitrary variable types.

definition
rel-assign :: $'a \Rightarrow (('a \Rightarrow 'b) \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$ *rel* (- :=$_r$ - [*70*, *65*] *61*) **where**
v :=$_r$ *e* = {(*s*, *set v e s*) |*s*. *True*}

lemma *rel-assign-iff*: ((*s*,*s'*) ∈ *v* :=$_r$ *e*) = (*s'* = *set v e s*)
by (*simp add*: *rel-assign-def*)

definition
sta-assign :: $'a \Rightarrow (('a \Rightarrow 'b) \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$ *sta* (- :=$_s$ - [*70*, *65*] *61*) **where**
v :=$_s$ *e* = $\eta \circ$ (*set v e*)

lemma *sta-assign-iff*: (*s'* ∈ (*v* :=$_s$ *e*) *s*) = (*s'* = *set v e s*)
by (*simp add*: *eta-def sta-assign-def*)

The assignment rules (H-assign-iff), (H-assign) and (H-assign-floyd) of Hoare logic are then derivable in both semantics. The proofs are simple and uniform, only a renaming of symbols in formulas and lemmas in proofs is required to move between the relational and the state transformer model. We only show the variants for state transformers, the relational variants are identical up to indices.

lemma *sH-assign-iff* [*simp*]: *H*$_s$ *P* (*v* :=$_s$ *e*) *Q* = ($\forall$ *s*. *P s* $\longrightarrow$ *Q* (*set v e s*))
by (*simp add*: *sH-unfold sta-assign-iff*)

lemma *sH-assign*: *H*$_s$ (*P* $\circ$ (*set v e*)) (*v* :=$_s$ *e*) *P*
by *simp*

lemma *sH-assign-floyd*: *H*$_s$ *P* (*v* :=$_s$ *e*) (λ*s*. $\exists$ *w*. *s v* = *e* (*set v w s*) $\wedge$ *P* (*set v w s*))

by (*simp*, *metis fup-simp1 fup-triv*)

Finally, we supply Isabelle notation for partial correctness specifications.

abbreviation
sH-sugar :: *'a pred* ⇒ *'a sta* ⇒ *'a pred* ⇒ *bool* (*sPRE - - POST - [64,64,64] 63*) **where**
sPRE P f POST Q ≡ H_s *P f Q*

We can now start verifying programs with Isabelle. Due to our previous interpretation statements with Isabelle, binary relations and state transformers form KATs, hence the inference rules of PHL hold in these models. This is enabled by the polymorphism of Isabelle's type classes, where KATs have type α, so that the two instances have type α *rel* and α *sta*. They are then instantiated further to α *store rel* and α *store sta*, and combined with the assignment laws to full fledged Hoare logics, or rather semantic variants thereof. Isabelle can pick up facts from all levels and instantiate them appropriately.

Remark 7.6. The semantic approach outlined works for any data domain supported by Isabelle. Programs are thus modelled entirely using functions, relations, predicates and data types provided by Isabelle. Such an approach is called a *shallow embedding* of Imp-style programs in Isabelle. A *deep embedding*, by contrast, would start with definitions of data types for expressions, tests and program commands in Isabelle. In the formalised mathematics and proof assistants communities, shallow embeddings are advocated because of their simplicity. In particular there is no need to specify grammars and semantic maps (evaluation or interpretation functions), which can be tedious and rather repetitive. For more complicated verification tasks, for instance those that depend on specific properties of program variables, deep embeddings may offer advantages, but these are beyond the scope of these lectures.

7.5. Examples: Program Verification with Hoare Logic

We express partial correctness specifications directly in the concrete semantics of programs. Our specifications and encodings of programs are therefore more detailed than those in typical textbooks on Hoare logic and they capture the dynamics of program stores more precisely.

Example 7.7. A typical textbook partial correctness specification for the integer division algorithm in a language like Imp might look like

$$
\begin{array}{l}
PRE\ 0 < y \\
q := 0; \\
r := x; \\
\textbf{while}\ y \leq r\ \textbf{inv}\ x = q \cdot y + r\ \textbf{do} \\
\quad q := q + 1; \\
\quad r := r - y \\
POST\ x = q \cdot y + r \wedge r < y
\end{array}
$$

We need to interpret this syntactic statement in our minds to make it meaningful, that is, translate it into our formal semantics. We need to interpret $0 < y$, for instance, to mean that the value of y in some program store equals the number 0.

In our formal semantics, we would have to write $\lambda s.\ 0 < s\, y$. Likewise, $q := q + 1$ means that program variable q is assigned the value of $q + 1$ in some program store. Formally, therefore, $q := (\lambda s.\ s\, q + 1)$. The following Isabelle code shows this difference.

```
rPRE (λs::nat store. 0 < s ''y'')
 (''q'' :=r (λs. 0));
 (''r'' :=r (λs. s ''x''));
 (rwhile (λs. s ''y'' ≤ s ''r'') inv (λs. s ''x'' = s ''q'' * s ''y'' + s ''r'' ∧ 0 ≤ s ''r'')
  do
   (''q'' :=r (λs. s ''q'' + 1));
   (''r'' :=r (λs. s ''r'' − s ''y''))
  od)
POST (λs. s ''x'' = s ''q'' * s ''y'' + s ''r'' ∧ 0 ≤ s ''r'' ∧ s ''r'' < s ''y'')
```

With Isabelle we even need to decorate program variables with quotes because variables are represented as strings and strings are written that way in Isabelle. □

7.5.1. First Steps. The following examples consider simple assignments and their effects on Hoare triples in our semantic framework.

EXAMPLE 7.8. In the following paper-and-pencil proofs, we omit quotes around program variables.

(1) Suppose the assignment command $q := (\lambda s.\ 0)$ executes in any store. We expect that the value of q will be equal to 0 after execution in the new store. This is represented by the partial correctness specification

$$\mathsf{H}\,(\lambda s.\ \mathit{True})\,(q := \lambda s.\ 0)\,(\lambda s.\ s\, q = 0).$$

Isabelle predicates have type $\alpha \Rightarrow \mathit{bool}$, thus $\lambda s.\ \mathit{True}$ maps any state to True. Further, we need to write $\lambda s.\ 0$ instead of 0 because a function from stores to (natural) numbers is expected.

To verify this partial correctness specification, we calculate

$$\begin{aligned}\mathsf{H}\,(\lambda s.\ \mathit{True})\,(q := \lambda s.\ 0)\,(\lambda s.\ s\, q = 0) &\Leftrightarrow (\forall s.\ \mathit{True} \Rightarrow (\lambda s'.\ s' q = 0)\,(\Delta\, q\, 0\, s))\\ &\Leftrightarrow \forall s.\ (\Delta\, q\, 0\, s)\, q = 0\\ &\Leftrightarrow 0 = 0\\ &\Leftrightarrow \mathit{True},\end{aligned}$$

applying (H-assign-iff) in the first and Lemma 7.2(1) in the third step. Intuitively, we have set the value of q in the store to 0 in the postcondition in the second step of the calculation and then evaluated this predicate. Luckily, Isabelle simplifies away the last three steps.

```
lemma rPRE (λs. True) (''q'' :=r (λs. 0)) POST (λs. s ''q'' = 0)
 by simp
```

(2) The Hoare triple $\mathsf{H}\,(\lambda s.\ \mathit{True})\,(r := \lambda s.\ s\, x)\,(\lambda s.\ s\, r = s\, x)$ is valid. Now (H-assign-iff) requires checking that $(\lambda s'.\ s'\, r = s'\, x)(\Delta\, r\, (s\, x)\, s)$ holds for all states s. We calculate

$$\begin{aligned}(\lambda s'.\ s'\, r = s'\, x)(\Delta\, r\, (s\, x)\, s) &\Leftrightarrow (\Delta\, r\, (s\, x)\, s)\, r = (\Delta\, r\, (s\, x)\, s)\, x\\ &\Leftrightarrow s\, x = s\, x,\end{aligned}$$

using Lemma 7.2(1) and (2) to reduce the left-hand and right-hand side of the equation in the last step, respectively. This last identity is obviously true for all states s. The Isabelle proof is again straightforward.

lemma *rPRE* (λ*s*. *True*) (''*r*'' :=$_r$ (λ*s*. *s* ''*x*'')) *POST* (λ*s*. *s* ''*r*'' = *s* ''*x*'')
by *simp*

(3) The Hoare triple $\mathsf{H}\,(\lambda s.\ s\,q = n)\,(q := \lambda s.\ s\,q + 1)\,(\lambda s.s\,q = n + 1)$ is valid. We need to check that

$$\forall s.\ s\,q = n \Rightarrow (\lambda s'.\ s'\,q = n + 1)\,(\Delta\,q\,(s\,q + 1)\,s).$$

Assuming that $s\,q = n$, we calculate

$$\begin{aligned}(\lambda s'.\ s'\,q = n + 1)\,(\Delta\,q\,(s\,q + 1)\,s) &\Leftrightarrow (\Delta\,q\,(s\,q + 1)\,s)\,q = n + 1\\ &\Leftrightarrow (\Delta\,q\,(n + 1)\,s)\,q = n + 1\\ &\Leftrightarrow n + 1 = n + 1,\end{aligned}$$

using the assumption in the second and Lemma 7.2(1) in the last step. Once again, the last equation is obviously true. With Isabelle,

lemma
rPRE (λ*s*. *s* ''*q*'' = *n*)
(''*q*'' :=$_r$ (λ*s*. *s* ''*q*'' + *1*))
POST (λ*s*. *s* ''*q*'' = *n* + *1*)
by *simp*

(4) It is straightforward to show with Isabelle that

lemma
rPRE (λ*s*. *s* ''*r*'' − *s* ''*y*'' = *n*)
(''*r*'' :=$_r$ (λ*s*. *s* ''*r*'' − *s* ''*y*''))
POST (λ*s*. *s* ''*r*'' = *n*)
by *simp*

(5) Finally, verifying Hoare triples with Floyd's assignment rule makes little difference with Isabelle.

lemma *sPRE* (λ*s*. *5* = *5*) (''*x*'' :=$_s$ (λ*s*. *5*)) *POST* (λ*s*. *s* ''*x*'' = *5*)
by (*rule sH-cons2*, *rule sH-assign-floyd*) *simp*

lemma
rPRE (λ*s*. *s* ''*x*'' = *1*) (''*x*'' :=$_r$ (λ*s*. *s* ''*x*'' + *1*)) *POST* (λ*s*. *s* ''*x*'' = *2*)
by (*rule rH-cons2*, *rule rH-assign-floyd*) *simp*

In these two proofs, we have used a consequence rule to link the postcondition with that generated by Floyd's assignment rule. □

7.5.2. Simple Programs and Proof Outlines. We now discuss partial correctness of three simple algorithms that show the other rules of Hoare logic at work. Two of them are straight-line programs without loops. The first one swaps the values of two variables, the second one computes the maximum of two numbers. The third one performs integer division. Before verifying programs with Isabelle, it may

be helpful to prove them in a systematic way on paper first. We therefore introduce the format of *proof outlines* for such proofs by example.

EXAMPLE 7.9. The following program, in Imp-style notation, swaps the values of the variables x and y in the program store.

$$\begin{aligned} &z := x;\\ &x := y;\\ &y := z \end{aligned}$$

Hence if $x = m$ and $y = m$ before its execution, then $x = n$ and $y = m$ should hold afterwards. The partial correctness specification with Isabelle should therefore be

$$\begin{aligned} &PRE\ (\lambda s.\ s\,x = m \wedge s\,y = n)\\ &\ (z := \lambda s.\ s\,x);\\ &\ (x := \lambda s.\ s\,y);\\ &\ (y := \lambda s.\ s\,z)\\ &POST\ (\lambda s.\ s\,x = n \wedge s\,y = m) \end{aligned}$$

Applying (H-seq) to the program generates two intermediate assertions, which we do not know. But when we calculate the precondition of $y := \lambda s.\ s\,z$ using the postcondition, we can use it as the postcondition for $x := \lambda s.\ s\,y$ and then use it further to determine the precondition for $z := \lambda s\,x$, which we can then compare with the global precondition $x = m \wedge y = n$. We thus inject these intermediate assertions between the command of the program, proceeding from bottom to top, from postconditions to preconditions. To simplify notation, we now put all assertions into braces $\{-\}$, including the pre and postcondition. We also drop all references to stores and write the semi-formal proof outline

$$\begin{aligned} &\{x = m \wedge y = n\}\\ &\ z := x;\\ &\ x := y;\\ &\ y := z\\ &\{x = n \wedge y = m\} \end{aligned}$$

Starting with $y := z$ and postcondition $\{x = n \wedge y = m\}$ we apply (H-assign) and calculate the precondition for this command like in Example 7.8. Obviously, the value of y must now be equal to the value of z in the store. Hence replacing y with z in the postcondition (formally: $s\,y$ with $s\,z$) yields the precondition $\{x = n \wedge z = m\}$ of $y := z$, which we inject into the proof outline:

$$\begin{aligned} &\{x = m \wedge y = n\}\\ &\ z := x;\\ &\ x := y;\\ &\{x = n \wedge z = m\}\\ &\ y := z\\ &\{x = n \wedge y = m\} \end{aligned}$$

We now use this assertion as the postcondition for $x := y$ and proceed as before. Applying (H-assign) we replace x with y in $\{x = n \wedge z = m\}$ and inject the resulting

assertion:

$$\begin{array}{l}\{x = m \wedge y = n\}\\ \quad z := x;\\ \{y = n \wedge z = m\}\\ \quad x := y;\\ \{x = n \wedge z = m\}\\ \quad y := z\\ \{x = n \wedge y = m\}\end{array}$$

Using $\{y = n \wedge z = m\}$ as the postcondition for $z := x$ and applying (H-assign) once again, we replace z with x and—magically— obtain the precondition. This finishes the proof outline. A more formal version is

$$\begin{array}{l}\{\lambda s.\ s\,x = m \wedge s\,y = n\}\\ \quad (z := (\lambda s.\ s\,x));\\ \{\lambda s.\ s\,y = n \wedge s\,z = m\}\\ \quad (x := (\lambda s.\ s\,y));\\ \{\lambda s.\ s\,x = n \wedge s\,z = m\}\\ \quad (y := (\lambda s.\ s\,z))\\ \{\lambda s.\ s\,x = n \wedge s\,y = m\}\end{array}$$

In Isabelle, we cannot mimic this format. We simply use the rules of Hoare logic to generate verification conditions in the data domain which we then discharge.

```
lemma svarible-swap:
  sPRE (λs. s ''x'' = a ∧ s ''y'' = b)
    (''z'' :=s (λs. s ''x'')) ∘K
    (''x'' :=s (λs. s ''y'')) ∘K
    (''y'' :=s (λs. s ''z''))
   POST (λs. s ''x'' = b ∧ s ''y'' = a)
  apply (intro sta-kat.H-seq)
   apply (subst sH-assign, simp)+
  by simp
```

Yet when we apply *sH-assign* step-by-step we can witness the intermediate assertions in the postcondition of the next Hoare triple to be checked.

The Isabelle proof in the relational semantics is the same up to renaming functions and lemmas used. □

Example 7.10. Next we consider the partial correctness specification

$$\begin{array}{l}\{\mathit{True}\}\\ \quad \textbf{if } x \geq y \textbf{ then}\\ \quad\quad z := x\\ \quad \textbf{else}\\ \quad\quad z := y\\ \{z = \max(x, y)\}\end{array}$$

The proof outline for this simple program, which computes the maximum of two numbers, is

$$
\begin{array}{l}
\{\mathit{True}\} \\
\quad \textbf{if } x \geq y \textbf{ then} \\
\{x \geq y\} \\
\{x = \max(x, y)\} \\
\quad\quad z := x \\
\{z = \max(x, y)\} \\
\quad \textbf{else} \\
\{x < y\} \\
\{y = \max(x, y)\} \\
\quad\quad z := y \\
\{z = \max(x, y)\} \\
\{z = \max(x, y)\}
\end{array}
$$

In the lines immediately after the **then** and the **else**, the test and its negation appear as preconditions, as dictated by the conditional rule of Hoare logic. After the **then**, the consequence rule is used with $x \geq y \Rightarrow x = \max(x, y)$; after the **else** it is used with $x < y \Rightarrow y = \max(x, y)$.

With Isabelle, typing **apply** *rh-cond* after the postcondition of the program generates the two verification conditions for the branches in the jEdit proof window. Explicit applications of the consequence rule are then unnecessary, as Isabelle can discharge these proof obligations automatically. In fact, here is a fully automated proof:

```
lemma rmaximum:
  rPRE (λs::int store. True)
   (rif (λs. s ''x'' ≥ s ''y'')
    then (''z'' :=r (λs. s ''x''))
    else (''z'' :=r (λs. s ''y''))
    fi)
   POST (λs. s ''z'' = max (s ''x'') (s ''y''))
  by simp
```

□

Example 7.11. Finally, we reconsider the integer division algorithm from Chapter 1. Its partial correctness specification (in proof outline style) is

$$
\begin{array}{l}
\{0 < y\} \\
\quad q := 0; \\
\quad r := x; \\
\quad \textbf{while } y \leq r \textbf{ inv } x = q \cdot y + r \textbf{ do} \\
\quad\quad q := q + 1; \\
\quad\quad r := r - y \\
\{x = q \cdot y + r \wedge r < y\}
\end{array}
$$

where all variables are supposed to be natural numbers. The proof outline is

$$
\begin{array}{l}
\{0 < y\} \\
\{\mathit{True}\} \\
\{x = 0 \cdot y + x\} \\
\quad q := 0; \\
\{x = q \cdot y + x\} \\
\quad r := x; \\
\{x = q \cdot y + r\} \\
\quad \textbf{while } y \leq r \textbf{ inv } x = q \cdot y + r \textbf{ do} \\
\{x = q \cdot y + r \wedge y \leq r\} \\
\{x = (q + 1) \cdot y + (r - y)\} \\
\qquad q := q + 1; \\
\{x = q \cdot y + (r - y)\} \\
\qquad r := r - y \\
\{x = q \cdot y + r\} \\
\{x = q \cdot y + r \wedge r < y\}
\end{array}
$$

Interestingly, the precondition $0 < y$ is not used in the proof—it is only required for termination, which is assumed and hence disregarded by partial correctness. A Hoare logic for total program correctness would be needed to show termination of the algorithm. With Isabelle we show once again an apply-style proof that uses the rules of Hoare logic for verification condition generation.

```
lemma sinteger-division:
  assumes q = ''q'' and r = ''r''
  shows
  sPRE (λs::nat store. 0 < y)
    (q :=s (λs. 0)) ∘K
    (r :=s (λs. x)) ∘K
    (swhile (λs. y ≤ s r) inv (λs. x = s q * y + s r)
     do
      (q :=s (λs. s q + 1)) ∘K
      (r :=s (λs. s r − y))
     od)
  POST (λs. x = s q * y + s r ∧ s r < y)
  unfolding assms
  apply (intro sta-kat.H-seq)
    apply (subst sH-while-inv, intro sta-kat.H-seq)
       apply (rule sH-assign, simp)
      apply force+
   apply (subst sH-assign, simp)
  by (subst sH-assign-iff, simp)
```

The assumptions at the beginning of the lemma lead to more readable program code. □

Remark 7.12. In verification applications, the details of verification proofs usually do not matter—program verification is not a beauty contest. The degree of

proof automation is often more important. With Isabelle, proof automation can be enhanced by writing tactics. These can iteratively try to apply the rules of Hoare logic to a program. As there is one single structural rule per program construct, tactics can usually blast away the entire program structure, including assignments, so that only data level verification conditions remain. This is known as automated *verification condition generation.* Writing such tactics is not too hard with Isabelle, but not our concern in these lecture notes.

Remark 7.13. Most examples in this section were about numbers. Yet the data domain D of the store is modelled polymorphically by type α in Isabelle. Programs in which variables with different types occur can be modelled by sum types. It is then somewhat tedious to inject to summand types in order to access specific variables. However, our algebraic approach is modular with respect to other kinds of store such as record types or even monads. We can simply plug such stores into the relational or state transformer semantics. This is once again beyond the scope of these lecture notes.

CHAPTER 8

Program Refinement

We have so far verified programs post-hoc: we have added assertions to programs and then checked whether the resulting partial correctness specifications hold, using rules in the style of of Hoare logic for generating data-level verification conditions. This section outlines an alternative approach by which programs are constructed from specifications.

Imagine that a specification statement $\mathsf{R}\,p\,q$ allowed us to represent the most general program that satisfies the partial correctness specification with precondition p and postcondition q. Such a program might not be implementable in a simple language like Imp, it could be highly nondeterministic. Nevertheless, we could use the order relation $\leq$ of KAT and its instances in the relational and state transformer semantics to "narrow it down" to an implementable deterministic program. To achieve this, we need a systematic method supporting this construction.

Such a method should have laws for introducing the program constructs of our while language. It should make the process of "narrowing down" compositional, so that we can use it on parts of programs independently. It should be incremental, using transitivity of $\leq$. The method outlined, the systematic incremental and compositional construction of programs from specifications is known as step-wise *program refinement*; the laws that come with it are called *refinement calculus*. By constrast to Hoare logic they are often presented in algebraic form.

At first sight, program refinement may seem like pulling rabbits out of hats. But in reality, the program to be constructed must be known in advance like with Hoare logic. One might therefore prefer to see refinement calculi merely as alternative ways of presenting verification proofs, and of relating programs more explicitly with their correctness specifications.

Refinement techniques can be very powerful and versatile. Here we focus on a simple variant which is inspired by Carroll Morgan's approach.

8.1. Refinement Kleene Algebras with Tests

Most of the ingredients for a refinement calculus are already present in KAT. Its order $\leq$ can serve as a *refinement order* because $x \leq y$ means that x has at most the behaviours of y, which implies that x is at most as nondeterministic as y. This view is consistent with the concrete program semantics of relation and state transformer KAT. We henceforth call x a *refinement* of y if $x \leq y$.

Program refinement with refinement order $\leq$ in KAT is automatically incremental by transitivity of this order. It is also compositional with respect to the

program constructs of our simple while language in the sense that

$$\begin{gathered} x \leq y \Rightarrow zx \leq zy\,, \\ x \leq y \Rightarrow xz \leq yz\,, \\ x \leq x' \wedge y \leq y' \Rightarrow \textbf{if } p \textbf{ then } x \textbf{ else } y \leq \textbf{if } p \textbf{ then } x' \textbf{ else } y'\,, \\ x \leq y \Rightarrow \textbf{while } p \textbf{ do } x \leq \textbf{while } p \textbf{ do } y\,. \end{gathered}$$

However, we cannot expect to express Morgan's specification statement, which represents the largest programs satisfying a given pre/postcondition pair, in KAT. This would require taking suprema over all programs that satisfy the Hoare triples for such pairs, but only finite suprema are guaranteed to exist in KAT. We thus extend KAT by a function and suitable axioms.

DEFINITION 8.1. A *refinement Kleene algebra with tests* (rKAT) is a Kleene algebra with tests (K, B) equipped with a map $\mathsf{R} : B \to B \to K$ that satisfies, for all $p, q \in B$,

$$x \leq \mathsf{R}\,p\,q \Leftrightarrow \mathsf{H}\,p\,x\,q\,.$$

For each $p, q \in B$, we call $\mathsf{R}\,p\,q$ the *refinement statement* for precondition p and postcondition q.

It is easy to check that R satisfies the following characteristic formulas.

LEMMA 8.2. *Let K be a* rKAT*. Then, for all $x \in K$ and $p, q \in B$,*

$$\mathsf{H}\,p\,(\mathsf{R}\,p\,q)\,q\,, \tag{R1}$$

$$\mathsf{H}\,p\,x\,q \Rightarrow x \leq \mathsf{R}\,p\,q\,. \tag{R2}$$

PROOF. Exercise. □

Formula (R1) states that $\mathsf{R}\,p\,q$ satisfies the partial correctness specification with precondition p and postcondition q, while (R2) states that $\mathsf{R}\,p\,q$ is indeed the greatest element of K that does so. It also follows directly from the definition of Hoare triples that, for all $x \in K$ and $p, q \in B$,

$$x \leq \mathsf{R}\,p\,q \Leftrightarrow px \leq xq\,.$$

In the relational and state transformer semantics, the suprema needed for defining the specification statement explicitly exist:

$$\mathsf{R}\,P\,Q = \bigcup\{R \mid \mathsf{H}\,P\,R\,Q\} \qquad \text{and} \qquad \mathsf{R}\,P\,Q = \bigvee\{f \mid \mathsf{H}\,P\,f\,Q\}\,,$$

where, for any $F \subseteq \mathsf{Sta}\,X$, we define $(\bigvee F)\,x = \bigcup\{f\,x \mid f \in F\}$.

PROPOSITION 8.3. *The structures $\mathsf{Rel}\,X$ and $\mathsf{Sta}\,X$, with R defined as above, form refinement Kleene algebras.*

PROOF. Exercise. □

The formalisation of rKAT and its relational and state transformer model with Isabelle is straightforward. rKAT is formalised as a type class extending that for KAT, the models are obtained via interpretation proofs extending those for KAT. Details can be found in the Isabelle theories.

8.2. A Simple Refinement Calculus

8.2.1. Propositional Refinement Calculus. It is straightforward to derive variants of Morgan's refinement laws in rKAT—ignoring assignment laws, as usual with algebra.

THEOREM 8.4. *Let K be a* rKAT*. Then, for all $p, p', q, q', r \in B$ and $x, y \in K$, the laws of the* propositional refinement calculus PRC *are derivable.*

$$1 \leq \mathsf{R}\, p\, p\,, \tag{R-skip}$$

$$p \leq p' \wedge q' \leq q \Rightarrow \mathsf{R}\, p'\, q' \leq \mathsf{R}\, p\, q\,, \tag{R-cons}$$

$$\mathsf{R}\, p\, r \cdot \mathsf{R}\, r\, q \leq \mathsf{R}\, p\, q, \tag{R-seq}$$

$$\textbf{if } t \textbf{ then } \mathsf{R}\,(tp)\, q \textbf{ else } \mathsf{R}\,(\bar{t}p)\, q \leq \mathsf{R}\, p\, q\,, \tag{R-cond}$$

$$\textbf{while } t \textbf{ do } \mathsf{R}\,(tp)\, p \leq \mathsf{R}\, p\,(\bar{t}p)\,, \tag{R-while}$$

$$x \leq \mathsf{R}\, 0\, 1\,, \tag{R-01}$$

$$\mathsf{R}\, 1\, 0 \leq x\,. \tag{R-10}$$

PROOF. For each program construct, the corresponding PHL rule allows deriving the refinement law in PRC.

For (R-skip), $\mathsf{H}\, p\, 1\, p \Rightarrow 1 \leq \mathsf{R}\, p\, p$ follows from (H-skip) and (R2).

For (R-cons), suppose $p \leq p'$ and $q' \leq q$. Then

$$\mathsf{H}\, p'\,(\mathsf{R}\, p'\, x\, q')\, q' \Rightarrow \mathsf{H}\, p\,(\mathsf{R}\, p'\, x\, q')\, q \Rightarrow \mathsf{R}\, p'\, q' \leq \mathsf{R}\, p\, q\,.$$

The initial Hoare triple holds by (R1); the steps follow from (H-cons) and (R2).

For (R-seq),

$$\mathsf{H}\, p\,(\mathsf{R}\, p\, r)\, r \wedge \mathsf{H}\, r(\mathsf{R}\, r\, q)\, q \Rightarrow \mathsf{H}\, p\,(\mathsf{R}\, p\, r \cdot \mathsf{R}\, r\, q)\, q \Rightarrow \mathsf{R}\, p\, r \cdot \mathsf{R}\, r\, q \leq \mathsf{R}\, p\, q\,.$$

The initial conjunction holds by (R1); the steps follow from (H-seq) and (R2).

For (R-cond),

$$\begin{aligned}\mathsf{H}\,(tp)\,(\mathsf{R}\,(tp)\, q)\, q \wedge \mathsf{H}\,(\bar{t}p)\,(\mathsf{R}\,(\bar{t}p)\, q)\, q &\Rightarrow \mathsf{H}\, p\,(\textbf{if } t \textbf{ then } \mathsf{R}\,(tp)\, q \textbf{ else } \mathsf{R}\,(\bar{t}p)\, q)\, q \\ &\Rightarrow \textbf{if } t \textbf{ then } \mathsf{R}\,(tp)\, q \textbf{ else } \mathsf{R}\,(\bar{t}p)\, q \leq \mathsf{R}\, p\, q\,.\end{aligned}$$

The initial conjunction holds by (R1); the steps follow from (H-cond) and (R2).

For (R-while),

$$\begin{aligned}\mathsf{H}\,(tp)\,(\mathsf{R}\,(tp)\, p)\, p &\Rightarrow \mathsf{H}\, p\,(\textbf{while } t \textbf{ do } \mathsf{R}\,(tp)\, p)\,(\bar{t}p) \\ &\Rightarrow \textbf{while } t \textbf{ do } \mathsf{R}\,(tp)\, p \leq \mathsf{R}\, p\,(\bar{t}p)\,.\end{aligned}$$

The initial conjunction holds by (R1); the steps follow from (H-while) and (R2).

The refinement laws (R-01) and (R-10) are left as exercises. They are not needed in refinement proofs. □

The refinement laws (R-seq)-(R-while)—the sequential composition law, conditional law and while law—introduce control structure to specification statements from right to left. By contrast to Hoare logic, (R-skip) is quite useful in the refinement calculus. It allows the refinement of residual specification statements to 1, which makes them vanish in the algebra. Rule (R-cons) is important for adapting pre- and postconditions in proofs, like its counterpart (H-cons) in Hoare logic. Rule (R-seq) introduces once again an intermediate assertion r; rule (R-while) is more specific than the other laws in that the loop invariant p occurs in pre- and postconditions. Introducing a while loop thus requires using (R-cons). Alternatively, one

can derive a macro law with a loop invariant from (H-while-inv) as in the proofs above.

LEMMA 8.5. *Let (K, B) be a* rKAT*. For all $x \in K$ and $p, q, i, t \in B$,*

(R-while-inv) $$p \leq i \wedge \bar{t}i \leq q \Rightarrow \textbf{while } t \textbf{ do } \mathsf{R}\,(ti)\,i \leq \mathsf{R}\,p\,q\,.$$

Annotating this invariant in the while loop is not necessary for refinement.

8.2.2. Specialised Refinement Laws. For refinement proofs with Isabelle it is once again helpful to derive specific structural refinement laws for Rel X and Sta X that expand directly to predicates and use the isomorphisms between subidentity relations and state transformers as well as predicates. We show only (part of) the relational development. Additional details can be found in our Isabelle theories.

```
abbreviation rR :: 'a pred ⇒ 'a pred ⇒ 'a rel (R_r) where
  R_r P Q ≡ rel-Re ⌈P⌉_r ⌈Q⌉_r

lemma rR-unfold: R_r P Q = ⋃{R. ∀ x y. P x ⟶ (x,y) ∈ R ⟶ Q y}
  ⟨Proof⟩

lemma rR-cons:
  assumes ∀ s. P s ⟶ P' s
  and ∀ s. Q' s ⟶ Q s
  shows R_r P' Q' ⊆ R_r P Q
  ⟨Proof⟩

lemma rR-skip [simp]: (Id ⊆ R_r P Q) = (∀ s. P s ⟶ Q s)
  ⟨Proof⟩

lemma rR-while: rwhile Q do R_r (λs. P s ∧ Q s) P od ⊆ R_r P (λs. P s ∧ ¬ Q s)
  ⟨Proof⟩

lemma rR-while-var:
  assumes R ⊆ R_r (λs. P s ∧ Q s) P
  shows rwhile Q do R od ⊆ R_r P (λs. P s ∧ ¬ Q s)
  ⟨Proof⟩
```

8.2.3. Refinement Laws for Assignments. We can reuse our simple store model and the relational and state transformer semantics of variable assignments from Chapter 7 to derive refinement laws for assignment in the relational and state transformer semantics of rKAT. First we present a statement with generic notation that abstracts from the particular semantics.

LEMMA 8.6. *In* Rel S *or* Sta S*, writing $\leq$ for the refinement order, the following assignment law of Morgan's refinement calculus is derivable.*

(R-assign) $$(v := e) \leq \mathsf{R}\,P\,Q \Leftrightarrow (\forall s.\ P\,s \Rightarrow Q\,(set\,v\,e\,s))\,.$$

PROOF. Exercise. □

With Isabelle, in the relational semantics, its derivation is straightforward.

```
lemma rR-assign [simp]: ((v :=_r e) ⊆ R_r P Q) = (∀ s. P s ⟶ Q (set v e s))
```

by (*simp add*: *rel-rkat.spec-def*)

Refinement proofs often require the introduction of an assignment command before or after a specification statement, separated by a sequential composition. The two following laws support this.

LEMMA 8.7. *In* $\mathsf{Rel}\,S$ *or* $\mathsf{Sta}\,S$*, writing* $\leq$ *for the refinement order,*

(R-assignl) $\qquad (\forall s.\ P\,s \Rightarrow P'\,(set\,v\,e\,s)) \Rightarrow (v := e) \cdot \mathsf{R}\,P'\,Q \leq \mathsf{R}\,P\,Q\,,$

(R-assignr) $\qquad (\forall s.\ Q'\,s \Rightarrow Q\,(set\,v\,e\,s)) \Rightarrow \mathsf{R}\,P\,Q' \cdot (v := e) \leq \mathsf{R}\,P\,Q\,.$

PROOF. Exercise. □

With Isabelle, in the relational model,

```
lemma rR-assignl:
  assumes ∀ s. P s ⟶ P' (set v e s)
  shows (v :=r e) ; (Rr P' Q) ⊆ Rr P Q
proof −
    have v :=r e ⊆ Rr P P'
    using assms by simp
  then show ?thesis
    by (meson order.trans rel-d.mult-isor rel-rkat.R-seq)
qed
```

The first step of the proof uses law *rR-assign*, the second one law *rel-rkat.R-seq*, the instance of the sequential composition law of rKAT in $\mathsf{Rel}\,S$. The derivation of (R-assignr is similar. Morgan calls (R-assignl) the *leading* and (R-assignr) the *following law* for assignments.

We can now start constructing programs by refinement.

8.3. Examples: Program Refinement

We construct the programs for variable swap, maximum and integer division from their specification statements as examples. We only show the Isabelle proofs. The structure of mathematical proofs is the same. Only the syntax in textbook style proofs can be simplified, as usual.

EXAMPLE 8.8. Starting from specification statement

$$\mathsf{R}\,(x = a \wedge y = b)\,(x = b \wedge y = a),$$

we first introduce the assigment $z := x$ using (R-assignl).

```
lemma var-swap-ref1:
  Rr (λs. s ''x'' = a ∧ s ''y'' = b) (λs. s ''x'' = b ∧ s ''y'' = a)
  ⊇ (''z'' :=r (λs. s ''x''));
     Rr (λs. s ''z'' = a ∧ s ''y'' = b) (λs. s ''x'' = b ∧ s ''y'' = a)
  by (rule rR-assignl) simp
```

The new specification statement $\mathsf{R}\,(z = a \wedge y = b)\,(x = b \wedge y = a)$ is generated by this refinement law. We use it to introduce $x := y$ by a second left assignment.

```
lemma var-swap-ref2:
  Rr (λs. s ''z'' = a ∧ s ''y'' = b) (λs. s ''x'' = b ∧ s ''y'' = a)
```

$\supseteq$ $(''x'' :=_r (\lambda s.\ s\ ''y''))$;
$R_r\ (\lambda s.\ s\ ''z'' = a \wedge s\ ''x'' = b)\ (\lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a)$
by (*rule rR-assignl*) *simp*

Using the resulting specification statement $\mathsf{R}\,(z = a \wedge x = b)\,(x = b \wedge y = a)$, we introduce $y := z$ in the third refinement step.

lemma *var-swap-ref3*:
$R_r\ (\lambda s.\ s\ ''z'' = a \wedge s\ ''x'' = b)\ (\lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a)$
$\supseteq$ $(''y'' :=_r (\lambda s.\ s\ ''z''))$;
$R_r\ (\lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a)\ (\lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a)$
by (*rule rR-assignl*) *simp*

The pre- and postcondition in the resulting specification statement are the same. We could now use (R-skip) to refine it to skip. Instead we call Isabelle to tie individual refinements together and perform this task under the hood.

lemma *var-swap-ref*:
$R_r\ (\lambda s.\ s\ ''x'' = a \wedge s\ ''y'' = b)\ (\lambda s.\ s\ ''x'' = b \wedge s\ ''y'' = a)$
$\supseteq$ $(''z'' :=_r (\lambda s.\ s\ ''x'')) ; (''x'' :=_r (\lambda s.\ s\ ''y'')) ; (''y'' :=_r (\lambda s.\ s\ ''z''))$
by (*simp add*: *rel-rkat.R2 rvarible-swap*)

This final step shows the specification statement in the first line and the program constructed by step-wise refinement in the second one. The variable swap program is now correct by construction. □

Remark 8.9. In the proofs above, the refined programs have not been found by Isabelle. The refinement steps need to be typed into Isabelle; Isabelle can only check their correctness. One might be able to get Isabelle to infer at least the new specification statement in a left or right assignment step, but we do not consider this any further.

Example 8.10. Next we construct the algorithm that computes the maximum of two numbers by step-wise refinement. This time we use the right assignment law in a first step and the skip law in a second step to construct the assignments in the branches of the conditional.

lemma *max1*:
$R_r\ (\lambda s{::}int\ store.\ s\ ''x'' \geq s\ ''y'')\ (\lambda s.\ s\ ''z'' = max\ (s\ ''x'')\ (s\ ''y''))$
$\supseteq R_r\ (\lambda s.\ s\ ''x'' \geq s\ ''y'')\ (\lambda s.\ s\ ''y'' \leq s\ ''x'') ; (''z'' :=_r (\lambda s.\ s\ ''x''))$
by (*smt fun-update-simp1 fun-update-simp2 rR-assignr*)

lemma *max11*:
$R_r\ (\lambda s.\ s\ ''x'' \geq s\ ''y'')\ (\lambda s.\ s\ ''y'' \leq s\ ''x'') ; (''z'' :=_r (\lambda s.\ s\ ''x''))$
$\supseteq\ ''z'' :=_r (\lambda s.\ s\ ''x'')$
using *rel-rkat.R-skip* **by** *fastforce*

lemma *max2*:
$R_r\ (\lambda s{::}int\ store.\ s\ ''x'' < s\ ''y'')\ (\lambda s.\ s\ ''z'' = max\ (s\ ''x'')\ (s\ ''y''))$
$\supseteq R_r\ (\lambda s.\ s\ ''x'' < s\ ''y'')\ (\lambda s.\ s\ ''x'' < s\ ''y'') ; (''z'' :=_r (\lambda s.\ s\ ''y''))$
by (*smt fup-simp1 fup-simp2 rR-assignr*)

lemma *max21*:
$R_r\ (\lambda s.\ s\ ''x'' < s\ ''y'')\ (\lambda s.\ s\ ''x'' < s\ ''y'')\ ;\ (''z'' :=_r\ (\lambda s.\ s\ ''y''))$
$\supseteq\ ''z'' :=_r\ (\lambda s.\ s\ ''y'')$
using *rel-rkat.R-skip* **by** *fastforce*

The conditional law can now be used to integrate the assignments of the two branches. Yet first we simply generate the refinement statements for these. An application of (R-cons) is used under the hood to adapt the pre- and postconditions in the specification statements.

lemma *max-cond*:
$R_r\ (\lambda s{::}int\ store.\ True)\ (\lambda s.\ s\ ''z'' = max\ (s\ ''x'')\ (s\ ''y''))$
$\supseteq\ rif\ (\lambda s.\ s\ ''x'' \geq s\ ''y'')$
$\quad then\ (R_r\ (\lambda s.\ s\ ''x'' \geq s\ ''y'')\ (\lambda s.\ s\ ''z'' = max\ (s\ ''x'')\ (s\ ''y'')))$
$\quad else\ (R_r\ (\lambda s.\ s\ ''x'' < s\ ''y'')\ (\lambda s.\ s\ ''z'' = max\ (s\ ''x'')\ (s\ ''y'')))$
fi
by (*simp add*: *rR-cond-var rR-cons*)

We can now pull the results together. Yet when we simply invoke Sledgehammer, it prefers to use our previous verification proof for the maximum algorithm in Hoare logic together with rule (R2).

lemma *maximum*:
$R_r\ (\lambda s{::}int\ store.\ True)\ (\lambda s.\ s\ ''z'' = max\ (s\ ''x'')\ (s\ ''y''))$
$\supseteq\ (rif\ (\lambda s.\ s\ ''x'' \geq s\ ''y'')$
$\quad then\ (''z'' :=_r\ (\lambda s.\ s\ ''x''))$
$\quad else\ (''z'' :=_r\ (\lambda s.\ s\ ''y''))$
$fi)$
using *rel-rkat.R2 rmaximum* **by** *blast*

We have not tried to force Isabelle to find another proof, as this one illustrates the tight relationship between post-hoc verification and refinement. □

Example 8.11. Our final example is integer division. A problem with refinement proofs is that specification statements can become quite long and hard to read. Hence we introduce abbreviations for the loop invariant $x = q \cdot y + r$ and the test of the loop $y \leq r$ before we start.

abbreviation $Idiv\ s \equiv s\ ''x'' = s\ ''q'' * s\ ''y'' + s\ ''r''$

abbreviation $Tdiv\ s \equiv s\ ''y'' \leq s\ ''r''$

First, proceeding like for the variable swap program, we introduce the two variable assignments that initialise q and r, starting from the intial specification statement

$$\mathsf{R}\,(0 < y)\,(x = q \cdot y + r \wedge r < y)$$

and using (R-assignl).

lemma *div-init1*: $R_r\ (\lambda s{::}nat\ store.\ 0 < s\ ''y'')\ (\lambda s.\ Idiv\ s\ \wedge \neg Tdiv\ s) \supseteq$
$(''r'' :=_r\ (\lambda s.\ s\ ''x''));$
$R_r\ (\lambda s.\ s\ ''r'' = s\ ''x'' \wedge s\ ''x'' \geq 0)\ (\lambda s.\ Idiv\ s \wedge \neg Tdiv\ s)$
by (*rule rR-assignl*) *simp*

lemma *div-init2*: $R_r\ (\lambda s{::}nat\ store.\ s\ ''r'' = s\ ''x'')\ (\lambda s.\ Idiv\ s \wedge \neg\ Tdiv\ s) \supseteq$
$(''q'' :=_r\ (\lambda s.\ 0))$;
$R_r\ (\lambda s.\ s\ ''r'' = s\ ''x'' \wedge s\ ''q'' = 0)\ (\lambda s.\ Idiv\ s \wedge \neg\ Tdiv\ s)$
by (*rule rR-assignl*) *simp*

We then use (R-cons) to obtain a specification statement that allows introducing a while loop. Its precondition must be the loop invariant and its postcondition the conjunction of the invariant with the negation of the test of the loop.

lemma *div-init3*:
$R_r\ (\lambda s{::}nat\ store.\ s\ ''r'' = s\ ''x'' \wedge s\ ''q'' = 0)\ (\lambda s.\ Idiv\ s \wedge \neg\ Tdiv\ s) \supseteq$
$R_r\ Idiv\ (\lambda s.\ Idiv\ s \wedge \neg\ Tdiv\ s)$
by (*simp-all add*: *rR-cons*)

We can now introduce the while loop, as intended.

lemma *div-loop*: $R_r\ Idiv\ (\lambda s.\ Idiv\ s \wedge \neg\ Tdiv\ s) \supseteq$
rwhile Tdiv do $(R_r\ (\lambda s.\ Idiv\ s \wedge\ Tdiv\ s)\ Idiv)$ *od*
by (*simp add*: *rR-while*)

The specification statement in the body of the loop can now be used to introduce the assignments in the body. We work backwards using (R-assignr).

lemma *div-body1*: $R_r\ (\lambda s.\ Idiv\ s \wedge\ Tdiv\ s)\ Idiv \supseteq$
$R_r\ (\lambda s.\ Idiv\ s \wedge\ Tdiv\ s)\ (\lambda s{::}nat\ store.\ s\ ''x'' = s\ ''q'' * s\ ''y'' + (s\ ''r'' - s\ ''y''))$;
$(''r'' :=_r\ (\lambda s.\ s\ ''r'' - s\ ''y''))$
by (*simp add*: *rR-assignr*)

lemma *div-body2*:
$R_r\ (\lambda s.\ Idiv\ s \wedge\ Tdiv\ s)\ (\lambda s{::}nat\ store.\ s\ ''x'' = s\ ''q'' * s\ ''y'' + (s\ ''r'' - s\ ''y'')) \supseteq$
$R_r\ (\lambda s.\ Idiv\ s \wedge\ Tdiv\ s)\ (\lambda s.\ Idiv\ s \wedge\ \ Tdiv\ s)$; $(''q'' :=_r\ (\lambda s.\ s\ ''q'' + 1))$
by (*simp add*: *rR-assignr*)

Next we compose the assignments in the body of the loop and integrate them into the loop, using the order-preservation laws for while-loops derived at the beginning of Section 8.1.

lemma *div-while*: *rwhile Tdiv do* $R_r\ (\lambda s.\ Idiv\ s \wedge\ Tdiv\ s)\ Idiv$ *od* $\supseteq$
rwhile Tdiv do
$(''q'' :=_r\ (\lambda f.\ f\ ''q'' + (1{::}nat)))$;
$(''r'' :=_r\ (\lambda f.\ f\ ''r'' - f\ ''y''))$
od
apply (*rule rel-kat.while-iso*)
using *div-body1 div-body2 rel-rkat.R-skip* **by** *blast*

Finally, we integrate the loop and initialisation into the full program.

lemma *integer-division*: $R_r\ (\lambda s{::}nat\ store.\ 0 < s\ ''y'')\ (\lambda s.\ Idiv\ s \wedge \neg\ Tdiv\ s) \supseteq$
$(''r'' :=_r\ (\lambda s.\ s\ ''x''))$;
$(''q'' :=_r\ (\lambda s.\ 0))$;
(*rwhile Tdiv do*

```
  (''q'' :=r (λs::nat store. s ''q'' + 1));
  (''r'' :=r (λs. s ''r'' − s ''y''))
od)
using div-init1 div-init2 div-init3 div-loop div-while by force
```

Once again, this program is correct by construction. □

In these simple examples, there is almost a one-to-one correspondence between refinement proofs and proof outlines. We can see refinement proofs as a slightly more systematic and more verbose way of writing such outlines, as already mentioned.

Full fledged refinement calculi are more complex than the one presented. They can express program equivalences and transformations similar to those of KAT and even beyond in total correcteness settings. Nevertheless, refinement calculi are rarely used in real world applications, except perhaps for data refinement, which deals with refinement relations between data types. Automation is of course an important feature, and pure refinement approaches seem too fine-grained in this regard. It seems more realistic to combine refinement and post hoc verification into hybrid approaches for the compositional verification of larger systems, where components can be post-hoc verified and integrated into the global system by refinement. rKAT provides a simple framework for this.

CHAPTER 9

Another Algebra of Programs

While Hoare logic investigates correctness specifications of programs, it does not does not explicitly *compute* the effect of assignments on preconditions, when using Hoare's assignment law, or that on postconditions, when using Floyd's. In the relational and state transformer model, such computations are nevertheless straightforward: for instance, $(s, s') \in (v :=_r e)\,;Q$ if and only if $s' = set\, v\, e\, s$ and $Q\, s'$. The precondition that characterises the input states from which $v :=_r e$ executes into states satisfying the postcondition Q can therefore be calculated as the domain of the relation $(v :=_r e)\,;Q$,

$$dom\,((v :=_r e)\,;Q) = Q\,(set\, v\, e\, s).$$

We have freely identified relations, predicates and sets in this explanation, as usual. More generally,

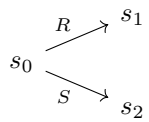
$$dom\,(R\,;Q) = \{s \mid \exists s'.\ (s, s') \in R \wedge Q\, s'\}$$

models the set of all those states from which the relational program R *may* terminate in states satisfying Q. For nondeterministic choices, however,

$$dom\,((R \cup S)\,;Q) = dom\,(R\,;Q) \cup dom\,(S\,;Q).$$

Executing R from states in $dom\,((R \cup S)\,;Q)$ therefore does not guarantee that it terminates in a state where Q holds (same for S, by symmetry).

Example 9.1. Consider the relations R and S defined by



and suppose that predicate Q holds in s_2, but not in s_1. Then

$$dom\,((R \cup S)\,;Q) = \{s_0\} = dom\,(S\,;Q), \qquad \text{and} \qquad dom\,(R\,;Q) = \emptyset,$$

because R terminates in s_1 when executed from s_0, but does not execute into s_2. □

With partial program correctness in mind, we should, in fact, rather model the set of those states from which program R *must* terminate in states satisfying postcondition Q. It is given by

$$-dom\,(R\,;\overline{Q}) = \{s \mid \forall s'.\ (s, s') \in R \Rightarrow Q\, s'\},$$

where $\overline{Q} = Id_X - Q$ defines an appropriate complement of Q in $\mathsf{Id}{\downarrow_X}$, as in Section 4.1.2, whereas $-$ is the complement on sets.

For nondeterministic choice, in particular,

$$-dom\,((R \cup S)\,;\overline{Q}) = -dom\,(R\,;\overline{Q}) \cap -dom\,(S\,;\overline{Q})\,.$$

This models the set of all states from which both R and S must terminate in states satisfying Q.

EXAMPLE 9.2. For the particular relations in Example 9.1, one might perhaps expect $-dom\,((R \cup S)\,;\overline{Q})$ to be empty. Nevertheless, $-dom\,(R\,;\overline{Q}) = \{s_1, s_2\}$ and $-dom\,(S\,;\overline{Q}) = \{s_0, s_1, s_2\}$, so that $-dom\,((R \cup S)\,;\overline{Q}) = \{s_1, s_2\}$. Obviously, neither R nor S can be executed from s_1 or s_2 and we can rewrite $-dom(R\,;\overline{Q}) = \{s \mid \forall s'.\ (s, s') \notin R \vee Q\, s'\}$, which explains this outcome. □

Note that $-dom\,(R\,;\overline{Q}) = dom\,(R\,;Q)$ holds whenever the relation $R \subseteq X \times X$ is deterministic, that is, $dom\,R = X$ and $(x, y) \in R$, $(x, y') \in R$ imply $y = y'$. For our program semantics however, in particular that of while-loops, nondeterminism is essential.

In sum, we need the *antidomain* operation $adom = -dom$ of a relation or state transformer, as we used antitests instead of tests to formalise KAT with Isabelle in Chapter 3. We would like to extend Kleene algebras with such an operation. With this alternative to KAT we could then develop a similar approach for verifying programs, but with more computational verification conditions.

9.1. Modal Kleene Algebras

We now extend a Kleene algebra K with an antidomain operation $\mathsf{ad} : K \to K$ so that its axioms generate a boolean subalgebra of assertions. This operation should be similar to the antitest operation in Chapter 3, whose axioms generated a boolean subalgebra of tests. In addition, the domain operation $\mathsf{d} = \mathsf{ad} \circ \mathsf{ad}$ should model the properties of the domain of a relation or state transformer we care about.

Intuitively, the antidomain of a program should model all those states from which the program cannot execute; the domain of a program should model all those all those states from which it can execute. By opposition we can add an antirange operation $\mathsf{ar} : K \to K$ and define a range or codomain operation $r = \mathsf{ar} \circ \mathsf{ar}$. The antirange of a program should model all those states into which a program cannot execute; the range of a program should model all those states in which it can. As with tests, we view these sets as programs that can be executed from some states to themselves, or not at all.

9.1.1. Definitions and Basic Properties. We capture these requirements by the following data.

DEFINITION 9.3. An *antidomain Kleene algebra* (AKA) is a Kleene algebra K equipped with an *antidomain operation* $\mathsf{ad} : K \to K$ that satisfies

$$\mathsf{ad}\,x \cdot x = 0, \qquad \mathsf{ad}\,x + \mathsf{ad}\,(\mathsf{ad}\,x) = 1, \qquad \mathsf{ad}\,(x \cdot \mathsf{ad}\,(\mathsf{ad}\,y)) \leq \mathsf{ad}\,(x \cdot y).$$

A *domain operation* $\mathsf{d} : K \to K$ is then defined as $\mathsf{d} = \mathsf{ad} \circ \mathsf{ad}$ as expected.

We explain the antidomain axioms first in our intuitive program semantics. By the first axiom, programs cannot be executed from their antidomains. By the second one, the domain and antidomain elements of a program are complements. By the third one, the set of states from which a sequential composition of two programs cannot be executed contains that from which the first program cannot execute into the domain of the second one.

Antirange Kleene algebras are defined by opposition.

DEFINITION 9.4. An *antirange Kleene algebra* is a Kleene algebra K equipped with an *antirange* operation $\mathsf{ar} : K \to K$ that satisfies

$$x \cdot \mathsf{ar}\, x = 0, \qquad \mathsf{ar}\, x + \mathsf{ar}\,(\mathsf{ar}\, x) = 1, \qquad \mathsf{ar}\,(\mathsf{ar}\,(\mathsf{ar}\, x) \cdot y) \leq \mathsf{ar}(x \cdot y).$$

A *range operation* $\mathsf{r} : K \to K$ can then be defined as $\mathsf{r} = \mathsf{ar} \circ \mathsf{ar}$. An antirange Kleene algebra is therefore nothing but a domain Kleene algebra in the opposite Kleene algebra.

DEFINITION 9.5. A *modal Kleene algebra* (MKA) is an antidomain Kleene algebra that is also an antirange Kleene algebra.

Here are some useful properties of the domain and antidomain operation.

LEMMA 9.6. *In every* AKA,

(1) $\mathsf{d}\, x x = x$,
(2) $\mathsf{ad}\,(xy) = \mathsf{ad}\,(x \mathsf{d}\, y)$ *and* $\mathsf{d}\,(xy) = \mathsf{d}\,(x \mathsf{d}\, y)$,
(3) $\mathsf{ad}\, x \leq 1$ *and* $\mathsf{d}\, x \leq 1$,
(4) $\mathsf{ad}\, 0 = 1$, $\mathsf{ad}\, 1 = 0$, $\mathsf{d}\, 0 = 0$ *and* $\mathsf{d}\, 1 = 1$,
(5) $\mathsf{ad}\,(x + y) = \mathsf{ad}\, x \cdot \mathsf{ad}\, y$ *and* $\mathsf{d}\,(x + y) = \mathsf{d}\, x + \mathsf{d}\, y$,
(6) $\mathsf{d}\,(\mathsf{d}\, x) = \mathsf{d}\, x$, $\mathsf{d}\,(\mathsf{ad}\, x) = \mathsf{ad}\, x$ *and* $\mathsf{ad}\,(\mathsf{d}\, x) = \mathsf{ad}\, x$,
(7) $x \leq y \Rightarrow \mathsf{ad}\, y \leq \mathsf{ad}\, x$ *and* $x \leq y \Rightarrow \mathsf{d}\, x \leq \mathsf{d}\, y$,
(8) $\mathsf{ad}\,(\mathsf{ad}\, x \cdot y) = \mathsf{d}\, x + \mathsf{ad}\, y$ *and* $\mathsf{d}\,(\mathsf{d}\, x \cdot y) = \mathsf{d}\, x \cdot \mathsf{d}\, y$,
(9) $\mathsf{ad}\, x \cdot \mathsf{ad}\, x = \mathsf{ad}\, x$ *and* $\mathsf{d}\, x \cdot \mathsf{d}\, x = \mathsf{d}\, x$,
(10) $\mathsf{ad}\, x \cdot \mathsf{ad}\, y = \mathsf{ad}\, y \cdot \mathsf{ad}\, x$ *and* $\mathsf{d}\, x \cdot \mathsf{d}\, y = \mathsf{d}\, y \cdot \mathsf{d}\, x$,
(11) $\mathsf{ad}\, y \cdot x = 0 \Leftrightarrow \mathsf{ad}\, y \leq ad\, x$ *and* $\mathsf{d}\, y \cdot x = x \Leftrightarrow \mathsf{d}\, x \leq \mathsf{d}\, y$,
(12) $xy = 0 \Leftrightarrow x \mathsf{d}\, y = 0$,
(13) $\mathsf{ad}\, x \cdot \mathsf{d}\, x = 0$.

Proofs and additional properties can be found in our Isabelle theories. Properties of domain are usually immediate from those of antidomain. Properties of range and antirange follow by opposition in MKA. We therefore do not list them.

Let us consider these properties in more detail. According to (1), restricting program x in its input to those states from which it can be executed is no restriction at all. By (2), the states from which a sequential composition of two programs can be executed equals that from which the first program can be executed into states allowing the execution of the second program, and likewise for antidomain. By (3), domain and antidomains are subidentities. By (4), the domain of *abort* is the empty set of states and its antidomain therefore the set of all states. Likewise, the domain of *skip* is the set of all states, and its antidomain therefore empty. By (5), the set from which the nondeterministic choice of two programs can be executed equals the union of those from which the individual programs can be executed. Accordingly, the set from which this nondeterministic choice cannot be executed is the intersection, represented by $\cdot$, of those from which the individual programs cannot be executed. The first two properties in (6) say that domain and antidomain elements, as programs, are their own domain elements. The third one states that the antidomain acts as complementation on domain elements. By (7), if x refines y, the set of states from which it can be executed is contained into the set of those states from which y can be executed. By (8), the set of states from which the input-restriction of program y to those states from which x can be executed equals the intersection of the set of states from which x and y can be executed. As before, the statement for antidomain is obtained by throwing in some negations.

The properties in (9) and (10) are standard properties of sets with $\cdot$ as intersection. By (11), the domain of x is the least domain element for which the left-absorption property in the left-hand side of the equivalence holds. Likewise, the antidomain of x is the greatest (anti)domain element for which the left-annihilation in the left-hand side of the equivalence holds. By (12), one cannot sequentially compose two programs if the first doesn't execute into states from which the second can be executed. Finally, (13) adds the missing second complementation property to the second AKA axiom.

9.1.2. Boolean Subalgebra of Domain and Range Elements. The fact that $\mathsf{d}^2 = \mathsf{d}$ holds in AKA has important consequences. Let $\mathsf{d}(K)$ denote the image of K under d, as in Chapter 3. We call this set the set of *domain elements* of K.

LEMMA 9.7. *Let K be an* AKA*. Then, for all $x \in K$,*

$$\mathsf{d}\, x = x \Leftrightarrow x \in \mathsf{d}(K).$$

PROOF. Suppose $\mathsf{d}\, x = x$. Then there is trivially some $y \in K$ such that $x = \mathsf{d}\, y$ (namely x) and therefore $x \in \mathsf{d}(K)$. Suppose $x \in \mathsf{d}[K]$. Then $x = \mathsf{d}\, y$ for some $y \in K$ and therefore $\mathsf{d}\, x = \mathsf{d}^2\, y = \mathsf{d}\, y = x$ by Lemma 9.6(6). □

We can thus define

$$K_{\mathsf{d}} = \{x \in K \mid \mathsf{d}\, x = x\},$$

the set of all fixpoints of d in K, and know that $K_{\mathsf{d}} = \mathsf{d}(K)$. The domain elements are precisely the fixpoints of d.

REMARK 9.8. Interestingly, $K_{\mathsf{d}} = \mathsf{ad}(K)$ holds as well: $\mathsf{ad}(K) \subseteq \mathsf{d}(K)$ because $\mathsf{ad}\, x = \mathsf{d}\,(\mathsf{ad}\, x) \in \mathsf{d}(K)$ for every $x \in K$ by Lemma 9.6(6), and $\mathsf{d}(K) \subseteq \mathsf{ad}(K)$ because $\mathsf{d}\, x = \mathsf{ad}\,(\mathsf{ad}\, x) \in \mathsf{ad}(K)$ for every $x \in K$, by definition of d.

We can therefore write $\mathsf{d}\, x = x$ to express that x is a domain element in $\mathsf{d}(K)$ as we could write $\tau\, x = x$ to express than an element in a test in a KAT.

With these preparations we can show that the domain algebra K_{d} has the desired structure.

PROPOSITION 9.9. *Let K be an* AKA*. Then $(K_{\mathsf{d}}, \cdot, +, \mathsf{ad}, 0, 1)$ forms a boolean subalgebra of K.*

PROOF. We need to prove two properties. First, domain elements in K_{d} must be closed under the operations $\cdot$, $+$ and ad; and 0 and 1 must be domain elements. Second, elements in K_{d} must satisfy the axioms of boolean algebra.

Closure under the operations is immediate from the properties in Lemma 9.6. Closure with respect to $\cdot$ follows from

$$\mathsf{d}\, x \cdot \mathsf{d}\, y = \mathsf{d}\,(\mathsf{d}\, x \cdot y) = \mathsf{d}\,(\mathsf{d}\, x \cdot \mathsf{d}\, y),$$

that of $+$ from

$$\mathsf{d}\, x + \mathsf{d}\, y = \mathsf{d}\,(\mathsf{d}\, x) + \mathsf{d}\,(\mathsf{d}\, y) = \mathsf{d}\,(\mathsf{d}\, x + \mathsf{d}\, y),$$

that of complementation from

$$\mathsf{ad}\,(\mathsf{d}\, x) = \mathsf{d}\,(ad\, x).$$

Finally, $\mathsf{d}\, 0 = 0$ and $\mathsf{d}\, 1 = 1$ show that 0 and 1 are in K_{d}. Thus K_{d} forms a subdioid of K; even a sub-Kleene algebra: $(\mathsf{d}\, x)^* = 1$ because $\mathsf{d}\, x \leq 1$ by Lemma 9.6(3) and $y^* = 1$ for all $y \leq 1$ by Lemma 2.16(11).

Some of the boolean algebra axioms (see Definition 2.22) are simply dioid axioms; others are again consequences of properties from Lemma 9.6. Addition is associative, commutative and idempotent in every dioid; multiplication is associative in every dioid, and commutative and idempotent on K_{d} by Lemma 9.6. The identity laws $x + 0 = 0$ and $x \cdot 0 = 0$ for joins and meets are dioid axioms as well. This gives us two semilattices.

Next we check the absorption laws $\mathsf{d}\, x + \mathsf{d}\, x \cdot \mathsf{d}\, y = \mathsf{d}\, x$ and $\mathsf{d}\, x \cdot (\mathsf{d}\, x + \mathsf{d}\, y) = \mathsf{d}\, x$. For the first one,

$$\mathsf{d}\, x + \mathsf{d}\, x \cdot \mathsf{d}\, y = \mathsf{d}\, x \cdot 1 + \mathsf{d}\, x \cdot \mathsf{d}\, y = \mathsf{d}\, x \cdot (1 + \mathsf{d}\, y) = \mathsf{d}\, x \cdot 1 = \mathsf{d}\, x.$$

For the second one, using the first absorption law,

$$\mathsf{d}\, x \cdot (\mathsf{d}\, x + \mathsf{d}\, y) = \mathsf{d}\, x \cdot \mathsf{d}\, x + \mathsf{d}\, x \cdot \mathsf{d}\, y = \mathsf{d}\, x + \mathsf{d}\, x \cdot \mathsf{d}\, y = \mathsf{d}\, x.$$

This gives us a lattice.

Next we check the distributivity laws $\mathsf{d}\, x \cdot (\mathsf{d}\, y + \mathsf{d}\, z) = \mathsf{d}\, x \cdot \mathsf{d}\, y + \mathsf{d}\, x \cdot \mathsf{d}\, z$ and $\mathsf{d}\, x + \mathsf{d}\, y \cdot \mathsf{d}\, z = (\mathsf{d}\, x + \mathsf{d}\, y) \cdot (\mathsf{d}\, x + \mathsf{d}\, z)$. The first holds in every semiring. We can use it to derive the second one:

$$\begin{aligned}(\mathsf{d}\, x + \mathsf{d}\, y) \cdot (\mathsf{d}\, x + \mathsf{d}\, z) &= (\mathsf{d}\, x + \mathsf{d}\, y) \cdot \mathsf{d}\, x + (\mathsf{d}\, x + \mathsf{d}\, y) \cdot \mathsf{d}\, z \\ &= \mathsf{d}\, x + (\mathsf{d}\, x + \mathsf{d}\, y) \cdot \mathsf{d}\, z \\ &= \mathsf{d}\, x + \mathsf{d}\, x \cdot \mathsf{d}\, z + \mathsf{d}\, y \cdot \mathsf{d}\, z \\ &= \mathsf{d}\, x + \mathsf{d}\, y \cdot \mathsf{d}\, z,\end{aligned}$$

using notably absorption. (This proof can be generalised to showing that any lattice satisfying one of the two distributivity laws satisfies the other, too.)

The complementation axiom $ad\, x + \mathsf{d}\, x = 1$ is an axiom of AKA; $ad\, x \cdot \mathsf{d}\, x = 0$ is part of Lemma 9.6. □

The boolean algebra is not an arbitrary subalgebra of the algebra of subidentities in K, as it is for KAT. Proving this requires a definition and a technical lemma.

By analogy to the definition of complemented lattices in Section 2.4.1, call an element of $x \in K$ *complemented* if $x + y = 1$, $xy = 0$ and $yx = 0$ holds for some $y \in K$, which accordingly is a *complement* of x in K. By definition, all complemented elements are subidentities.

LEMMA 9.10. *The set B_K of complemented elements of an AKA K is a boolean subalgebra of K.*

PROOF. The elements of B_K satisfy the dioid axioms, so it remains to check commutativity and idempotency of $\cdot$ and the absorption laws. As domain and antidomain elements are complemented, the proofs are very similar to those in Proposition 9.9.

For idempotency $xx = x$ if x has complement x',

$$x = x(x + x') = xx + xx' = xx + 0 = xx.$$

The proof of $\mathsf{d}\, x \cdot \mathsf{d}\, x = \mathsf{d}\, x$ in Lemma 9.6 proceeds along the same lines.

For the absorption laws for complemented elements, we can replay the proofs in Proposition 9.9. I show the first one as an example:

$$x + xy = x1 + xy = x(1 + y) = x1 = x.$$

For commutativity $xy = yx$ of complemented elements,

$$\begin{aligned} xy &= (x + yx)(y + yx) \\ &= xy + xyx + yxy + yxyx \\ &= yx + yxy + xyx + xyxy \\ &= (y + xy)(x + xy) \\ &= yx. \end{aligned}$$

Once again, commutativity of domain and antidomain elements in Lemma 9.6 can be proved along the same lines. The distributive lattice axioms therefore hold, and hence the boolean algebra axioms, because elements are complemented.

Finally we consider closure properties. Here we need to take an approach that differs from that in Proposition 9.9. First we show that sums of complemented elements are complemented. So suppose x and y be complemented with complements x' and y'. Then

$$x + y + x'y' = (x + y + x')(x + y + y') = (1 + y)(1 + x) = 11 = 1$$

and

$$(x + y)x'y' = xx'y' + yx'y' = 0 + 0 = 0.$$

Next we show that products of complemented elements are complemented. Suppose again x and y are complemented with complements x' and y'. Then

$$xy + (x' + y') = (x + x' + y')(y + x' + y') = (1 + y')(1 + x') = 11 = 1$$

and

$$xy(x' + y') = xyx' + yx'y' = 0 + 0 = 0.$$

Moreover, by symmetry of the definition, all complements of complemented elements are complemented, and of course 0 and 1 are mutual complements.

This shows that B_K is indeed a complemented distributive lattice and hence a boolean subalgebra of K, according to Section 2.4.1. □

PROPOSITION 9.11. *Let K be an* AKA. *Then K_d forms the largest boolean subalgebra of K bounded by* 0 *and* 1.

PROOF. By Lemma 9.10, the complemented elements form a boolean subalgebra of K bounded by 0 and 1. It is the largest boolean algebra between 0 and 1 by definition. Suppose $x \in B_K$ has complement y. Then

$$x = xx = x \cdot \mathsf{d}\, x \cdot x \leq x\mathsf{d}\, x \leq x,$$

that is $x = x\mathsf{d}\, x$, and

$$y\mathsf{d}\, x = \mathsf{d}\,(y \cdot \mathsf{d}\, x) \cdot y \cdot \mathsf{d}\, x = \mathsf{d}\,(y \cdot x) \cdot y \cdot \mathsf{d}\, x = \mathsf{d}\, 0 \cdot y \cdot \mathsf{d}\, x = 0 \cdot y \cdot \mathsf{d}\, x = 0.$$

Then $\mathsf{d}\, x = (x + y)\mathsf{d}\, x = x$ and therefore $x \in \mathsf{d}(K)$. This shows that $B_K \subseteq \mathsf{d}(K)$ and $B_K = \mathsf{d}(K)$ follows. □

Note that Proposition 9.11 subsumes Proposition 9.9, because Lemma 9.10 performs the proof steps for Proposition 9.9 in a more general setting. Nevertheless the proof of Proposition 9.9 is more direct.

A dual property holds for antirange Kleene algebras and the set

$$K_\mathsf{r} = \{x \mid \mathsf{r}\, x = x\}$$

by opposition. Consequently, K_{d} and K_{r} coincide in any MKA by maximality, but we prove this result directly.

LEMMA 9.12. *In every* MKA,

(1) $\mathsf{d} \circ \mathsf{r} = \mathsf{r}$,
(2) $\mathsf{r} \circ \mathsf{d} = \mathsf{d}$.

PROOF. Exercise. □

PROPOSITION 9.13. *Let K be a* MKA. *Then $K_{\mathsf{d}} = K_{\mathsf{r}}$.*

PROOF. If $\mathsf{d}\, x = x$, then $\mathsf{r}\, x = \mathsf{r}\,(\mathsf{d}\, x) = \mathsf{d}\, x = x$. Dually, $\mathsf{d}\, x = \mathsf{d}\,(\mathsf{r}\, x) = \mathsf{r}\, x = x$ follows from $\mathsf{r}\, x = x$. Hence $\mathsf{d}\, x = x \Leftrightarrow \mathsf{r}\, x = x$ and therefore $x \in K_{\mathsf{d}} \Leftrightarrow x \in K_{\mathsf{r}}$. □

As for KAT we write $p, q, r, \ldots$ for elements of K_{d} and $\overline{p}$ instead of $\mathsf{ad}\, p$. The development for MKA so far seems much more complicated than for KAT. Yet defining an antitest operation of type $K \to K$ to obtain tests, like in the Isabelle theories, makes the effort comparable.

9.1.3. Relational and State Transformer Model. Next we link MKAs with the relational and state transformer semantics of programs.

PROPOSITION 9.14. *The structures*

$$(\mathsf{Rel}\, X, ;, \cup, \emptyset_X, Id_X, ^*, \mathsf{ad}, \mathsf{ar}) \quad \textit{and} \quad (\mathsf{Sta}\, X, \circ_K, +, 0_X, \eta_X, ^{*_K}, \mathsf{ad}, \mathsf{ar})$$

form MKA*s with*

$$\mathsf{ad}\, R = \{(a,a) \mid \neg\exists b.\ (a,b) \in R\}, \qquad \mathsf{ar}\, R = \{(b,b) \mid \neg\exists a.\ (a,b) \in R\},$$

$$\mathsf{ad}\, f\, a = \begin{cases} \{a\} & \textit{if } f\, a = \emptyset, \\ \emptyset & \textit{otherwise}, \end{cases} \qquad \mathsf{ar}\, f\, b = \begin{cases} \emptyset & \textit{if } \exists a.\ b \in f\, a, \\ \{b\} & \textit{otherwise}. \end{cases}$$

PROOF. Relative to Proposition 4.6 and 4.18 we must check the antidomain and antirange axioms of MKA. This can be done with Isabelle. Soundness of the antirange axioms follows from opposition. □

Explicit formulas for domain and range operations are immediate consequences of the definitions for antidomain and antirange.

$$\mathsf{d}\, R = \{(a,a) \mid \exists b.\ (a,b) \in R\}, \qquad \mathsf{r}\, R = \{(b,b) \mid \exists a\ (a,b) \in R\},$$

$$\mathsf{d}\, f\, a = \begin{cases} \emptyset & \text{if } f\, a = \emptyset, \\ \{a\} & \text{otherwise}, \end{cases} \qquad \mathsf{r}\, f\, b = \begin{cases} \{b\} & \text{if } \exists a.\ b \in f\, a, \\ \emptyset & \text{otherwise}. \end{cases}$$

9.1.4. Modal Operators. Terms such as $-dom\,(R\,;\overline{P})$ and $-ran(\overline{P}\,;R)$, as discussed at the beginning of this Chapter, can now be expressed algebraically as $\mathsf{ad}\,(x \cdot \mathsf{ad}\, y)$ and $\mathsf{ar}\,(\mathsf{ar}\, y \cdot x)$. We introduce, in any MKA, and for all $x \in K$ and $p \in K_{\mathsf{d}}$, the following notation:

$$|x\rangle p = \mathsf{d}\,(x \cdot p), \qquad \langle x| p = \mathsf{r}\,(p \cdot x), \qquad |x] p = \mathsf{ad}\,(x \cdot \mathsf{ad}\, p), \qquad [x| p = \mathsf{ar}\,(\mathsf{ar}\, p \cdot x).$$

The functions $|-\rangle$, $\langle -|$, $|-]$ and $[-|$ of type $K \to K \to K$, or strictly speaking $K \to K_{\mathsf{d}} \to K_{\mathsf{d}}$, are *modal operators*: $|-\rangle$ is a *forward diamond*, $\langle -|$ a *backward diamond*, $|-]$ a *forward box* and $[-|$ a *backward box* operator. In program verification, $|-]$ is also known as *weakest liberal precondition* (*wlp*) operator. More generally,

the functions $|x\rangle$, $\langle x|$, $|x]$ and $[x|$ of type $K_{\mathsf{d}} \to K_{\mathsf{d}}$ are examples of *predicate transformers*; they are endofunctions on the boolean algebra K_{d} which, in the relational and state transformer models, transform predicates to predicates.

In the relational model,

$$|R\rangle Q = \{(s,s) \mid \exists s'.\ (s,s') \in R \wedge (s',s') \in Q\},$$
$$|R]Q = \{(s,s) \mid \forall s'.\ (s,s') \in R \Rightarrow (s',s') \in Q\}$$

and, by opposition,

$$\langle RP = \{(s,s) \mid \exists s'.\ (s',s) \in R \wedge (s',s') \in P\},$$
$$[R|P = \{(s,s) \mid \forall s'.\ (s',s) \in R \Rightarrow (s',s') \in P\}.$$

This shows that $|R\rangle Q$ is the relational preimage of Q under R, and $\langle R|P$ the relational image of P under R. This is consistent with the standard Kripke semantics of modal logics, where propositions can have different truth values in different states of a system or *possible worlds* and R models accessibility between worlds. Explaining further details of modal logics is beyond the scope of these lectures. The modal logic most closely related to MKA is *propositional dynamic logic*, which has been designed specifically as a logic for reasoning about while programs such as Imp. MKA can be seen as a semantic analogue. State transformer semantics are less common in traditional modal logics, and we leave the derivation of expressions similar to those above as an exercise.

9.1.5. Symmetries and Dualities. The modal operators satisfy interesting symmetries and dualities in MKA, which we examine next.

By definition, the forward and backward operators are related by opposition. In the relational model, this can be expressed explicitly by relational converse:

$$\begin{aligned} |R\rangle P &= \{a \mid \exists b.\ (a,b) \in R \wedge P\, b\} \\ &= \{a \mid \exists b.\ (b,a) \in R^{\smile} \wedge P\, b\} \\ &= \langle R^{\smile}|P, \end{aligned}$$

$$\begin{aligned} |R]P &= \{a \mid \forall b.\ (a,b) \in R \Rightarrow P\, b\} \\ &= \{a \mid \forall b.\ (b,a) \in R^{\smile} \Rightarrow P\, b\} \\ &= [R^{\smile}|P, \end{aligned}$$

from which $\langle R|P = |R^{\smile}\rangle P$ and $[R|P = |R^{\smile}]P$ follow by opposition.

Boxes and diamonds are related by De Morgan duality.

LEMMA 9.15. *Let K be a* MKA. *For all $x \in K$ and $p \in K_{\mathsf{d}}$,*

(1) $\overline{|x]p} = |x\rangle\overline{p}$ *and* $\overline{|x\rangle p} = |x]\overline{p}$,
(2) $\overline{[x|p} = \langle x|\overline{p}$ *and* $\overline{\langle x|p} = [x|\overline{p}$.

PROOF. Exercise. □

Proving additional symmetries of modal operators requires two lemmas. The first one is similar to Lemma 2.31.

LEMMA 9.16. *In every* MKA *K, for all $x \in K$ and $p, q \in K_{\mathsf{d}}$, the following identities are equivalent.*

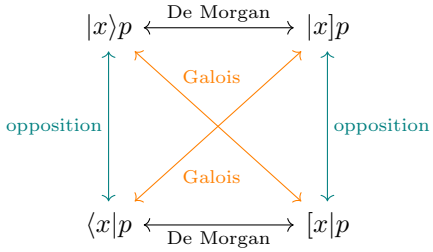


FIGURE 1. Symmetries between Modal Operators

(1) $px \leq xq$,
(2) $px\overline{q} = 0$.
(3) $x\overline{q} \leq \overline{p}x$.

PROOF. Let $px \leq xq$. Then $px\overline{q} \leq xq\overline{q} = 0$.
Let $p\overline{q} = 0$. Then $x\overline{q} = (p + \overline{p})x\overline{q} = \overline{p}x\overline{q} \leq \overline{p}x$.
Let $x\overline{q} \leq \overline{p}x$. Then $px\overline{q} \leq p\overline{p}x = 0$. □

LEMMA 9.17. *In every* MKA K, *for all* $x \in K$ *and* $p, q \in K_{\mathsf{d}}$,

(1) $|x\rangle p \leq q \Leftrightarrow xp \leq qx$ *and* $\langle x|p \leq q \Leftrightarrow px \leq xq$,
(2) $p \leq |x]q \Leftrightarrow px \leq xq$ *and* $p \leq [x|q \Leftrightarrow xp \leq qx$.

PROOF.

(1) Suppose $|x\rangle p \leq q$. Then $\mathsf{d}\,(xp) \leq q$ and $xp = \mathsf{d}\,(xp) \cdot (xp) \leq qxp \leq qx$. Suppose $xp \leq qx$. Then $|x\rangle p = \mathsf{d}\,(xp) \leq \mathsf{d}\,(qx) = q\mathsf{d}\,x \leq q$. The second equivalence follows by opposition.
(2) We calculate

$$p \leq |x]q \Leftrightarrow p \leq \overline{|x\rangle\overline{q}} \Leftrightarrow |x\rangle\overline{q} \leq \overline{p} \Leftrightarrow x\overline{q} \leq \overline{p}x \Leftrightarrow px \leq xq.$$

The second equivalence follows by opposition. □

Using these properties we can show that box and diamond operators are adjoints in a Galois connection.

LEMMA 9.18. *Let* K *be a* MKA. *For all* $x \in K$ *and* $p, q \in K_{\mathsf{d}}$,

(1) $|x\rangle p \leq q \Leftrightarrow p \leq [x|q$,
(2) $\langle x|p \leq q \Leftrightarrow p \leq |x]q$.

PROOF. Immediate from Lemma 9.17. □

REMARK 9.19. Let $(P_1, \leq_1)$ and $(P_2, \leq_2)$ be posets. The functions $f : P_1 \to P_2$ and $g : P_2 \to P_1$ are *adjoints in a Galois connection* if

$$f\,x \leq_2 y \Leftrightarrow x \leq_1 g\,y$$

holds for each $x \in P_2$ and $y \in P_1$. More specifically, f is called the *left adjoint* and g the *right adjoint* of the Galois connection. They satisfy a number of algebraic properties, which may depend on additional structure present in P_1 and P_2. Explaining them is beyond the scope of these lecture notes.

Finally, the following conjugation properties hold.

LEMMA 9.20. *Let* K *be a* MKA. *For all* $x \in K$ *and* $p, q \in K_{\mathsf{d}}$,

(1) $\langle x|p \cdot q = 0 \Leftrightarrow p \cdot |x\rangle q = 0$,
(2) $[x|p + q = 1 \Leftrightarrow p + |x]q = 1$.

PROOF.

(1) $\langle x|p \cdot q = 0 \Leftrightarrow \langle x|p \leq \overline{q} \Leftrightarrow q \leq \overline{\langle x|p} \Leftrightarrow q \leq [x|\overline{p} \Leftrightarrow |x\rangle q \leq \overline{p} \Leftrightarrow p \cdot |x\rangle q = 0$.
(2) $[x|p + q = 1 \Leftrightarrow \langle x|\overline{p} \cdot \overline{q} = 0 \Leftrightarrow \overline{p} \cdot |x\rangle \overline{q} = 0 \Leftrightarrow p + |x]q = 1$.

□

Conjugation properties express opposition in the absence of converse.

9.2. Formalising Modal Kleene Algebras

Declaring the type classes for antidomain, antirange and modal Kleene algebras presents no surprises.

```
class antidomain-kleene-algebra = kleene-algebra +
  fixes ad :: 'a ⇒ 'a
  assumes ad-annil [simp]: ad x · x = 0
  and ad-local-sub [simp]: ad (x · y) ≤ ad (x · ad (ad y))
  and ad-compl1 [simp]: ad (ad x) + ad x = 1

begin

definition dom-op :: 'a ⇒ 'a (do) where
  do x = ad (ad x)

definition fdia :: 'a ⇒ 'a ⇒ 'a where
  fdia x y = do (x · y)

definition fbox :: 'a ⇒ 'a ⇒ 'a where
  fbox x y = ad (x · ad y)

end

class antirange-kleene-algebra = kleene-algebra +
  fixes ar :: 'a ⇒ 'a
  assumes ar-annil [simp]: x · ar x = 0
  and ar-local-sub [simp]: ar (x · y) ≤ ar (ar (ar x) · y)
  and ar-compl1 [simp]: ar (ar x) + ar x = 1

begin

definition range-op :: 'a ⇒ 'a (ra) where
  ra x = ar (ar x)

definition bdia :: 'a ⇒ 'a ⇒ 'a where
  bdia x y = ra (y · x)

definition bbox :: 'a ⇒ 'a ⇒ 'a where
  bbox x y = ar (ar y · x)

end

class modal-kleene-algebra = antidomain-kleene-algebra + antirange-kleene-algebra
```

More interestingly, we can formalise opposition duality as sublocale statements.

```
sublocale antirange-kleene-algebra ⊆ op-arka:
  antidomain-kleene-algebra (+) 0 1 λx y. y · x (≤) (<) star ar
  rewrites op-arka.dom-op x = ra x
  and op-arka.fdia x y = bdia x y
  and op-arka.fbox x y = bbox x y
proof −
  show class.antidomain-kleene-algebra (+) 0 1 (λx y. y · x) (≤) (<) star ar
    by unfold-locales (simp-all add: mult-assoc distr distl star-inductl star-inductr)
  then interpret op-arka:
    antidomain-kleene-algebra (+) 0 1 (λx y. y · x) (≤) (<) star ar.
  show op-arka.dom-op x = ra x
    by (simp add: range-op-def op-arka.dom-op-def)
  show op-arka.fdia x y = bdia x y
    by (simp add: bdia-def range-op-def op-arka.dom-op-def op-arka.fdia-def)
  show op-arka.fbox x y = bbox x y
    by (simp add: bbox-def op-arka.fbox-def)
qed

sublocale antidomain-kleene-algebra ⊆ arka-op:
  antirange-kleene-algebra (+) 0 1 (λx y. y · x) (≤) (<) star ad
  rewrites arka-op.range-op x = do x
  and arka-op.bdia x y = fdia x y
  and arka-op.bbox x y = fbox x y
proof −
  show class.antirange-kleene-algebra (+) 0 1 (λx y. y · x) (≤) (<) star ad
    by unfold-locales (simp-all add: mult-assoc distl distr star-inductl star-inductr)
  then interpret arka-op:
    antirange-kleene-algebra (+) 0 1 (λx y. y · x) (≤) (<) star ad.
  show arka-op.range-op x = do x
    by (simp add: arka-op.range-op-def dom-op-def)
  show arka-op.bdia x y = fdia x y
    by (simp add: arka-op.bdia-def arka-op.range-op-def dom-op-def fdia-def)
  show arka-op.bbox x y = fbox x y
    by (simp add: arka-op.bbox-def fbox-def)
qed
```

The opposites of all facts proved for domain and antidomain or forward boxes and diamonds in antidomain Kleene algebras become automatically available in antirange Kleene algebras through proving this theorem. This simplifies theory engineering considerably.

9.3. Predicate Transformers and Structural Verification Conditions

For verification purposes, we need in particular the forward box or *wlp* operator $|-]-$. In the relational and state transformer semantics,

$$|R]Q = \{a \mid \forall b.\ (a,b) \in R \Rightarrow Q\,a\} \quad \text{and} \quad |f]Q = \{a \mid f\,a \subseteq Q\},$$

where we identify sets, predicates and subidentity relations, as usual. We have already mentioned that this models the set of all states from which one must end

up in states satisfying Q when executing R or f, respectively. This explains why $|-]-$ is called *weakest liberal precondition* operator.

We can now express partial correctness specifications more generally in AKA as

$$p \leq |x]q.$$

Such specifications can now be verified in two steps:

(1) compute the *wlp* of program x and postcondition q,
(2) show that precondition p lies below this *wlp*.

The following facts are useful for calculating *wlp*s in the first step.

PROPOSITION 9.21. *Let K be an* AKA*. For all $x, y \in K$ and $p, q \in K_{\mathsf{d}}$,*

(fbox-seq) $$|xy]q = |x]|y]q,$$

(fbox-cond) $$|\textbf{if } p \textbf{ then } x \textbf{ else } y]q = (\overline{p} + |x]q)(p + |y]q),$$

(fbox-cond-var) $$|\textbf{if } p \textbf{ then } x \textbf{ else } y]q = p|x]q + \overline{p}|y]q,$$

(fbox-while-inv) $$p \leq i \wedge it \leq |x]i \wedge i\overline{t} \leq q \Rightarrow p \leq |\textbf{while } t \textbf{ inv } i \textbf{ do } x]q.$$

PROOF. Exercise. □

For straight-line programs, verification conditions for the control structure of programs can thus be computed by equational reasoning and ultimately simplification with Isabelle.

The following laws are used in Section 9.6 below.

LEMMA 9.22. *Let K be an* AKA*. For all $x, y \in K$ and $p, q \in K_d$,*

(1) $p \cdot |\textbf{if } p \textbf{ then } x \textbf{ else } y]q = p|x]q$,
(2) $\overline{p} \cdot |\textbf{if } p \textbf{ then } x \textbf{ else } y]q = \overline{p} \cdot |y]q$,
(3) $p \cdot |\textbf{while } p \textbf{ do } x]q = p|x](|\textbf{while } p \textbf{ do } x]q)$,
(4) $\overline{p} \cdot |\textbf{while } p \textbf{ do } x]q = \overline{p} \cdot q$.

PROOF. For (3) and (4) we use the identity

$$\textbf{while } p \textbf{ do } x = \textbf{if } p \textbf{ then } (x \cdot \textbf{while } p \textbf{ do } x) \textbf{else } 1,$$

which we have proved in Lemma 2.34 for KAT. We have verified it in AKA with Isabelle and do not repeat its proof.

(1) We calculate

$$\begin{aligned} p \cdot |\textbf{if } p \textbf{ then } x \textbf{ else } y]q &= p(\overline{p} + |x]q)(p + |y]q) \\ &= p|x]q(p + |y]q) \\ &= p|x]q. \end{aligned}$$

The first step uses (fbox-cond), the second one simple boolean algebra, and the third one in particular commutativity of meet and the absorption law of boolean algebra (the second equation in Definition 2.22).

(2) The proof is similar to (1) and left as an exercise.
(3) We calculate

$$p \cdot |\textbf{while } p \textbf{ do } x]q = p \cdot |\textbf{if } p \textbf{ then } (x \cdot \textbf{while } p \textbf{ do } x) \textbf{else } 1]q = p \cdot |x]q,$$

using (1) and idempotency of meet in boolean algebra.

(4) We calculate, along similar lines,

$$\begin{aligned}\overline{p} \cdot |\textbf{while } p \textbf{ do } x]q &= \overline{p} \cdot |\textbf{if } p \textbf{ then } (x \cdot \textbf{while } p \textbf{ do } x) \textbf{else } 1]q \\ &= \overline{p} \cdot |1]q \\ &= \overline{p}q.\end{aligned}$$

□

Once again it is helpful to derive specific structural *wlp*-laws for $\mathsf{Rel}\, X$ and $\mathsf{Sta}\, X$ that make predicates available for Isabelle. This time we only show the state transformer semantics and refer to our Isabelle theories for further details.

abbreviation
scond :: *'a pred* ⇒ *'a sta* ⇒ *'a sta* ⇒ *'a sta* (*sif - then - else - fi* [*64*,*64*,*64*] *63*) **where**
sif P then f else g fi ≡ *sta-aka.cond* ⌈*P*⌉$_s$ *f g*

abbreviation *swhile* :: *'a pred* ⇒ *'a sta* ⇒ *'a sta* (*swhile - do - od* [*64*,*64*] *63*) **where**
swhile P do f od ≡ *sta-aka.while* ⌈*P*⌉$_s$ *f*

abbreviation
swhile-inv :: *'a pred* ⇒ *'a pred* ⇒ *'a sta* ⇒ *'a sta*
(*swhile - inv - do - od* [*64*,*64*,*64*] *63*) **where**
swhile P inv I do f od ≡ *sta-aka.while-inv* ⌈*P*⌉$_s$ ⌈*I*⌉$_s$ *f*

abbreviation *sfbox R Q* ≡ ⌊*sta-aka.fbox R* ⌈*Q*⌉$_s$⌋$_s$

The coercion function $\lfloor - \rfloor_s$ maps state transformers to predicates, and we have defined an analogous function $\lfloor - \rfloor_r$ for relations.

We can now derive the *wlp* laws for state transformers mentioned.

lemma *sfbox-unfold*: *sfbox f P s* = (∀ *s'*. *s'* ∈ *f s* ⟶ *P s'*)
⟨*Proof*⟩

lemma *sfbox-seq* [*simp*]: *sfbox* (*f* ∘$_K$ *g*) *P s* = *sfbox f* (*sfbox g P*) *s*
⟨*Proof*⟩

lemma *sfbox-seq-var*:
assumes ∀ *s*. *w s* ⟶ *sfbox y z s*
and ∀ *s*. *v s* ⟶ *sfbox x w s*
shows ∀ *s*. *v s* ⟶ *sfbox* (*x* ∘$_K$ *y*) *z s*
⟨*Proof*⟩

lemma *sfbox-cond* [*simp*] :
sfbox (*sif P then f else g fi*) *Q s* = ((*P s* ⟶ *sfbox f Q s*) ∧ (¬ *P s* ⟶ *sfbox g Q s*))
⟨*Proof*⟩

lemma *sfbox-cond-var*:
sfbox (*sif P then f else g fi*) *Q s* = ((*P s* ∧ *sfbox f Q s*) ∨ (¬ *P s* ∧ *sfbox g Q s*))
⟨*Proof*⟩

lemma *sfbox-while-inv*:
assumes ∀ *s*. *P s* ⟶ *I s*
and ∀ *s*. *I s* ⟶ ¬ *T s* ⟶ *Q s*

and $\forall\, s.\ I\ s \longrightarrow T\ s \longrightarrow \mathit{sfbox}\ f\ I\ s$
shows $\forall\, s.\ P\ s \longrightarrow \mathit{sfbox}\ (\mathit{swhile}\ T\ \mathit{inv}\ I\ \mathit{do}\ f\ \mathit{od})\ Q\ s$
⟨*Proof*⟩

lemma *sfbox-while-inv-break*:
assumes $\forall\, s.\ P\ s \longrightarrow \mathit{sfbox}\ g\ I\ s$
and $\forall\, s.\ I\ s \longrightarrow \neg\ T\ s \longrightarrow Q\ s$
and $\forall\, s.\ I\ s \longrightarrow\ \ T\ s \longrightarrow \mathit{sfbox}\ f\ I\ s$
shows $\forall\, s.\ P\ s \longrightarrow \mathit{sfbox}\ (g \circ_K (\mathit{swhile}\ T\ \mathit{inv}\ I\ \mathit{do}\ f\ \mathit{od}))\ Q\ s$
⟨*Proof*⟩

The laws *sfbox-seq-var* and *sfbox-while-inv-break* are needed because we have no equational laws for while-loops and must therefore break the chain of equational reasoning. In MKA, invariants satisfy $i \leq |x]i$—if they hold before the execution of a program, then they must hold afterwards. Alternatively one can use *sfbox-while-inv* in combination with *sfbox-seq-var* or else the macro law *sfbox-while-inv-break*, which has been derived from *sfbox-while-inv* using *sfbox-seq-var*.

9.4. Integrating the Program Store

Integrating the simple program store from Chapter 7 and computing the *wlp*s for assignments is now straightforward and completely compositional with respect to the algebra and the relational and state transformer semantics.

PROPOSITION 9.23. *In* $\mathsf{Rel}\, S$ *or* $\mathsf{Sta}\, S$*, where* $S = D^V$*,*

(fbox-assign) $$|v := e]Q = \lambda s.\ Q\,(\mathit{set}\ v\ e\ s).$$

In fact, with Isabelle, we have proved the following equations.

lemma *mka-rel-assign* [*simp*]: *rel-aka.fbox* $(v :=_r e)\ \lceil Q\rceil_r = \lceil Q \circ (\mathit{set}\ v\ e)\rceil_r$
⟨*Proof*⟩

lemma *mka-sta-assign* [*simp*]: *sta-aka.fbox* $(v :=_s e)\ \lceil Q\rceil_s = \lceil Q \circ (\mathit{set}\ v\ e)\rceil_s$
⟨*Proof*⟩

lemma *rfbox-assign* [*simp*]: *rfbox* $(v :=_r e)\ Q\ = Q \circ (\mathit{set}\ v\ e)$
⟨*Proof*⟩

lemma *sfbox-assign* [*simp*]: *sfbox* $(v :=_s e)\ Q\ = Q \circ (\mathit{set}\ v\ e)$
⟨*Proof*⟩

We can now verify simple while programs. Overall, the set up for relations and state transformers is identical and should therefore lead to the same data level verification conditions. This must of course be tested in practice by example.

9.5. Examples: Program Verification with Predicate Transformers

We present the usual verification examples: variable swap, maximum of two numbers and integer division. This time we have not programmed syntactic sugar for partial correctness specifications like for Hoare logic. We simply show that preconditions imply the *wlp*s of programs and their postconditions.

EXAMPLE 9.24. First we verify variable swap. We only show the state transformer partial correctness specification and proof. Those for relations are identical up to minor changes of syntax and can be found in our Isabelle theories.

```
lemma svar-swap:
 s ''x'' = m ∧ s ''y'' = n ⟹
   sfbox ((''z'' :=s (λs. s ''x''))∘K
   (''x'' :=s (λs. s ''y''))∘K
   (''y'' :=s (λs. s ''z'')))
  (λs. s ''x'' = n ∧ s ''y'' = m) s
 by simp
```

For straight-line programs, *simp* suffices for automated data-level verification condition generation. For such a simple program it can also discharge the resulting proof obligations. The proof in the relational semantics is also by *simp*. □

EXAMPLE 9.25. Next we verify the maximum-of-two-numbers program in the relational semantics. That for state transformers is identical up-to notation.

```
lemma rmaximum:
 ∀ s::int store.
  rfbox (rif (λs. s ''x'' ≥ s ''y'')
   then (''z'' :=r (λs. s ''x''))
   else (''z'' :=r (λs. s ''y''))
   fi)
  (λs. s ''z'' = max (s ''x'') (s ''y'')) s
 by force
```

This time, *simp* is too weak to discharge the data-level verification conditions, but *force* can still handle it. □

EXAMPLE 9.26. Our final examples verifies integer division in both semantics. In the first proof we use *rfbox-seq-var* to split the initialisation from the while loop and then *rfbox-while-inv* do deal with the loop. In the second one we use the derived rule *sfbox-while-inv-break* instead.

```
lemma rinteger-division:
∀ s::nat store. 0 < y ⟶
    rfbox ((''q'' :=r (λs. 0));
    (''r'' :=r (λs. x));
    (rwhile (λs. y ≤ s ''r'') inv (λs. x = s ''q'' * y + s ''r'')
     do
      (''q'' :=r (λs. s ''q'' + 1)) ;
      (''r'' :=r (λs. s ''r'' − y))
     od))
  (λs. x = s ''q'' * y + s ''r'' ∧ s ''r'' < y) s
  by (intro rfbox-seq-var rfbox-while-inv, auto simp: imp-refl)

lemma sinteger-division:
∀ s::nat store. 0 < y ⟶
    sfbox ((''q'' :=s (λs. 0)) ∘K
    (''r'' :=s (λs. x))  ∘K
    (swhile (λs. y ≤ s ''r'') inv (λs. x = s ''q'' * y + s ''r'')
```

```
  do
   (''q'' :=_s (λs. s ''q'' + 1)) ∘_K
   (''r'' :=_s (λs. s ''r'' − y))
  od))
(λs. x = s ''q'' * y + s ''r'' ∧ s ''r'' < y) s
by (rule sfbox-while-inv-break) simp-all
```

□

Overall the proofs with MKA and predicate transformers are noticeably simpler than those with KAT and Hoare logic. Yet tactics for Hoare logic would probably yield comparable proof automation.

9.6. Relative Completeness of Hoare Logic

We have so far separated KAT and MKA, in particular because that helps avoiding name clashes with Isabelle. But the following fact is easy to show.

PROPOSITION 9.27. *Every* AKA K *is a* KAT *with* $B = K_{\mathsf{d}}$ *and* $- = \mathsf{ad}$.

In particular, it is easy to check that ad satisfies the axioms for the antitest operation α in class *kat*. It also follows that every antirange Kleene algebra and therefore every MKA is a KAT. Validity of Hoare triples can therefore be expressed,

$$\mathsf{H}\, p\, x\, q \Leftrightarrow px \leq xq \Leftrightarrow p \leq |x]q,$$

and the rules of PHL be derived in AKA. Those of Hoare logic are then derivable in the relational and state transformer semantics of the program store.

We have already seen in Chapter 6 and 7 that the rules of Hoare logic are sound with respect to the relational and state transformer semantics of the program store. This simply means that they hold in concrete relational and predicate transformer semantics of the program store.

The additional expressivity of AKA and its models allow us to prove completeness of Hoare logic as well—yet only in a relative sense, which is typical for Hoare logic. Relative completeness arises because, in program verification, one often works with data domains such as numbers for which, by Gödel's incompleteness theorem, no deductive system is powerful enough to derive all statements that hold in the semantics. Completeness can therefore only be proved relative to implications $p \leq q$ between assertions that arise in the consequence rules. In computational parlance one assumes that some oracle supplies all consequences needed. Whether this is practically relevant is of course another question.

Relative to this data level incompleteness, completeness of Hoare logic means that the inference rules of Hoare logic suffice for deriving all valid partial correctness specifications $\mathsf{H}\, p\, x\, q$ for any pre- and postcondition p and q and any program x. The traditional approach aims to show that the rules allow deriving any Hoare triple of the form

$$\mathsf{H}\, (|x]q)\, x\, q,$$

while assuming that the oracle proves $p \leq |x]q$ and any data-level inequalities $p \leq p'$ and $q' \leq q$ in antecedents of (H-cons). We give a "semantic completeness proof"

PROPOSITION 9.28. *Let* K *be an* AKA *in which all elements are generated by* 0, 1 *and a set* $G \subseteq K$ *of basic commands, by sequential compositions, conditionals*

and while loops. Let K be expressive*, that is, $\mathsf{H}\,(|g]q)\,g\,q$ holds for all $g \in G$ and $q \in K_{\mathsf{d}}$. Then, for all $x \in K$ and $q \in K_d$,*

$$\mathsf{H}\,(|x]q)\,x\,q$$

is derivable using PHL*, while restricting any other use of* AKA *to the antecedents $p \leq p'$ and $q' \leq q$ of (H-cons).*

Proof. We proceed by induction on the structure of x. There are three base cases.

First, $x \in G$ is covered by expressivity. Second, if $x = 0$, then $|0]q = 1$ and we could add an axiom $H\,1\,0\,q$ to PHL for this case. Third, if $x = 1$, then $|1]q = q$ and therefore $\mathsf{H}\,(|1]q)\,1\,q = \mathsf{H}\,q\,1\,q$, which is (H-skip).

For $x = yz$, we have

$$\mathsf{H}(|y]|z]q)\,y\,(|z]q) \wedge \mathsf{H}\,(|z]q)\,z\,q \Rightarrow \mathsf{H}\,(|y]|z]q)\,x\,q \Rightarrow \mathsf{H}\,(|x]q)\,x\,q.$$

The initial conjunction consists of induction hypotheses. The first step uses (H-seq); the second one (H-cons) with $|y]|z]q = |yz]q$, which is (fbox-seq), as a hypothesis.

For $x =$ **if** p **then** y **else** z, we have

$$\mathsf{H}\,(|y]p)\,x\,q \wedge \mathsf{H}\,(|z]p)\,y\,q \Rightarrow \mathsf{H}\,(p|x]q)\,y\,q \wedge \mathsf{H}\,(\overline{p}|x]q)\,z\,q \Rightarrow \mathsf{H}\,(|x]q)\,x\,q.$$

The initial conjunction consists of induction hypotheses. The first step uses (H-cons) with hypotheses $p|x]q = p|y]q \leq |y]q$ and $\overline{p}|x]q = \overline{p}|z]q \leq |z]q$, which hold by Lemma 9.22(1), and (2). The second step uses (H-cond).

Finally, for $x =$ **while** p **do** y, we have

$$\begin{aligned}\mathsf{H}\,(|y]|x]q)\,y\,(|x]q) &\Rightarrow \mathsf{H}\,(p \cdot |x]q)\,y\,(|x]q)\\ &\Rightarrow \mathsf{H}\,(p \cdot |x]q)\,x\,(\overline{p}|x]q)\\ &\Rightarrow \mathsf{H}\,(p \cdot |x]q)\,x\,q.\end{aligned}$$

The initial Hoare triple is the induction hypothesis. The first and third step use (H-cond); the second one (H-while). The hypothesis of the first use of (H-cond) is $p \cdot |y]|x]q \leq p \cdot |x]q$, which holds by Lemma 9.22(3); the hypothesis of its second use is $\overline{p} \cdot |x]q = \overline{p} \cdot q \leq q$, which holds by Lemma 9.22(4). □

Lemma 9.29. *In the* AKA Rel S *or* Sta S*, the Hoare triple*

$$\mathsf{H}\,(|v := e)]Q)\,(v := e)\,Q$$

is derivable from (H-assign) for all subidentities Q, $v \in V$ and $e : S \to D$.

Proof.

$$\mathsf{H}\,(\lambda s.\ Q\,(\mathit{set}\,v\,e\,s))\,(v := e)\,Q \Rightarrow \mathsf{H}\,(|v := e)]Q)\,(v := e)\,Q,$$

where the initial Hoare triple is (H-assign) and (H-cons) is used in the step with hypothesis (fbox-assign). □

Theorem 9.30. *Hoare logic is relatively complete.*

Proof. By Proposition 9.28 and Lemma 9.29, $\mathsf{H}\,(|X]P)\,X\,P$ is derivable from instances of the rules of Hoare logic in the relational or state transformer semantics. An oracle allows deriving $P \to |X]Q$ and all implications $P' \to Q'$ arising from applications of (H-cons). If we had a syntax for programs and Hoare logic, our semantic proofs would correspond directly to the syntactic ones in Hoare logic, using substitutions in the latter instead of store updates in the former, as discussed in Chapter 7. □

9.7. KAT vs MKA

We have already argued that MKA is more expressive than KAT. Here is an example. In MKA we can show the reverse of (H-seq):

$$\forall x, y. \in K, p, q \in K_{\mathsf{d}}.\ \{p\}\, x\, \{q\} \Rightarrow \exists r \in K_{\mathsf{d}}.\ \{p\}\, x\, \{r\} \wedge \{r\}\, y\, \{q\},$$

using the encoding of Hoare triples from the previous section. Suppose $p \cdot x \cdot \overline{q} = 0$. We need an expression $r \in K_{\mathsf{d}}$ such that $p \cdot x \cdot \overline{r} = 0$ and $r \cdot y \cdot \overline{q} = 0$. Let $r = \mathsf{ad}\,(y \cdot \overline{q})$. Then $p \cdot x \cdot \overline{r} = p \cdot x \cdot d(y \cdot \overline{q})$, which equals 0 because of the assumption and Lemma 9.6(12), and $r \cdot y \cdot \overline{q} = a\,(y \cdot \overline{q}) \cdot y \cdot \overline{q} = 0$ by the first antidomain axiom.

However the same implication, with K_{d} replaced by B, does not hold in KAT. Consider the KAT $(\{0, \alpha, 1\}, \{0, 1\}, +, \cdot, \overline{(-)}, 0, 1, ^*)$ with $0 < \alpha < 1$, $\alpha^2 = 0$ and $\alpha^* = 1$. Then $\{1\}\alpha^2\{0\}$ holds because $1 \cdot 0 \cdot 1 = 0$, but one of $\{1\}\alpha\{r\}$ and $\{r\}\alpha\{0\}$ is false for $r \in \{0, 1\}$ (for $r = 0$, the first triple is false, for $r = 1$, the second one).

Note that $\mathsf{ad}\,(y \cdot \overline{q}) = |y]q$, which means nothing but that KAT cannot express the *wlp* operator.

Expressivity, however comes at a price. The equational theory of MKA is still decidable, but EXPTIME complete. Recall, by contrast, that the equational theory of KAT is PSPACE complete.