UTP by Example: Designs

Jim Woodcock and Simon Foster

University of York

Abstract. We present a tutorial introduction to the semantics of a basic nondeterministic imperative programming language in Unifying Theories of Programming (UTP). First, we give a simple relational semantics that accounts for a theory of partial correctness. Second, we give a semantics based on the theory of precondition-postcondition pairs, known in UTP as designs. This paper should be read in conjunction with the UTP book by Hoare & He. Our contribution lies in the large number of examples we introduce.

1 Introduction

Our paper is structured as follows. We give an overview of UTP in Sect. 2. We illustrate the ideas by constructing a UTP theory to capture Boyle's Law, which describes the relationship between the temperature, volume, and pressure of an ideal gas. We describe the meta-language used in UTP in Sect. 3. It is a point-wise variant of Tarski's alphabetised relational calculus. We introduce our nondeterministic imperative programming language in Sect. 4. We describe the semantics of the assignment, conditional, nondeterministic choice, and sequential composition statements. Before we can give a meaning to iteration and recursion, we need to cover some basic theory that underpins these constructs. In Sect. 5, we give an introduction to lattice theory, before returning in Sect. 6 to discuss recursion. We conclude our discussion of partial correctness in Sect. 7, by describing how the axioms of Hoare logic and the weakest precondition calculus can be validated by proving them as theorems in our relational semantics.

The second half of the paper deals with total correctness. Sect. 8 introduces the notion of a design: a precondition-postcondition pair embedded in the larger theory of relations. In Sect. 9, we describe the complete lattice of designs. We connect our two theories, relations and designs, by exhibiting in Sect. 10 a Galois connection that maps between them. Finally, we return to the theory of designs in Sect. 11, and show the two principal healthiness conditions that characterise the lattice.

In all these sections, we illustrate the ideas with a large number of examples.

2 Unifying Theories of Programming (UTP)

UTP is Hoare & He's long-term research agenda to provide a common basis for understanding the semantics of the modelling notations and programming

languages used in describing the behaviour of computer-based systems [45]. The technique they employ is to describe different modelling and programming paradigms in a common semantic setting: the alphabetised relational calculus. They isolate individual features of these languages in order to be able to emphasise commonalities and differences. They record formal links between the resulting theories, so that predicates from one theory can be translated into another, often as approximations. These links can also be used to translate specifications into designs and programs as part of a program development method.

UTP has been used to describe a wide variety of programming theories. In [45], Hoare & He formalise theories of sequential programming, with assertional reasoning techniques for both partial and total correctness; a theory of correct compilation; concurrent computation with reactive processes and communications; higher-order logic programming; and theories that link denotational, algebraic, and operational semantics.

Other contributions to UTP theories of programming language semantics, including: angelic nondeterminism [22, 23, 58]; aspect-oriented programming [25]; component systems [76]; event-driven programming [47, 77, 80]; lazy evaluation semantics [35]; object-oriented programming [18, 59, 63]; pointer-based programming [37]; probabilistic programming [43, 40, 64, 9, 79]; real-time programming [42, 38]; reversible computation [65, 64]; timed reactive programming [61, 62, 69, 60, 66]; and transaction programming [39, 40]. Individual programming languages have been given semantics in UTP. This includes the hardware description languages Handel-C [55, 56] and Verilog [78]; the multi-paradigm languages *Circus* [52, 13, 74, 53, 66] and CML [73, 70]; Safety-Critical Java [21, 19, 24, 54, 20]; and Simulink [17]. A wide variety of programming theories have been formalised in UTP, including confidentiality [6, 7]; general correctness [27, 29, 36, 28]; theories of testing [15, 67, 16]; hybrid systems; and theories of undefinedness [71, 5]. These are complemented by a collection of meta-theory, including work on higher-order UTP [75]; UTP and temporal-logic model checking [2]; and CSP as a retract of CCS [41].

Mechanisation is a key aspect of any formalisation, and UTP has been embedded in a variety of theorem provers, notably in ProofPower-Z and Isabelle [51, 50, 74, 10, 26, 12, 31, 33, 34]. This allows a theory engineer to mechanically construct UTP theories, experiment with them, prove properties, and eventually deploy them for use in program verification. In these notes we focus on our Isabelle embedding of the UTP called Isabelle/UTP [32].

UTP has its origins in the work on predicative programming, which was started by Hehner; see [44] for a summary. The UTP research agenda has as its ultimate goal to cover all the interesting paradigms of computing, including both declarative and procedural, hardware and software. It presents a theoretical foundation for understanding software and systems engineering, and has already been exploited in areas such as hardware [56, 80], hardware/software co-design [8] and component-based systems [76]. But it also presents an opportunity when constructing new languages, especially ones with heterogeneous paradigms and techniques.

Having studied the variety of existing programming languages and identified the major components of programming languages and theories, we can select theories for new, perhaps special-purpose languages. The analogy here is of a theory supermarket, where you shop for exactly those features you need while being confident that the theories plug-and-play together nicely.

Hoare & He define three axes for their classification of language semantics: (a) The first is by computational model, such as programming in the following styles: imperative, functional, logical, object-based, real-time, concurrent, or probabilistic. (b) The second is by level of abstraction, with requirements orientation at the very highest level, through architectural and algorithmic levels, down to platform dependence and hardware specificities at the lowest level. (c) The third axis is in the method of the presentation of semantics, such as denotational, operational, algebraic, or axiomatic. Language semantics are usually structured as complete lattices of predicates linked by Galois connections.

Example 1 (UTP theory: Boyle's Law). Building a UTP theorem is not unlike describing a physical phenomenon in physics or chemistry, and so we take as our first example modelling the behaviour of gas with varying volume and pressure. This is a physical phenomenon subject to Boyle's Law, which states

> "For a fixed amount of an ideal gas kept at a fixed temperature k, p (pressure), and V (volume) are inversely proportional (while one doubles, the other halves)."

Suppose that we want to build a computer simulation of this physical phenomenon. We need to decide what we can observe in this electronic experiment. Fortunately, the statement of Boyle's Law tells us which observations we can make in an experiment: the temperature k, the pressure p, and the volume V. These three variables form the alphabet of predicates of interest: the state of the system. In fact, they are real-world observations, and this is the model-based agenda: k, p, and V are all variables shared with real world. There is another observation hidden in the statement of Boyle's Law: the fixed amount of the gas. In a perfect world, we could count n, the number of molecules of the gas, for that is what we mean by stating that we have a fixed amount of it. But this observation is finessed by the implicit assumption that the gas is perfectly confined. If ϕ is a condition in our theory, then its alphabet is given by $\alpha(\phi) = \{p, V, k\}$; if it is a relation, then its alphabet is given by $\alpha(\phi) = \{p, V, k, p', V', k'\}$.

Having fixed on an alphabet for our theory of ideal gases, our next task is to decide on its signature: the syntax for denoting objects of the theory. Here, this will comprise three operations on the state of the system: initialisation and the manipulation of the volume and pressure of the gas. There is no call for an operation to change the temperature.

The next task is to define some healthiness conditions for predicates in our theory. These can be thought of as enforcing state and dynamic invariants, and the statement of Boyle's Law suggests one of each type. The static invariant applies to conditions on states and requires that V and p are inversely proportional: $p * V = k$. The dynamic invariant applies to relations describing state transitions and requires that k must be constant: $k' = k$.

In UTP, the technique for dealing with invariants is to create a function that enforces the invariant. Define the function $\boldsymbol{B}$ on predicates as follows:

$$\boldsymbol{B}(\phi) \;=\; (\exists\, k \bullet \phi) \wedge (k = p * V)$$

In this definition, we preserve the values of the pressure and volume and create a possibly new temperature that is in the right relationship to p and V. So, regardless of whether or not ϕ was healthy before application of $\boldsymbol{B}$, it certainly is afterwards. For example, suppose that we have

$$\phi \;=\; (p = 10) \wedge (V = 5) \wedge (k = 100)$$

then we have the following derivation

$$\begin{aligned}\boldsymbol{B}(\phi) &= (\exists\, k \bullet \phi) \wedge (k = p * V)\\ &= (\exists\, k \bullet (p = 10) \wedge (V = 5) \wedge (k = 100)) \wedge (k = p * V)\\ &= (p = 10) \wedge (V = 5) \wedge (k = p * V)\\ &= (p = 10) \wedge (V = 5) \wedge (k = 50)\end{aligned}$$

An obvious and very desirable property is that $\boldsymbol{B}$ is idempotent: $\boldsymbol{B}(\boldsymbol{B}(\phi)) = \boldsymbol{B}(\phi)$. This means that taking the medicine twice leaves you as healthy as taking it once (no overdoses). This gives us a simple test for healthiness. A predicate ϕ is already healthy if applying $\boldsymbol{B}$ leaves it unchanged: $\phi = \boldsymbol{B}(\phi)$. So, in UTP, the healthy predicates of a theory are the fixed points of idempotent functions, such as $\boldsymbol{B}$.

Now suppose that we know that the pressure of the gas is somewhere between 10 and 20 Pa; this is recorded by the predicate ψ:

$$\psi \;=\; (p \in 10\,..\,20) \wedge (V = 5)$$

The predicate ψ is rather weak in that it describes a variety of valid states (p and k are loosely constrained), as well as invalid states where the state invariant doesn't hold. In particular, ψ is satisfied by our other predicate ϕ:

$$\phi \Rimp \psi$$

Notice that this is still true if we make both predicates healthy with $\boldsymbol{B}$:

$$\boldsymbol{B}(\phi) \Rimp \boldsymbol{B}(\psi)$$

$$(p = 10) \wedge (V = 5) \wedge (k = 50) \Rimp (p \in 10\,..\,20) \wedge (V = 5) \wedge (p * V = k)$$

In this way, $\boldsymbol{B}$ is monotonic with respect to the lattice ordering. □

3 Relational calculus

As we saw in Example 1, UTP is based on an alphabetised version of the relational calculus. Relations are written pointwise, as predicates on free variables,

each of which must be in the alphabet of the relation. For example, as we'll find out below, the assignment $P = (x := x+y)$ has semantics $x' = x+y \land y' = y$. It is a relation between two states. The value of the programming variables x and x in the after-state are denoted by x' and y', respectively; the values of x and y in the before-state are denoted by x and y, respectively. These four variables must all be in the alphabet of the relation P: $\alpha P = \{x, y, x', y'\}$. It is not possible to determine the exact alphabet of a relation simply from its free variables, even though they must be included. For this reason, alphabets should be specified separately. The alphabet is partitioned between before-variables ($in\alpha P$) and after-variables ($out\alpha P$). A relation with an empty output alphabet is called a condition.

The principal operators of the relational calculus are:

Operator	**Syntax**	**Operator**	**Syntax**
conjunction	$P \land Q$	disjunction	$P \lor Q$
negation	$\neg P$	implication	$P \Rightarrow Q$
universal quantification	$\forall x \bullet P$	existential quantification	$\exists x \bullet P$
relational composition	$P \; ; \; Q$		

When two relations P and Q are used to specify programs, there is a correctness relation between, the former viewed as a specification and the latter as an implementation. Suppose that both relations are on a vector of program variable x, then they each relate the values of the variables in this vector in the states before and after their execution; we denote these values by x and x', respectively. If every pair (x, x') that satisfies Q also satisfies P, then Q is said to be a refinement of P. To formalise this, we introduce the universal closure of a predicate

$$[\, P \,] \;=\; \forall x, y, ..z \bullet P \qquad [\textit{for } \alpha P = \{x, y, ..z\}]$$

Refinement is then universal inverse implication:

$$P \sqsubseteq Q \;\; \textbf{\textit{iff}} \;\; [\, Q \Rightarrow P \,]$$

An important law for reasoning about existential quantification is the one-point rule:

$$(\exists x : T \bullet P \land (x = e)) \;=\; e \in T \land P[e/x] \qquad [\text{providing } x \text{ is not free in } e]$$

4 Nondeterministic imperative programming language

We now consider a simple nondeterministic programming language with the following syntax:

$$Prog ::= \mathit{I\!I} \mid x := e \mid P \lhd b \rhd Q \mid P \sqcap Q \mid \textbf{\textit{while}} \; b \; \textbf{\textit{do}} \; P$$

The syntax is the signature of the theory of nondeterministic imperative programming. The alphabet of predicates in this theory consists of a vector of the programming variables in scope. If P is a condition, then its alphabet is $\{v\}$ and if it is a relation, then $\{v, v'\}$. We now give the semantics for each of the program constructs.

4.1 Skip

The program II (skip) does nothing (many programming languages have such a no-op instruction). Suppose that the program state consists of a vector of variables v, then this vector is unchanged by the execution of the program:

$$\mathit{II}_{\{v\}} \;\widehat{=}\; (v' = v) \qquad\qquad \alpha\mathit{II}_{\{v\}} \;\widehat{=}\; \{v, v'\}$$

Skip plays an important role in the algebra of programs, since as shown below, it is both a left and a right unit for sequential composition.

$$P \,;\, \mathit{II}_{\alpha P} \;=\; P \;=\; \mathit{II}_{\alpha P} \,;\, P$$

4.2 Conditional

The conditional program is written in an infix notation:

$$P \lhd b \rhd Q \;\widehat{=}\; (b \wedge P) \vee (\neg\, b \wedge Q) \qquad\qquad \alpha(P \lhd b \rhd Q) \;\widehat{=}\; \alpha P$$

The condition b constrains the common before-state; the two relations P and Q must have the same alphabet:

$$\alpha b \subseteq \alpha P = \alpha Q$$

The infix notation is chosen so as to make the algebraic properties of conditional more apparent. The following laws of the conditional are familiar algebraic properties.

$$P \lhd b \rhd = P \qquad \text{idempotence}$$
$$P \lhd b \rhd Q = Q \lhd \neg\, b \rhd P \qquad \text{commutativity}$$
$$(P \lhd b \rhd Q) \lhd c \rhd R = P \lhd b \wedge c \rhd (Q \lhd c \rhd R) \qquad \text{associativity}$$
$$P \lhd b \rhd (Q \lhd c \rhd R) = (P \lhd b \rhd Q) \lhd c \rhd (P \lhd b \rhd R) \qquad \text{distributivity}$$
$$P \lhd \mathit{true} \rhd Q = P = Q \lhd \mathit{false} \rhd P \qquad \text{unit}$$

The next two examples are laws that simplify the conditional when one of its operands is either ***true*** or ***false***.

Example 2 (Conditional).

$$(P \lhd b \rhd \textbf{\textit{true}}) = (b \Rightarrow P) \qquad [\textit{conditional-right-true}]$$

Proof.

$$\begin{array}{ll} & (P \lhd b \rhd \textbf{\textit{true}}) \\ = & \{\ \text{conditional}\ \} \\ & (b \wedge P) \vee (\neg\, b \wedge \textbf{\textit{true}}) \\ = & \{\ \text{and-unit}\ \} \\ & (b \wedge P) \vee \neg\, b \\ = & \{\ \text{absorption}\ \} \\ & P \vee \neg\, b \\ = & \{\ \text{implication}\ \} \\ & b \Rightarrow P \end{array}$$

Example 3 (Conditional).

$$(P \lhd b \rhd \mathbf{false}) = (b \wedge P) \qquad \text{[conditional-right-false]}$$

Proof.

$$\begin{array}{ll} & (P \lhd b \rhd \mathbf{false}) \\ = & \{ \text{ conditional } \} \\ & (b \wedge P) \vee (\neg\, b \wedge \mathbf{false}) \\ = & \{ \text{ and-zero } \} \\ & (b \wedge P) \vee \mathbf{false} \\ = & \{ \text{or-unit } \} \\ & b \wedge P \end{array}$$

The next law imports the condition into its left-hand operand.

Example 4 (Conditional).

$$(P \lhd b \rhd Q) = ((b \wedge P) \lhd b \rhd Q) \qquad \text{[left-condition]}$$

Proof.

$$\begin{array}{ll} & (P \lhd b \rhd Q) \\ = & \{ \text{ conditional } \} \\ & (b \wedge P) \vee (\neg\, b \wedge Q) \\ = & \{ \text{ idempotence of conjunction } \} \\ & (b \wedge b \wedge P) \vee (\neg\, b \wedge Q) \\ = & \{ \text{ conditional } \} \\ & (b \wedge P) \lhd b \rhd Q \end{array}$$

Our next law is reminiscent of modus ponens: it allows us to simplify the conditional if we know the condition is true.

Example 5 (Conditional).

$$b \wedge (P \lhd b \rhd Q) = (b \wedge P) \qquad \text{[left-simplification-1]}$$

Proof.

$$\begin{array}{ll} & b \wedge (P \lhd b \rhd Q) \\ = & \{ \text{ conditional-conjunction } \} \\ & b \wedge P \lhd b \rhd b \wedge Q \\ = & \{ \text{ right-condition } \} \\ & b \wedge P \lhd b \rhd \neg\, b \wedge b \wedge Q \\ = & \{ \text{ contradiction } \} \\ & b \wedge P \lhd b \rhd \mathbf{false} \\ = & \{ \text{ conditional-right-false } \} \\ & b \wedge P \end{array}$$

The next law demonstrates that the conditional is associative, taking the encapsulated conditions into account.

Example 6 (Conditional).

$$(P \lhd b \rhd Q) \lhd c \rhd R = P \lhd b \wedge c \rhd (Q \lhd c \rhd R) \qquad \text{[associativity]}$$

Proof.

$$\begin{array}{ll} & P \lhd b \wedge c \rhd (Q \lhd c \rhd R) \\ = & \{ \text{ conditional } \} \\ & (b \wedge c \wedge P) \vee ((\neg\, b \vee \neg\, c) \wedge (Q \lhd c \rhd R)) \\ = & \{ \text{ and-or-dist. } \} \\ & (b \wedge c \wedge P) \vee (\neg\, b \wedge (Q \lhd c \rhd R)) \vee (\neg\, c \wedge (Q \lhd c \rhd R)) \\ = & \{ \text{ right-simpl. } \} \\ & (b \wedge c \wedge P) \vee (\neg\, b \wedge (Q \lhd c \rhd R)) \vee (\neg\, c \wedge R) \\ = & \{ \text{ conditional } \} \\ & (b \wedge c \wedge P) \vee (\neg\, b \wedge c \wedge Q) \vee (\neg\, b \wedge \neg\, c \wedge R) \vee (\neg\, c \wedge R) \\ = & \{ \text{ absorption } \} \\ & (b \wedge c \wedge P) \vee (\neg\, b \wedge c \wedge Q) \vee (\neg\, c \wedge R) \\ = & \{ \text{ and-or-dist } \} \\ & (c \wedge ((b \wedge P) \vee (\neg\, b \wedge Q))) \vee (\neg\, c \wedge R) \\ = & \{ \text{ conditional } \} \\ & ((b \wedge P) \vee (\neg\, b \wedge Q)) \lhd c \rhd R \\ = & \{ \text{ conditional } \} \\ & (P \lhd b \rhd Q) \lhd c \rhd R \end{array}$$

Our final example in this section is taken from [45]. It expresses in a general way the relationship between the conditional and any truth functional operator. A logical operator is truth-functional if the truth-value of a compound predicate is a function of the truth-value of its component predicates. A key fact about truth-functional operators is that substitution distributes through them.

Example 7 (Conditional).

$$(P \odot Q) \lhd b \rhd (R \odot S) = (P \lhd b \rhd R) \odot (Q \lhd b \rhd S) \qquad \text{[exchange]}$$

where $\odot$ is any truth-functional operator.

Proof.

$$
\begin{array}{ll}
 & (P \lhd b \rhd R) \odot (Q \lhd b \rhd S) \\
= & \{ \text{ propositional calculus: excluded middle } \} \\
 & (b \vee \neg\, b) \wedge ((P \lhd b \rhd R) \odot (Q \lhd b \rhd S)) \\
= & \{ \text{ and-or-distribution } \} \\
 & (b \wedge ((P \lhd b \rhd R) \odot (Q \lhd b \rhd S))) \vee (\neg\, b \wedge ((P \lhd b \rhd R) \odot (Q \lhd b \rhd S))) \\
= & \{ \text{ Leibniz } \} \\
 & (b \wedge ((P[\mathit{true}/b] \lhd \mathit{true} \rhd R[\mathit{true}/b]) \odot (Q[\mathit{true}/b] \lhd \mathit{true} \rhd S[\mathit{true}/b]))) \\
 & \vee (\neg\, b \wedge ((P[\mathit{false}/b] \lhd \mathit{false} \rhd R[\mathit{false}/b]) \odot (Q[\mathit{false}/b] \lhd \mathit{false} \rhd S[\mathit{false}/b]))) \\
= & \{ \text{ conditional-unit } \} \\
 & (b \wedge (P[\mathit{true}/b] \odot Q[\mathit{true}/b])) \vee (\neg\, b \wedge (R[\mathit{false}/b] \odot S[\mathit{false}/b])) \\
= & \{ \text{ Leibniz } \} \\
 & (b \wedge (P \odot Q)) \vee (\neg\, b \wedge (R \odot S)) \\
= & \{ \text{ conditional } \} \\
 & (P \odot Q) \lhd b \rhd (R \odot S)
\end{array}
$$

4.3 Sequential composition

The composition of two programs $(P\ ;\ Q)$ first executes P, and then executes Q on the result of P. If $out\alpha P = in\alpha Q' = \{v'\}$, then

$$P(v')\ ;\ Q(v) \ \widehat{=}\ \exists\, v_0 \bullet P(v_0) \wedge Q(v_0)$$

$$in\alpha(P(v')\ ;\ Q(v)) \ \widehat{=}\ in\alpha P \qquad out\alpha(P(v')\ ;\ Q(v)) \ \widehat{=}\ out\alpha Q$$

Sequential composition is associative and distributes leftwards into the conditional.

$$P\ ;\ (Q\ ;\ R) = (P\ ;\ Q)\ ;\ R \qquad \text{associativity}$$

$$(P \lhd b \rhd Q)\ ;\ R = (P\ ;\ R) \lhd b \rhd (Q\ ;\ R) \qquad \text{left distributivity}$$

The following trading law allows us to move a condition from the after-state of P to the before-state of Q.

Example 8 (Sequential composition).

$$(P \wedge b')\ ;\ Q \ =\ P\ ;\ (b \wedge Q) \qquad [\mathit{trading}]$$

Proof.

$$
\begin{array}{ll}
 & (P \wedge b')\ ;\ Q \\
= & \{ \text{ sequence } \} \\
 & \exists\, v_0 \bullet P[v_0/v'] \wedge b'[v_0/v'] \wedge Q[v_0/v] \\
= & \{ \text{ decoration } \} \\
 & \exists\, v_0 \bullet P[v_0/v'] \wedge b[v_0/v] \wedge Q[v_0/v] \\
= & \{ \text{ sequence } \} \\
 & P\ ;\ (b \wedge Q)
\end{array}
$$

A special case of the last example is a one-point rule for sequential composition.

Example 9 (Sequential composition). For constant k and x' not free in P:

$$(P \wedge x' = k)\ ;\ Q \ =\ P\ ;\ Q[k/x] \qquad \text{[left one-point]}$$

Proof.

$$\begin{array}{ll} & (P \wedge x' = k)\ ;\ Q \\ = & \{\ \text{sequence}\ \} \\ & \exists\, v_0, x_0 \bullet P[v_0/v'] \wedge x_0 = k \wedge Q[v_0, x_0/v, x] \\ = & \{\ \text{one-point rule}\ \} \\ & \exists\, v_0 \bullet P[v_0/v'] \wedge Q[v_0, k/v, x] \\ = & \{\ \text{sequence}\ \} \\ & P\ ;\ Q[k/x] \end{array}$$

A similar one-point rule exists for moving in the other direction:

$$P\ ;\ (x = k \wedge Q) \ =\ P[k/x']\ ;\ Q$$

4.4 Assignment

The assignment $(x :=_A e)$ relates two states with alphabet A and A', respectively, which together include x, x', and the free variables of e. It changes x to take the value e, keeping all other variables constant. For $A = \{x, y, \ldots, z\}$ and $\alpha e \subseteq A$, we have

$$x :=_A e \ \widehat{=}\ (x' = e \wedge y' = y \wedge \cdots \wedge z' = z) \qquad \alpha(x :=_A e) \ \widehat{=}\ A \cup A'$$

The subscript to the assignment operator is omitted when it can be inferred from context.

$$(x := e) = (x, y := e, y) \qquad \text{contract frame}$$
$$(x, y, z := e, f, g) = (y, x, z := f, e, g) \qquad \text{commutativity}$$
$$(x := e\ ;\ x := f(x)) = (x := f(e)) \qquad \text{assignment-conditional distributivity}$$

A leading assignment can be pushed into a following conditional.

Example 10 (Sequential composition).

$$\begin{array}{l} (x := e\ ;\ (P \lhd b(x) \rhd Q)) \qquad \text{[left-assignment-conditional]} \\ \qquad =\ ((x := e\ ;\ P) \lhd b(e) \rhd (x := e\ ;\ Q)) \end{array}$$

Proof.

$$\begin{array}{ll} & x := e\ ;\ (P \lhd b(x) \rhd Q) \\ = & \{\ \text{assignment}\ \} \\ & (x' = e \wedge v' = v)\ ;\ (P \lhd b(x) \rhd Q) \\ = & \{\ \text{left-one-point, twice}\ \} \\ & (P[e/x] \lhd b(e) \rhd Q[e/x]) \\ = & \{\ \text{left-one-point, twice}\ \} \\ & ((x' = e \wedge v' = v)\ ;\ P) \lhd b(e) \rhd ((x' = e \wedge v' = v)\ ;\ Q) \\ = & \{\ \text{assignment}\ \} \\ & (x := e\ ;\ P) \lhd b(e) \rhd (x := e\ ;\ Q) \end{array}$$

Notice how this proof is entirely algebraic.

4.5 Nondeterministic choice

The nondeterministic choice $P \sqcap Q$ behaves either like P or like Q:

$$P \sqcap Q \;\widehat{=}\; P \vee Q$$

$P \sqcap P = P$	idempotence
$P \sqcap Q = Q \sqcap P$	commutativity
$P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R$	associativity
$P \lhd b \rhd (Q \sqcap R) = (P \lhd b \rhd Q) \sqcap (P \lhd b \rhd R)$	$\lhd\,\rhd$-$\sqcap$ distributivity
$P \sqcap (Q \lhd b \rhd R) = (P \sqcap Q) \lhd b \rhd (P \sqcap R)$	$\sqcap$-$\lhd\,\rhd$ distributivity
$(P \sqcap Q)\,;\,R = (P\,;\,R) \sqcap (Q\,;\,R)$	sequence disjunctivity
$P\,;\,(Q \sqcap R) = (P\,;\,Q) \sqcap (P\,;\,R)$‘	sequence disjunctivity

5 Lattices

Let $(L, \sqsubseteq)$ be a partially ordered set and let a and b be any pair of elements in L. The meet of a and b, the lattice operator denoted by $a \sqcap b$, is the greatest lower-bound of a and b:

$$a \sqcap b \;\widehat{=}\; \max\{\, c : L \mid c \sqsubseteq a \wedge c \sqsubseteq b \,\}$$

The join of a and b, denoted by $a \sqcup b$, is the least upper-bound of a and b:

$$a \sqcup b \;\widehat{=}\; \min\{\, c : L \mid a \sqsubseteq c \wedge b \sqsubseteq c \,\}$$

Both operators are idempotent, commutative, and associative, and satisfy a pair of absorption laws:

$a \sqcap a = a$	$\sqcap$-idempotent
$a \sqcap b = b \sqcap a$	$\sqcap$-commutative
$a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c$	$\sqcap$-associative
$a \sqcup a = a$	$\sqcup$-idempotent
$a \sqcup b = b \sqcup a$	$\sqcup$-commutative
$a \sqcup (b \sqcup c) = (a \sqcup b) \sqcup c$	$\sqcup$-associative
$a \sqcup (a \sqcap b) = a$	$\sqcup$-$\sqcap$-absorption
$a \sqcap (a \sqcup b) = a$	$\sqcap$-$\sqcup$-absorption

A lattice consists of a partially set $(L, \sqsubseteq)$, such that any two elements have both a meet and a join. L is a complete lattice if every subset A of L has both a meet and a join. The greatest lower-bound of the whole of L is the bottom element $\bot$; the least upper-bound of the whole of L is the top element $\top$.

Example 11 (Powerset lattice). The powerset of S ordered by inclusion is a lattice. The empty set is the least element and S is the greatest element. Intersection is the meet operation and union is the join. Fig. 1 depicts the lattice $(\{0, 1, 2\}, \subseteq)$.

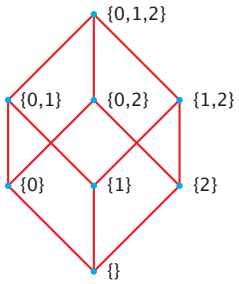

Fig. 1. The lattice $(\{0, 1, 2\}, \subseteq)$.

Example 12 (Divisibility lattice). The natural numbers ordered by divisibility form a partial order. Divisibility is defined as follows:

$$m \text{ divides } n \;\widehat{=}\; \exists\, k \bullet k * m = n$$

The natural number 1 is the bottom element: it exactly divides every other number. The natural number 0 is the top element: it can be divided exactly by every other number. Fig. 2 depicts the lattice $(0 \mathinner{..} 8, \text{divides})$.

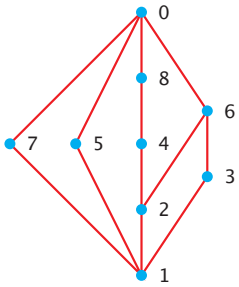

Fig. 2. The lattice $(0 \mathinner{..} 8, \text{divides})$.

A function f is monotonic with respect to an ordering $\sqsubseteq$, providing that

$$\forall\, x, y : \text{dom}\, f \bullet x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

Now we come to the theorem that justifies our interest in complete lattices. Tarski's fixed-point theorem states the following:

Let L be a complete lattice and let $f : L \to L$ be a monotonic function; then the set of fixed points of f in L is also a complete lattice.

Example 13 (Fixed points in Powerset lattice). Let $f : \mathbb{P}\{0,1,2\} \to \mathbb{P}\{0,1,2\}$ be defined as $f(s) = s \cup \{0\}$. Clearly, f is monotonic with respect to the subset ordering. Fig. 4 depicts the lattice of the fixed points of f.

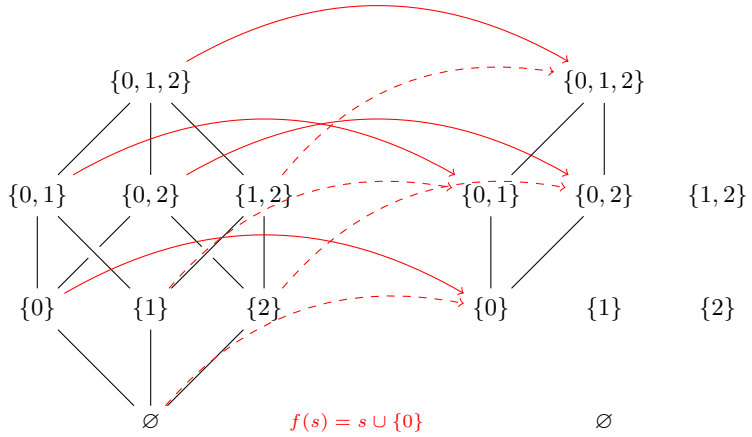

Fig. 3. Fixed points of $f(s) = s \cup \{0\}$.

Tarski's theorem is interesting for us, since we want to give semantics to iteration and recursion in terms of fixed points. The theorem guarantees the existence of a fixed point, so long as the body of the iteration or recursion is monotonic. Furthermore, it helps us to choose which fixed point to use, by guaranteeing the arrangement of all fixed points in a lattice. The bottom element of the fixed-point lattice is conventionally denoted by μF and the top element by νF. The former is the weakest fixed-point of F and the latter the strongest fixed-point of F. Fig. 4 shows the complete lattice of fixed points of a function F. The diagram also shows how the lattice of fixed points can be defined using the order relation on the lattice, since

$$(X = F(X)) \;=\; (X \sqsubseteq F(X)) \wedge (F(X) \sqsubseteq X)$$

A pre-fixed point of F is any X such that $F(X) \sqsubseteq X$; a post-fixed point of F is any X such that $X \sqsubseteq F(X)$. Now, another way to express Tarski's fixed-point theorem is

A monotonic function on a complete lattice has a weakest fixed-point that coincides with its weakest pre-fixed-point; its strongest fixed-point coincides with its strongest post-fixed-point.

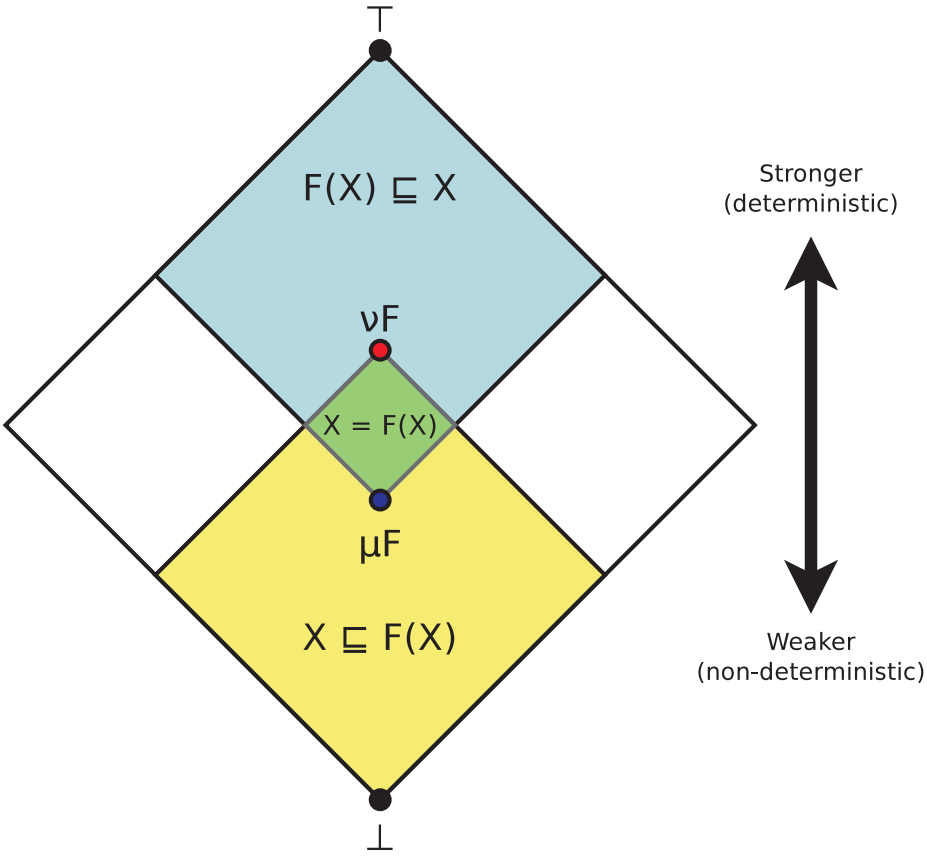

Fig. 4. Complete Lattice of Fixed Points.

6 Recursion

After our discussion of complete lattices in the last section, we return now to the alphabetised relational calculus. Predicates with a particular alphabet form a complete lattice under a refinement ordering that is universal inverse implication

$$(P \sqsubseteq Q) \;=\; [\, Q \Rightarrow P \,]$$

The bottom of the lattice is *abort*, the worst program because it can behave without constraint: ***true***. The top of the lattice is *miracle*, the best program because it can achieve the impossible: ***false***.

$$\begin{array}{ll} \bot_A \;\widehat{=}\; \textbf{\textit{true}} & \alpha\bot_A \;\widehat{=}\; A \\ \top_A \;\widehat{=}\; \textbf{\textit{false}} & \alpha\top_A \;\widehat{=}\; A \end{array}$$

The lattice greatest lower-bound ($\bigsqcap$) is simply disjunction and the least upper-bound ($\bigsqcup$) is simply conjunction. Two axioms give the essential properties of these two operators.

$$P \sqsubseteq \bigsqcap S \;\textbf{iff}\; \forall X : S \bullet (P \sqsubseteq X) \qquad \text{[greatest lower-bound axiom]}$$
$$\bigsqcap S \sqsubseteq P \;\textbf{iff}\; \forall X : S \bullet (X \sqsubseteq P) \qquad \text{[least upper-bound axiom]}$$

The next four laws specify useful properties of the two operators:

$$\begin{array}{ll} \forall X : S \bullet (\bigsqcap S \sqsubseteq X) & \text{lower bound} \\ (\forall X : S \bullet P \sqsubseteq X) \Rightarrow (P \sqsubseteq \bigsqcap S) & \text{greatest lower-bound} \\ \forall X : S \bullet (X \sqsubseteq \bigsqcap S) & \text{upper bound} \\ (\forall X : S \bullet X \sqsubseteq P) \Rightarrow (\bigsqcap S \sqsubseteq P) & \text{least upper-bound} \end{array}$$

Finally the least and greatest elements have the obvious properties:

$$\begin{array}{ll} \bot \sqsubseteq P & \text{bottom element} \\ P \sqsubseteq \top & \text{top element} \end{array}$$

In this setting, recursion is given a semantics as the strongest fixed-point, the least upper bound of all the post-fixed points of the recursive function.

$$\nu F \;\widehat{=}\; \bigsqcup\{\, X \mid X \sqsubseteq F(X) \,\}$$

The weakest fixed-point has the dual definition:

$$\mu F \;\widehat{=}\; \bigsqcap\{\, X \mid F(X) \sqsubseteq X \,\}$$

These two operators have the following characteristic properties:

$$\begin{array}{ll} (F(Y) \sqsubseteq Y) \Rightarrow (\mu F \sqsubseteq Y) & \text{weakest fixed-point} \\ \mu F = F(\mu F) & \text{fixed point} \\ (S \sqsubseteq F(S)) \Rightarrow (S \sqsubseteq \nu F) & \text{strongest fixed-point} \\ \nu F = F(\nu F) & \text{fixed point} \end{array}$$

Example 14 (Hoare logic for while loop). Strongest fixed-point semantics leads to a simple rule for reasoning about iteration, which is defined in terms of recursion.

$$\frac{\{\, b \wedge c \,\}\ P\ \{\, c \,\}}{\{\, b \wedge c \,\}\ \textbf{\textit{while}}\ b\ \textbf{\textit{do}}\ P\ \{\, \neg\, b \wedge c \,\}}$$

The validity of this rule depends on the strongest fixed-point law:

$$(S \sqsubseteq F(S)) \Rightarrow (S \sqsubseteq \nu F)$$

This allows us to reason about a recursive implementation, at the risk of producing an infeasible program: the miracle is always a correct implementation. Of course, since it is the predicate ***false***, it has no behaviour, and in particular, cannot be guaranteed to terminate. So the simplicity of the rule must be balanced by proving termination separately.

In contrast, the weakest fixed-point law doesn't allow us to reason about a recursive implementation, but instead about a recursive specification, since the fixed-point operator is on the left of the refinement, which is not useful here:

$$(F(Y) \sqsubseteq Y) \Rightarrow (\mu\, F \sqsubseteq Y)$$

If we can show that the recursive program terminates, then the weakest and strongest fixed-points actually coincide.

Our next law shows how to unfold a weakest fixed-point involving the composition of two functions. This is known as the rolling rule.

Example 15 (Fixed points).

$$\mu\, X \bullet F(G(X)) \;=\; F(\mu\, X \bullet G(F(X)))$$

Proof. We prove this by mutual refinement.

1. ($\sqsubseteq$)

$$\begin{array}{ll} & \mu\, X \bullet F(G(X)) \sqsubseteq F(\mu\, X \bullet G(F(X))) \\ = & \{\ \text{weakest fixed-point}\ \} \\ & \bigsqcap \{\, X \mid F(G(X)) \sqsubseteq X \,\} \sqsubseteq F(\mu\, X \bullet G(F(X))) \\ \Leftarrow & \{\ \text{lower bound}\ \} \\ & F(\mu\, X \bullet G(F(X))) \in \{\, X \mid F(G(X)) \sqsubseteq X \,\} \\ \Leftarrow & \{\ \text{comprehension}\ \} \\ & F(G(F(\mu\, X \bullet G(F(X))))) \sqsubseteq F(\mu\, X \bullet G(F(X))) \\ = & \{\ \text{fixed point}\ \} \\ & F(\mu\, X \bullet G(F(X))) \sqsubseteq F(\mu\, X \bullet G(F(X))) \\ = & \{\ \text{refinement reflexive}\ \} \\ & \mathit{true} \end{array}$$

2. ($\sqsupseteq$) Suppose by hypothesis that $F(G(X)) \sqsubseteq X$.

$$
\begin{array}{ll}
& F(G(X)) \sqsubseteq X \\
\Rightarrow & \{\ G \text{ monotonic } \} \\
& G(F(G(X))) \sqsubseteq G(X) \\
= & \{\ \text{comprehension } \} \\
& G(X) \in \{\, X \mid G(F(X)) \sqsubseteq X \,\} \\
\Rightarrow & \{\ \text{lower bound } \} \\
& \bigsqcap \{\, X \mid G(F(X)) \sqsubseteq X \,\} \sqsubseteq G(X) \\
= & \{\ \text{weakest fixed-point } \} \\
& \mu\, X \bullet G(F(X)) \sqsubseteq G(X) \\
\Rightarrow & \{\ F \text{ monotonic } \} \\
& F(\mu\, X \bullet G(F(X))) \sqsubseteq F(G(X)) \\
\Rightarrow & \{\ \text{monotonicity of refinement, hypothesis } \} \\
& F(\mu\, X \bullet G(F(X))) \sqsubseteq X
\end{array}
$$

Therefore,

$$\forall\, X \in \{\, X \mid F(G(X)) \sqsubseteq X \,\} \bullet F(\mu\, X \bullet G(F(X))) \sqsubseteq X$$

and so by the definition of least upper-bound, we have

$$F(\mu\, X \bullet G(F(X))) \sqsubseteq \bigsqcap \{\, X \mid F(G(X)) \sqsubseteq X \,\}$$

and so by the definition of weakest fixed-point we have

$$F(\mu\, X \bullet G(F(X))) \sqsubseteq \mu\, X \bullet F(G(X))$$

Example 16. Haskell B. Curry's $\boldsymbol{Y}$ combinator is a higher-order function that computes a fixed point of other functions.

$$\boldsymbol{Y} \ \widehat{=}\ \lambda\, G \bullet (\lambda\, g \bullet G(g\ g))(\lambda\, g \bullet G(g\ g))$$

We prove that $\boldsymbol{Y}\, F$ really is a fixed point of F.

Proof.

$$
\begin{array}{ll}
& \boldsymbol{Y}\, F \\
= & \{\ \boldsymbol{Y} \text{ definition } \} \\
& (\lambda\, G \bullet (\lambda\, g \bullet G(g\ g))(\lambda\, g \bullet G(g\ g)))\ F \\
= & \{\ \text{reduction } \} \\
& (\lambda\, g \bullet F(g\ g))(\lambda\, g \bullet F(g\ g)) \\
= & \{\ \text{above } \} \\
& (\lambda\, g \bullet F(g\ g))(\lambda\, g \bullet F(g\ g)) \\
= & \{\ \text{reduction } \} \\
& F((\lambda\, g \bullet F(g\ g))(\lambda\, g \bullet F(g\ g))) \\
= & \{\ \text{above } \} \\
& F(\boldsymbol{Y}\, F)
\end{array}
$$

Example 17. Define the body of a function that calculates factorials as follows:

$$F \;\widehat{=}\; \lambda f \bullet \lambda x \bullet (1 \lhd x = 0 \rhd x * f(x-1))$$

Calculate the value of $(\boldsymbol{Y}F)(n)$ in terms of $(\boldsymbol{Y}F)(n-1)$.

$$\begin{array}{ll} & (\boldsymbol{Y}F)(n) \\ = & \{\ \boldsymbol{Y} \text{ is a fixed point of } F\ \} \\ & (F(\boldsymbol{Y}F))(n) \\ = & \{\ F \text{ definition}\ \} \\ & (\lambda x \bullet (1 \lhd x = 0 \rhd x * (\boldsymbol{Y}F)(x-1)))(n) \\ = & \{\ \beta \text{ reduction}\ \} \\ & 1 \lhd n = 0 \rhd n * (\boldsymbol{Y}F)(n-1) \end{array}$$

Example 18 (Lattices). Suppose that we know that a function F has a unique fixed-point, modulo C.

$$(C \wedge \mu F) = (C \wedge \nu F)$$

Suppose in addition that C is itself a fixed-point of F. Prove that F has an unconditional unique fixed-point. That is, the weakest and strongest fixed-points are equal, modulo C. But C is also a fixed point. The last two facts mean that the strongest fixed-point is actually C.

$$\begin{array}{ll} & C \wedge \mu F = C \wedge \nu F \\ = & \{\ \text{predicate calculus}\ \} \\ & [\, C \Rightarrow (\mu F = \nu F)\,] \\ \Rightarrow & \{\ C \text{ is a fixed point of } F\ \} \\ & [\, C \Rightarrow (\mu F = \nu F) \wedge \mu F \sqsubseteq C \sqsubseteq \nu F\,] \\ = & \{\ \text{Leibniz}\ \} \\ & [\, C \Rightarrow (\mu F = \nu F) \wedge \nu F \sqsubseteq C \sqsubseteq \nu F\,] \\ \Rightarrow & \{\ \text{propositional calculus}\ \} \\ & [\, C \Rightarrow (\nu F \sqsubseteq C)\,] \\ \Rightarrow & \{\ \text{refinement}\ \} \\ & [\, C \Rightarrow [\, C \Rightarrow \nu F\,]\,] \\ \Rightarrow & \{\ \text{propositional calculus}\ \} \\ & [\, C \Rightarrow \nu F\,] \\ = & \{\ \text{refinement}\ \} \\ & \nu F \sqsubseteq C \\ = & \{\ \nu F \text{ is strongest fixed-point, so } C \sqsubseteq \nu F, \text{ equality}\ \} \\ & \nu F = C \end{array}$$

7 Assertional reasoning

Hoare logic is a system for reasoning about computer programs, in this case, about programs written in the nondeterministic programming language we have

introduced. In this kind of program logic, each syntactic construct in the language's signature is provided with an introduction rule that can be used to reason about this construct.

The key notion in Hoare logic is the Hoare triple $\{p\}\ Q\ \{r\}$:

> If precondition p holds of the state before the execution of program Q, then, if Q terminates, postcondition r will hold afterwards.

Notice that this is a statement of partial correctness. The Hoare triple is defined in UTP as follows:

$$\{p\}\ Q\ \{r\}\ \mathrel{\widehat{=}}\ (p \Rightarrow r') \sqsubseteq Q$$

The definition constructs a relational specification from the precondition p and postcondition r as an implication: $p \Rightarrow r'$. (Note how the postcondition must be decorated as a predicate on the after-state to distinguish it from the precondition, which is a predicate on the before-state.) If the precondition doesn't hold, then this is simply ***true***, which is the semantics of the *abort* program, which is the bottom of the refinement lattice and Q automatically refines it.

The rules of Hoare logic can now all be proved valid as theorems from the definition of the Hoare triple.

L1 **if** $\{p\}\ Q\ \{r\}$ **and** $\{p\}\ Q\ \{s\}$ **then** $\{p\}\ Q\ \{r \wedge s\}$
L2 **if** $\{p\}\ Q\ \{r\}$ **and** $\{q\}\ Q\ \{r\}$ **then** $\{p \vee q\}\ Q\ \{r\}$
L3 **if** $\{p\}\ Q\ \{r\}$ **then** $\{p \wedge q\}\ Q\ \{r \vee s\}$

L4 $\{\, r[e/x] \,\}\ x := e\ \{\, r \,\}$
L5 **if** $\{p \wedge b\}\ Q_1\ \{r\}$ **and** $\{p \wedge \neg\, b\}\ Q_2\ \{r\}$
then $\{\, p \,\}\ Q_1 \lhd b \rhd Q_2\ \{\, r \,\}$
L6 **if** $\{p\}\ Q_1\ \{s\}$ **and** $\{s\}\ Q_2\ \{r\}$ **then** $\{\, p \,\}\ Q_1 \,;\, Q_2\ \{\, r \,\}$

L7 **if** $\{p\}\ Q_1\ \{r\}$ **and** $\{p\}\ Q_2\ \{r\}$ **then** $\{\, p \,\}\ Q_1 \sqcap Q_2\ \{\, r \,\}$
L8 **if** $\{b \wedge c\}\ Q\ \{c\}$
then $\{\, c \,\}\ \nu X \bullet (Q \,;\, X) \lhd b \rhd \mathit{II}\ \{\, \neg\, b \wedge c \,\}$
L9 $\{\mathit{false}\}\ Q\ \{r\}$ **and** $\{p\}\ Q\ \{\mathit{true}\}$
and $\{p\}$ **false** $\{\mathit{false}\}$ **and** $\{p\}\ \mathit{II}\ \{p\}$

We prove the axiom for reasoning about the conditional as a theorem in the underlying semantics of Hoare logic.

Example 19 (Hoare logic).

if $\{p\}\ Q\ \{r\}$ ***and*** $\{q\}\ Q\ \{r\}$ ***then*** $\{(p \vee q)\}\ Q\ \{r\}$

Proof.

$$
\begin{array}{ll}
 & \{(p \vee q)\}\ Q\ \{r\} \\
= & \{ \text{ Hoare triple } \} \\
 & [\, Q \Rightarrow ((p \vee q) \Rightarrow r')\,] \\
= & \{ \text{ collecting antecedents } \} \\
 & [\, Q \wedge (p \vee q) \Rightarrow r'\,] \\
= & \{ \text{ and-or-distribution } \} \\
 & [\, (Q \wedge p) \vee (Q \wedge q) \Rightarrow r'\,] \\
= & \{ \text{ or-implies } \} \\
 & [\, (Q \wedge p \Rightarrow r') \wedge (Q \wedge q \Rightarrow r')\,] \\
= & \{ \text{ for-all-associativity } \} \\
 & [\, Q \wedge p \Rightarrow r'\,] \wedge [\, Q \wedge q \Rightarrow r'\,] \\
= & \{ \text{ collecting antecedents } \} \\
 & [\, Q \Rightarrow (p \Rightarrow r')\,] \wedge [\, Q \Rightarrow (q \Rightarrow r')\,] \\
= & \{ \text{ Hoare triple } \} \\
 & (\{p\}\ Q\ \{r\}) \wedge (\{q\}\ Q\ \{r\})
\end{array}
$$

Next, we prove the rule for reasoning about assignment.

Example 20 (Assignment rule).

$$\{r[e/x]\}\ x := e\ \{r\}$$

Proof.

$$
\begin{array}{ll}
 & \{r[e/x]\}\ x := e\ \{r(x)\} \\
= & \{ \text{ Hoare triple } \} \\
 & [\, x := e \Rightarrow (r[e/x] \Rightarrow r[x'/x])\,] \\
= & \{ \text{ assignment } \} \\
 & [\, x' = e \wedge v' = v \Rightarrow (r[e/x] \Rightarrow r[x'/x])\,] \\
= & \{ \text{ universal one-point rule } \} \\
 & [\, (r[e/x] \Rightarrow r[x'/x][e/x'])\,] \\
= & \{ \text{ substitution, implication } \} \\
 & [\, \mathbf{true}\,] \\
= & \{ \text{ universal quantification } \} \\
 & \mathit{true}
\end{array}
$$

The Hoare triple $\{p\}\ Q\ \{r\}$ is a tertiary relation between a precondition p, postcondition r and program Q. If we fix any two of these, then we can find solutions for the third. The weakest precondition calculus is based on this idea: it fixes the program Q and a postcondition r and provides the weakest solution for p.

Example 21 (Weakest precondition derivation).

$$
\begin{array}{ll}
& \{\, p \,\}\, Q \,\{\, r \,\} \\
= & \{ \text{ Hoare triple } \} \\
& [\, Q \Rightarrow (\, p \Rightarrow r' \,)\,] \\
= & \{ \text{ implication } \} \\
& [\, p \Rightarrow (\, Q \Rightarrow r' \,)\,] \\
= & \{ \text{ universal closure } (v' \text{ in the alphabet}) \ \} \\
& [\, p \Rightarrow (\, \forall\, v' \bullet Q \Rightarrow r' \,)\,] \\
= & \{ \text{ De Morgan's law } \} \\
& [\, p \Rightarrow \neg\, (\, \exists\, v' \bullet Q \wedge \neg\, r' \,)\,] \\
= & \{ \text{change of bound variable (fresh } v_0) \ \} \\
& [\, p \Rightarrow \neg\, (\, \exists\, v_0 \bullet Q[v_0/v'] \wedge \neg\, r_0 \,)\,] \\
= & \{ \text{sequential composition } \} \\
& [\, p \Rightarrow \neg\, (\, Q \,;\, \neg\, r \,)\,]
\end{array}
$$

The final line of this derivation suggests the weakest solution for Q to guarantee r: p can be equal to any predicate that satisfies this expression, but it cannot be weaker than $\neg\, (\, Q \,;\, \neg\, r \,)$. That is, the behaviours other than those where Q violates the postcondition r. This leads us to the definition:

$$Q \ \mathbf{wp}\ r \ \mathrel{\widehat{=}}\ \neg\, (Q \,;\, \neg\, r)$$

We now use this definition to prove some of the laws of the weakest precondition calculus as theorems of the relational theory.

Example 22 (Weakest precondition for sequential composition).

$$((P \,;\, Q)\ \mathbf{wp}\ r) = (P\ \mathbf{wp}\ (Q\ \mathbf{wp}\ r))$$

Proof.

$$
\begin{array}{ll}
 & ((P \,;\, Q)\ \mathbf{wp}\ r) \\
= & \{\ \mathbf{wp}\ \} \\
 & \neg\,((P \,;\, Q) \,;\, \neg\, r) \\
= & \{\ \text{sequence}\ \} \\
 & \neg\,(\exists\, v_0 \bullet (P \,;\, Q[v_0/v']) \wedge \neg\, r_0) \\
= & \{\ \text{sequence}\ \} \\
 & \neg\,(\exists\, v_0 \bullet (\exists\, v_1 \bullet P[v_1/v'] \wedge Q[v_1, v_0/v, v']) \wedge \neg\, r_0) \\
= & \{\ \text{expand scope}\ \} \\
 & \neg\,(\exists\, v_1, v_0 \bullet P[v_1/v'] \wedge Q[v_1, v_0/v, v'] \wedge \neg\, r_0) \\
= & \{\ \text{restrict scope}\ \} \\
 & \neg\,(\exists\, v_1 \bullet P[v_1/v'] \wedge (\exists\, v_0 \bullet Q[v_1, v_0/v, v'] \wedge \neg\, r_0)) \\
= & \{\ \text{sequence}\ \} \\
 & \neg\,(\exists\, v_1 \bullet P[v_1/v'] \wedge (Q[v_1/v] \,;\, \neg\, r)) \\
= & \{\ \text{double negation}\ \} \\
 & \neg\,(\exists\, v_1 \bullet P[v_1/v'] \wedge \neg\,\neg\,(Q[v_1/v] \,;\, \neg\, r)) \\
= & \{\ \mathbf{wp}\ \} \\
 & \neg\,(\exists\, v_1 \bullet P[v_1/v'] \wedge \neg\,(Q[v_1/v]\ \mathbf{wp}\ r)) \\
= & \{\ \text{sequence}\ \} \\
 & \neg\,(P \,;\, \neg\,(Q\ \mathbf{wp}\ r)) \\
= & \{\ \mathbf{wp}\ \} \\
 & (P\ \mathbf{wp}\ (Q\ \mathbf{wp}\ r))
\end{array}
$$

Example 23 (Weakest precondition conjunctive).

$$(Q\ \mathbf{wp}\ (\bigwedge R)) = \bigwedge \{\, (Q\ \mathbf{wp}\ r) \mid r \in R \,\}$$

Proof.

$$
\begin{array}{ll}
 & Q\ \mathbf{wp}\ (\bigwedge R) \\
= & \{\ \mathbf{wp}\ \} \\
 & \neg\,(Q \,;\, \neg\,(\bigwedge R)) \\
= & \{\ \text{duality}\ \} \\
 & \neg\,(Q \,;\, \bigvee \{\, \neg\, r \mid r \in R \,\}) \\
= & \{\ \text{sequence disjunction}\ \} \\
 & \neg\,(\bigvee \{\, Q \,;\, \neg\, r \mid r \in R \,\}) \\
= & \{\ \text{duality}\ \} \\
 & \bigwedge \{\, \neg\,(Q \,;\, \neg\, r) \mid r \in R \,\} \\
= & \{\ \mathbf{wp}\ \} \\
 & \bigwedge \{\, Q\ \mathbf{wp}\ r) \mid r \in R \,\}
\end{array}
$$

8 Designs

We now turn to an important theory in UTP that describes the semantics of our nondeterministic imperative programming once more, but this time in a

theory of total correctness. Termination is captured in the semantics by using assumption-commitment pairs. This gives a way of specifying behaviour that is similar to VDM [46], B [1], and the refinement calculus [3, 48, 49].

The theory of designs involves two boolean observations: ok, which signals that the program has started; and ok', which signals that the program has terminated. The use of these two observations allows us to encode the precondition and postcondition as a single relation:

$$(P \vdash Q) \mathrel{\widehat{=}} (ok \wedge P \Rightarrow ok' \wedge Q)$$

for P and Q not containing ok or ok'. This definition can be read as

> "If the program has started (ok) and the precondition P holds, then it must terminate (ok') in a state where the postcondition Q holds."

Example 24 (Search with sentinel). Suppose that we want to specify a program that searches an *array* for an element x, and that we assume that x is somewhere in the array (maybe in multiple occurrences). We can arrange for this assumption to hold by extending the array by one element and inserting x at the end (Dijkstra's "sentinel"). We model the array as a function from indexes to elements. Here is our specification:

$$x \in \operatorname{ran} array \vdash array' = array \wedge i' \in \operatorname{dom} array \wedge array(i') = x$$

The precondition states that we can assume $x \in \operatorname{ran} array$. The postcondition states that the array isn't changed by this operation $array' = array$, that the index ends up pointing to an element of the array $i' \in \operatorname{dom} array$, and that it ends up pointing to an occurrence of x in the array $array(i') = x$.

We now re-express the semantics of the nondeterministic programming language in terms of designs.

8.1 Skip

Skip still does nothing, as before, but we must add a precondition to insist that it always terminates:

$$\mathit{II}_D \mathrel{\widehat{=}} (\mathbf{true} \vdash \mathit{II})$$

8.2 Conditional

In design semantics, the conditional is a choice between two designs. The result is, of course, a design:

$$(P_1 \vdash P_2) \lhd b \rhd (Q_1 \vdash Q_2) \;=\; (P_1 \lhd b \rhd Q_1) \vdash (P_2 \lhd b \rhd Q_2)$$

Actually, this is not a definition, but a theorem that relies on the previous definition of the conditional and on the definition of a design.

8.3 Sequential composition

For the sequential composition operator, we have another theorem:

$$(p_1 \vdash P_2)\ ;\ (Q_1 \vdash Q_2)\ =\ (p_1 \wedge (P_2\ \textbf{\textit{wp}}\ Q_1) \vdash P_2\ ;\ Q_2)$$

The meaning of the sequential composition augments this precondition by the weakest precondition for the first postcondition to establish the second precondition. This guarantees that control can be passed successfully from the first design to the second. Finally, the overall postcondition is simply the relational composition of the individual postconditions.

8.4 Assignment

For the design assignment, we need to consider a precondition that guarantees that the assignment will not abort. In the case of $(x := 1/y)$, the precondition establishes the definedness of the expression $1/y$, which includes $y \neq 0$, as well as considerations of overflow and underflow. In this paper, we assume that the expression is well-defined, without these problems. As a result, we simply lift the semantics of the relational assignment:

$$x := e\ \mathrel{\widehat{=}}\ (\textbf{\textit{true}} \vdash x := e)$$

8.5 Nondeterministic choice

For nondeterministic choice, we have another theorem:

$$(P_1 \vdash P_2) \sqcap (Q_1 \vdash Q_2)\ =\ (P_1 \wedge Q_1 \vdash P_2 \vee Q_2)$$

The resulting design must satisfy the assumptions of both designs, but need establish the postcondition of only one of them.

9 The complete lattice of designs

The greatest lower-bound of a set of designs has a similar form to the binary case for nondeterministic choice. Since we don't know which design will be selected, all the preconditions must hold in advance of the selection. The postcondition is nondeterministically selected.

$$\textstyle\prod_i (P_i \vdash Q_i)\ \mathrel{\widehat{=}}\ (\bigwedge_i P_i) \vdash (\bigvee_i Q_i)$$

The least upper-bound of a set of designs has a weaker precondition than each individual design (see the discussion on refinement, below). But at the same time, since it is the least upper-bound, this precondition needs to be as strong as possible. Thus, the actual precondition is $(\bigvee_i P_i)$. The postcondition is the

conjunction of all the individual postconditions, each modified to assume its individual precondition.

$$\bigsqcup_i (P_i \vdash Q_i) \mathrel{\widehat{=}} (\bigvee_i P_i) \vdash (\bigwedge_i P_i \Rightarrow Q_i)$$

To exemplify this, we show how modulus operation can be constructed from the least upper bound of the positive and negative cases.

Example 25 (Least upper-bound of designs).

$$\begin{aligned} & (x \geq 0 \vdash x' = x) \sqcap (x \leq 0 \vdash x' = -x) \\ = & (x \geq 0 \vee x \leq 0 \vdash (x \geq 0 \Rightarrow x' = x) \wedge (x \leq 0 \Rightarrow x' = -x)) \\ = & (\mathit{true} \vdash x' = |x|) \end{aligned}$$

With these definitions, designs form a complete lattice. The bottom of the lattice is abort

$$\bot_D \mathrel{\widehat{=}} \mathit{false} \vdash \mathit{true}$$

The definition of a design allows us to simplify this to ***true***. The top of the lattice is miracle:

$$\top_D \mathrel{\widehat{=}} \mathit{true} \vdash \mathit{false}$$

Again, we can simplify this, and we obtain $\neg\ ok$. So, the program that can achieve the impossible is the program that cannot be started.

10 Galois connections

In UTP, the links between different theories are expressed as Galois connections. Backhouse [4] introduces a useful example, which we adopt here.

Example 26 (The floor function). The floor function is defined informally as follows:

> For all real numbers x, the floor of x is the greatest integer that is at most x.

More formally, the floor function is an extreme solution for n in the following equivalence:

$$\mathit{real}(n) \leq x \textbf{ iff } n \leq \mathit{floor}(x)$$

Where $\mathit{real} : \int \to$ is a function that casts an integer to its real number representation. It should be noted that we're overloading the inequality relation. On one side of the equivalence, it is inequality between two real numbers, whilst on the other side, it is inequality between integers.

Example 27 (Floor rounds downwards). Instantiating n to $\mathit{floor}(x)$, our equivalence gives us

$$\mathit{real}(\mathit{floor}(x)) \leq x \textbf{ iff } \mathit{floor}(x) \leq \mathit{floor}(x)$$

which simplifies to $\mathit{real}(\mathit{floor}(x)) \leq x$. So, we now know that the floor function rounds downwards.

Example 28 (Floor is inverse for real). Instantiating x to $\mathit{real}(n)$, we get

$$\mathit{real}(n) \leq \mathit{real}(n) \textbf{ iff } n \leq \mathit{floor}(\mathit{real}(n))$$

which simplifies to $n \leq \mathit{floor}(\mathit{real}(n))$. Now, using our previous result, with x instantiated to $\mathit{real}(n)$, we have the conjunction

$$n \leq \mathit{floor}(\mathit{real}(n)) \wedge \mathit{real}(\mathit{floor}(\mathit{real}(n))) \leq \mathit{real}(n)$$

Next, the function that maps an integer to its real representation is injective, so we have

$$n \leq \mathit{floor}(\mathit{real}(n)) \wedge \mathit{floor}(\mathit{real}(n)) \leq n$$

which is equivalent to

$$n = \mathit{floor}(\mathit{real}(n))$$

So, *floor* is an exact inverse for *real*.

Example 29 (Floor brackets real). Let's take the contrapositive of the equivalence defining the floor function:

$$\begin{array}{ll} & \mathit{real}(n) \leq x \textbf{ iff } n \leq \mathit{floor}(x) \\ = & \{ \text{ contraposition } \} \\ & \neg\,(\mathit{real}(n) \leq x) \textbf{ iff } \neg\,(n \leq \mathit{floor}(x)) \\ = & \{ \text{ arithmetic } \} \\ & x < \mathit{real}(n) \textbf{ iff } \mathit{floor}(x) < n \\ = & \{ \text{ arithmetic } \} \\ & x < \mathit{real}(n) \textbf{ iff } \mathit{floor}(x) + 1 \leq n \end{array}$$

Now, instantiate n with $\mathit{floor}(x) + 1$:

$$x < \mathit{real}(\mathit{floor}(x) + 1) \textbf{ iff } \mathit{floor}(x) + 1 \leq \mathit{floor}(x) + 1$$

But we already know that $\mathit{floor}(x) \leq x$, so we have

$$\mathit{floor}(x) \leq x \leq \mathit{floor}(x) + 1$$

Example 30 (Floor monotonic). We want to prove that

$$x \leq y \;\Rightarrow\; \mathit{floor}(x) \leq \mathit{floor}(y)$$

First, we specialise the definition of the Galois connection between *real* and *floor*:

$$
\begin{array}{ll}
 & \mathit{real}(n) \leq x \ \textbf{iff}\ n \leq \mathit{floor}(x) \\
\Rightarrow & \{\ \text{specialisation with x,n} := \text{y,floor(x)}\ \} \\
 & \mathit{real}(\mathit{floor}(x)) \leq y = \mathit{floor}(x) \leq \mathit{floor}(y)
\end{array}
$$

Now we can use this result to prove the monotonicity of *floor*:

$$
\begin{array}{ll}
 & \mathit{floor}(x) \leq \mathit{floor}(y) \\
= & \{\ \text{above}\ \} \\
 & \mathit{real}(\mathit{floor}(x)) \leq y \\
\Leftarrow & \{\ \text{transitivity of} \leq\ \} \\
 & \mathit{real}(\mathit{floor}(x)) \leq x \leq y \\
= & \{\ \text{since } \mathit{floor}(x) \leq x\ \} \\
 & x \leq y
\end{array}
$$

What we have achieved in the last example is to prove that *real* and *floor* form a Galois connection between the real numbers and the integers and to explore some of the consequences of this result. Specifically, the *floor* function provides the best approximation of a real number as an integer. We now describe the notion of Galois connections more generally.

Let $\mathbf{S}$ and $\mathbf{T}$ both be complete lattices. Let L be a function from $\mathbf{S}$ to $\mathbf{T}$. Let R be a function from $\mathbf{T}$ to $\mathbf{S}$. The pair (L, R) is a Galois connection if

$$
\begin{array}{l}
\text{for all } X \in \mathbf{S} \text{ and } Y \in \mathbf{T} : \\
L(X) \sqsupseteq Y \quad \textbf{iff} \quad X \sqsupseteq R(Y)
\end{array}
$$

R is a weak inverse of L (right adjoint); L is a strong inverse of R (left adjoint).

Example 31 (Galois connection: relational theory and designs). There is a Galois connection between the two semantics that we have provided for the nondeterministic imperative programming language.

The left adjoint, which we'll call $\mathit{Des}(R)$, maps a plain relation R to a design. The relation comes from the theory of partial correctness, where we assume that a relational program R terminates. We record this assumption by adding the precondition $\textbf{\textit{true}}$ when we map to the design $\textbf{\textit{true}} \vdash R$.

The right adjoint, which we'll call *Rel*, maps a design back to a plain relation. In the theory of designs, we can observe the start and termination of execution, but these observations cannot be made in the theory of relations. So we must assume initiation and termination by setting ok and ok' both the $\textbf{\textit{true}}$. Thus we have $\mathit{Rel}(D) = D[\textbf{\textit{true}}, \textbf{\textit{true}}/ok, ok']$.

We introduce the abbreviations: $D^b = D[b/ok']$, $D^t = D^{\textbf{\textit{true}}}$, $D^f = D^{\textbf{\textit{false}}}$.

Example 32 (Des is the inverse of Rel).

Proof.

$$
\begin{array}{ll}
& Des \circ Rel(P \vdash Q) \\
= & \{ \text{ definition of } Rel \ \} \\
& Des((P \vdash Q)^t[\mathbf{true}/ok]) \\
= & \{ \text{ substitution } \} \\
& Des(P \Rightarrow Q) \\
= & \{ \text{ definition of } Des \ \} \\
= & \mathbf{true} \vdash P \Rightarrow Q \\
= & \{ \text{ definition of design, propositional calculus } \} \\
= & P \vdash Q
\end{array}
$$

Example 33 (Extraction of precondition and postcondition). Every design D can be expressed as $(\neg\, D^f \vdash D^t)$. Without loss of generality, we exploit the fact that we have characterised designs syntactically. So it is sufficient to prove that

$$P \vdash Q \ = \ \neg\, (P \vdash Q)^f \vdash (P \vdash Q)^t$$

Proof.

$$
\begin{array}{ll}
& \neg\, (P \vdash Q)^f \vdash (P \vdash Q)^t \\
= & \{ \text{ definition of design, substitution } \} \\
& \neg\, (ok \wedge P \Rightarrow false \wedge Q) \vdash ok \wedge P \Rightarrow true \wedge Q \\
= & \{ \text{ propositional calculus } \} \\
& ok \wedge P \vdash ok \wedge P \Rightarrow Q \\
= & \{ \text{ definition of design } \} \\
& ok \wedge P \Rightarrow ok' \wedge (ok \wedge P \Rightarrow Q) \\
= & \{ \text{ propositional calculus } \} \\
& ok \wedge P \Rightarrow ok' \wedge Q \\
= & \{ \text{ definition of design } \} \\
& P \vdash Q
\end{array}
$$

This example allows us to write the following equation for *Rel*:

$$Rel(D) = (\neg\, D^f \Rightarrow D^t)$$

Example 34 (Refinement for designs). Recall the definition of refinement for relations:

$$P \sqsubseteq Q \ = \ [\, Q \Rightarrow P \,]$$

We keep the same order relation on designs; after all, a design is a rather special kind of relation. In VDM and B, refinement is usually expressed through the two slogans:

> Weaken the precondition, strengthen the postcondition.

More formally,

$$(P_1 \vdash P_2) \sqsubseteq (Q_1 \vdash Q_2) \ = \ [\, P_1 \Rightarrow Q_1 \,] \wedge [\, P_1 \wedge Q_2 \Rightarrow Q_1 \,]$$

We show that the VDM/B slogan is a consequence of the relational view of refinement. That is,

$$((P_1 \vdash P_2) \sqsubseteq (Q_1 \vdash Q_2)) \;=\; [\,P_1 \wedge Q_2 \Rightarrow P_2\,] \wedge [\,P_1 \Rightarrow Q_1\,]$$

Proof.

$$\begin{array}{ll}
& (P_1 \vdash P_2) \sqsubseteq (Q_1 \vdash Q_2) \\
= & \{\ \text{definition of refinement}\ \} \\
& [\,(Q_1 \vdash Q_2) \Rightarrow (P_1 \vdash P_2)\,] \\
= & \{\ \text{universal closure}\ \} \\
& [\,(Q_1 \vdash Q_2)[\mathit{true}/\mathit{ok}] \Rightarrow (P_1 \vdash P_2)[\mathit{true}/\mathit{ok}]\,] \\
& \wedge\, [\,(Q_1 \vdash Q_2)[\mathit{false}/\mathit{ok}] \Rightarrow (P_1 \vdash P_2)[\mathit{false}/\mathit{ok}]\,] \\
= & \{\ \text{definition of design}\ \} \\
& [\,(Q_1 \Rightarrow \mathit{ok}' \wedge Q_2) \Rightarrow (P_1 \Rightarrow \mathit{ok}' \wedge P_2)\,] \\
= & \{\ \text{universal closure}\ \} \\
& [\,(Q_1 \Rightarrow \mathit{ok}' \wedge Q_2)[\mathit{true}/\mathit{ok}'] \Rightarrow (P_1 \Rightarrow \mathit{ok}' \wedge P_2)[\mathit{true}/\mathit{ok}']\,] \\
& \wedge\, [\,(Q_1 \Rightarrow \mathit{ok}' \wedge Q_2)[\mathit{false}/\mathit{ok}'] \Rightarrow (P_1 \Rightarrow \mathit{ok}' \wedge P_2)[\mathit{false}/\mathit{ok}']\,] \\
= & \{\ \text{propositional calculus}\ \} \\
& [\,(Q_1 \Rightarrow Q_2) \Rightarrow (P_1 \Rightarrow P_2)\,] \wedge [\,\neg\, Q_1 \Rightarrow \neg\, P_1\,] \\
= & \{\ \text{propositional calculus}\ \} \\
& [\,P_1 \wedge (Q_1 \Rightarrow Q_2) \Rightarrow P_2\,] \wedge [\,P_1 \Rightarrow Q_1\,] \\
= & \{\ \text{predicate calculus}\ \} \\
& [\,P_1 \wedge Q_2 \Rightarrow P_2\,] \wedge [\,P_1 \Rightarrow Q_1\,]
\end{array}$$

Finally, we use this result to show that *Des* and *Rel* form a Galois connection.

Example 35 ((Des, Rel) is a Galois connection).

Proof.

$$\begin{array}{ll}
& \mathit{Des}(R) \sqsupseteq D \\
= & \{\ \text{definition of } \mathit{Des}\ \} \\
& (\mathbf{\mathit{true}} \vdash R) \sqsupseteq D \\
= & \{\ \text{refinement of designs}\ \} \\
& [\,\neg\, D^f \Rightarrow \mathbf{\mathit{true}}\,] \wedge [\,\neg\, D^f \wedge R \Rightarrow D^t\,] \\
= & \{\ \text{propositional calculus}\ \} \\
& [\,\neg\, D^f \wedge R \Rightarrow D^t\,] \\
= & \{\ \text{propositional calculus}\ \} \\
& [\,R \Rightarrow (\neg\, D^f \Rightarrow D^t)\,] \\
= & \{\ \text{refinement of relations}\ \} \\
& R \sqsupseteq (\neg\, D^f \Rightarrow D^t) \\
= & \{\ \text{definition of } \mathit{Rel}\ \} \\
& R \sqsupseteq \mathit{Rel}(D)
\end{array}$$

11 Design healthiness conditions

There are two principal healthiness conditions for design-hood: one for ok and one for ok'.

The first concerns starting programs: no observation can be made before the program starts.

$$\mathbf{H1}(P) \;=\; ok \Rightarrow P$$

The second concerns terminating programs: anything is better than nontermination

$$\mathbf{H2} : [\, P[\mathbf{false}/ok'] \Rightarrow P[\mathbf{true}/ok'] \,]$$

This healthiness condition states that you mustn't require nontermination as a property of a program.

*Example 36 (**H2** as a monotonic idempotent).* We've expressed **H2** as a property, but it can also be expressed as a monotonic idempotent function. The **H2** property that we've specified requires a predicate to be monotonic in ok'. We can introduce a pseudo-identity to capture this:

$$J \;=\; (ok \Rightarrow ok') \wedge \mathit{II}(v)$$

and then redefine **H2** as a function:

$$\mathbf{H2}(P) \;=\; P \,;\, J$$

This leads to a useful lemma, for a **H2**-healthy predicate P:

$$P \;=\; P^f \vee (ok' \wedge P^t)$$

Proof.

$$\begin{array}{ll}
 & P \\
= & \{\ P \text{ is } \mathbf{H2}\ \} \\
 & P \,;\, J \\
= & \{\ \text{propositional calculus}\ \} \\
 & P \,;\, (\neg\, ok \vee ok') \wedge \mathit{II}(v) \\
= & \{\ \text{relational calculus}\ \} \\
 & (P \,;\, \neg\, ok \wedge \mathit{II}(v)) \vee (P \,;\, ok' \wedge \mathit{II}(v)) \\
= & \{\ \text{relational calculus}\ \} \\
 & (P^f \,;\, \mathit{II}(v)) \vee ((P \,;\, \mathit{II}(v)) \wedge ok') \\
= & \{\ \text{relational unit (alphabets match)}\ \} \\
 & P^f \vee ((P \,;\, \mathit{II}(v)) \wedge ok') \\
= & \{\ \text{relational calculus}\ \} \\
 & P^f \vee ((\exists\, ok' \bullet P) \wedge ok') \\
= & \{\ \text{case enumeration } (ok' \text{ is boolean})\ \} \\
 & P^f \vee ((P^t \vee P^f) \wedge ok') \\
= & \{\ \text{propositional calculus}\ \} \\
 & P^f \vee (P^t \wedge ok') \vee (P^f \wedge ok') \\
= & \{\ \text{absorption}\ \} \\
 & P^f \vee (P^t \wedge ok')
\end{array}$$

This is known as J-splitting, and it emphasises the asymmetry in the use of ok': you can observe when a program terminates, but not when it doesn't.

*Example 37 (**H1** relations).* We give four examples of **H1** relations.

1. The bottom of the design lattice is $\textbf{\textit{false}} \vdash \textbf{\textit{true}}$, which is equivalent to **true**, which, by the propositional calculus, is a fixed point of the **H1** healthiness condition: $(ok \Rightarrow \textbf{\textit{true}}) = \textbf{\textit{true}}$.
2. The top of the design lattice is $\textbf{\textit{true}} \vdash \textbf{\textit{false}}$, which is equivalent to $\neg\ ok$, which, by the propositional calculus, is also a fixed point of the **H1** healthiness condition: $(ok \Rightarrow \neg\ ok) = \neg\ ok$.
3. A property of implication means that any predicate with ok as an implicative antecedent must be **H1**-healthy. For example: $(ok \wedge x \neq 0 \Rightarrow x' < x)$.
4. Finally, every design must be **H1**-healthy, since ok is an implicit assumption. For example: $(x \neq 0 \vdash x' < x)$.

*Example 38 (**H2** predicates).* We give four examples of **H2** relations.

1. The bottom of the design lattice is **H2**-healthy:

$$\begin{array}{ll} & \bot_D^f \\ = & \textbf{\textit{true}}^f \\ = & \textbf{\textit{true}} \\ = & \textbf{\textit{true}}^t \\ = & \bot_D^t \end{array}$$

2. The top of the design lattice is also **H2**-healthy:

$$\begin{array}{ll} & \top_{\textbf{\textit{D}}}^f \\ = & (\neg\ ok)^f \\ = & \neg\ ok \\ = & (\neg\ ok)^t \\ = & \top_{\textbf{\textit{D}}}^t \end{array}$$

3. Any predicate that insists on termination is **H2**-healthy. For example:

$$\begin{array}{ll} & (ok' \wedge (x' = 0))^f \\ = & \textbf{\textit{false}} \\ \Rightarrow & (x' = 0) \\ = & (ok' \wedge x' = 0)^t \end{array}$$

4. Finally, any design is **H2**-healthy. For example:

$$\begin{array}{ll} & (x \neq 0 \vdash x' < x)^f \\ = & (ok \wedge x \neq 0 \Rightarrow ok' \wedge x' < x)^f \\ = & (ok \wedge x \neq 0 \Rightarrow \textbf{\textit{false}}) \\ \Rightarrow & (ok \wedge x \neq 0 \Rightarrow x' < x) \\ = & (ok \wedge x \neq 0 \Rightarrow ok' \wedge x' < x)^t \\ = & (x \neq 0 \vdash x' < x)^t \end{array}$$

12 In conclusion

This concludes our tutorial introduction to the theories of relations and designs in UTP. Other tutorial introductions may be found in [72, 14]. Of course, the interested reader is encouraged to go back to the source of the ideas and read the book.

References

1. J.-R. Abrial. *The B Book — Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. Hugh Anderson, Gabriel Ciobanu, and Leo Freitas. UTP and temporal logic model checking. In *[11]*, pages 22–41, 2008.
3. Ralph-Johan Back and Joakim Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.
4. Roland Carl Backhouse. Galois connections and fixed point calculus. In Roland Carl Backhouse, Roy L. Crole, and Jeremy Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction, International Summer School and Workshop, Oxford, UK, April 10–14, 2000, Revised Lectures*, volume 2297 of *Lecture Notes in Computer Science*, pages 89–148. Springer, 2000.
5. Victor Bandur and Jim Woodcock. Unifying theories of logic and specification. In Juliano Iyoda and Leonardo Mendonca de Moura, editors, *Formal Methods: Foundations and Applications — 16th Brazilian Symposium, SBMF 2013, Brasilia, 29 September–4 October 2013*, volume 8195 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2013.
6. Michael J. Banks and Jeremy L. Jacob. On modelling user observations in the UTP. In *[57]*, pages 101–119, 2010.
7. Michael J. Banks and Jeremy L. Jacob. Unifying theories of confidentiality. In *[57]*, pages 120–136, 2010.
8. A. Beg and A. Butterfield. Linking a state-rich process algebra to a state-free algebra to verify software/hardware implementation. In *FIT 2010, Proceedings of the 8th International Conference on Frontiers of Information Technology*, 2010.
9. Riccardo Bresciani and Andrew Butterfield. A probabilistic theory of designs based on distributions. In *[68]*, pages 105–123, 2012.
10. Andrew Butterfield. Saoithín: A theorem prover for UTP. In *[57]*, pages 137–156, 2010.
11. Andrew Butterfield, editor. *Unifying Theories of Programming, Second International Symposium, UTP 2008, Dublin, 8–10 September 2008, Revised Selected Papers*, volume 5713 of *Lecture Notes in Computer Science*. Springer, 2010.
12. Andrew Butterfield. The logic of $u \cdot (tp)^2$. In *[68]*, pages 124–143, 2012.
13. Andrew Butterfield, Adnan Sherif, and Jim Woodcock. Slotted Circus. In Jim Davies and Jeremy Gibbons, editors, *Integrated Formal Methods, 6th International Conference, IFM 2007, Oxford, 2–5 July 2007*, volume 4591 of *Lecture Notes in Computer Science*, pages 75–97. Springer, 2007.
14. A. Cavalcanti and J. Woodcock. A tutorial introduction to CSP in Unifying Theories of Programming. In Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock, editors, *Refinement Techniques in Software Engineering, First Pernambuco Summer School on Software Engineering, PSSE 2004*, volume 3167 of *LNCS*, pages 220–268. Springer, 2006.

15. Ana Cavalcanti and Marie-Claude Gaudel. A note on traces refinement and the *conf* relation in the Unifying Theories of Programming. In *[11]*, pages 42–61, 2008.
16. Ana Cavalcanti and Marie-Claude Gaudel. Specification coverage for testing in Circus. In *[57]*, pages 1–45, 2010.
17. Ana Cavalcanti, Alexandre Mota, and Jim Woodcock. Simulink timed models for program verification. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theories of Programming and Formal Methods — Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, volume 8051 of *Lecture Notes in Computer Science*, pages 82–99. Springer, 2013.
18. Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. Unifying classes and processes. *Software and System Modeling*, 4(3):277–296, 2005.
19. Ana Cavalcanti, Andy J. Wellings, and Jim Woodcock. The Safety-critical Java memory model: A formal account. In Michael Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods — 17th International Symposium on Formal Methods, Limerick, 20–24 June 2011*, volume 6664 of *Lecture Notes in Computer Science*, pages 246–261. Springer, 2011.
20. Ana Cavalcanti, Andy J. Wellings, and Jim Woodcock. The Safety-critical Java memory model formalised. *Formal Asp. Comput.*, 25(1):37–57, 2013.
21. Ana Cavalcanti, Andy J. Wellings, Jim Woodcock, Kun Wei, and Frank Zeyda. Safety-critical Java in Circus. In Andy J. Wellings and Anders P. Ravn, editors, *The 9th International Workshop on Java Technologies for Real-time and Embedded Systems, JTRES'11, York, 26–28 September 2011*, pages 20–29. ACM, 2011.
22. Ana Cavalcanti and Jim Woodcock. Angelic nondeterminism and Unifying Theories of Programming. *Electr. Notes Theor. Comput. Sci.*, 137(2):45–66, 2005.
23. Ana Cavalcanti, Jim Woodcock, and Steve Dunne. Angelic nondeterminism in the Unifying Theories of Programming. *Formal Asp. Comput.*, 18(3):288–307, 2006.
24. Ana Cavalcanti, Frank Zeyda, Andy J. Wellings, Jim Woodcock, and Kun Wei. Safety-critical Java programs from Circus models. *Real-Time Systems*, 49(5):614–667, 2013.
25. Xin Chen, Nan Ye, and Wenxu Ding. A formal approach to analyzing interference problems in aspect-oriented designs. In *[57]*, pages 157–171, 2010.
26. Yifeng Chen. Programmable verifiers in imperative programming. In *[57]*, pages 172–187, 2010.
27. Moshe Deutsch and Martin C. Henson. A relational investigation of UTP designs and prescriptions. In *[30]*, pages 101–122, 2006.
28. Steve Dunne. Conscriptions: A new relational model for sequential computations. In *[68]*, pages 144–163, 2012.
29. Steve Dunne, Ian J. Hayes, and Andy Galloway. Reasoning about loops in total and general correctness. In *[11]*, pages 62–81, 2008.
30. Steve Dunne and Bill Stoddart, editors. *Unifying Theories of Programming, First International Symposium, UTP 2006, Walworth Castle, County Durham, 5–7 February 2006, Revised Selected Papers*, volume 4010 of *Lecture Notes in Computer Science*. Springer, 2006.
31. Abderrahmane Feliachi, Marie-Claude Gaudel, and Burkhart Wolff. Unifying Theories in Isabelle/HOL. In *[57]*, pages 188–206, 2010.
32. S. Foster, F. Zeyda, and J. Woodcock. Isabelle/UTP: A mechanised theory engineering framework. In *5th International Symposium on Unifying Theories of Programming (To appear)*, 2014.
33. Simon Foster and Jim Woodcock. Unifying Theories of Programming in Isabelle. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Unifying Theories of*

Programming and Formal Engineering Methods — International Training School on Software Engineering, Held at ICTAC 2013, Shanghai, 26–30 August 2013, Advanced Lectures, volume 8050 of *Lecture Notes in Computer Science*, pages 109–155. Springer, 2013.
34. Simon Foster, Frank Zeyda, and Jim Woodcock. Unifying heterogeneous state-spaces with lenses. In Augusto Sampaio and Farn Wang, editors, *Theoretical Aspects of Computing — ICTAC 2016 - 13th International Colloquium, Taipei, Taiwan, ROC, October 24–31, 2016, Proceedings*, volume 9965 of *Lecture Notes in Computer Science*, pages 295–314, 2016.
35. Walter Guttmann. Lazy UTP. In *[11]*, pages 82–101, 2008.
36. Walter Guttmann. Unifying recursion in partial, total and general correctness. In *[57]*, pages 207–225, 2010.
37. Will Harwood, Ana Cavalcanti, and Jim Woodcock. A theory of pointers for the UTP. In John S. Fitzgerald, Anne Elisabeth Haxthausen, and Hüsnü Yenigün, editors, *Theoretical Aspects of Computing — ICTAC 2008, 5th International Colloquium, Istanbul, 1–3 September 2008*, volume 5160 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2008.
38. Ian J. Hayes. Termination of real-time programs: Definitely, definitely not, or maybe. In *[30]*, pages 141–154, 2006.
39. Jifeng He. Transaction calculus. In *[11]*, pages 2–21, 2008.
40. Jifeng He. A probabilistic BPEL-like language. In *[57]*, pages 74–100, 2010.
41. Jifeng He and C. A. R. Hoare. Csp is a retract of CCS. In *[30]*, pages 38–62, 2006.
42. Jifeng He, Shengchao Qin, and Adnan Sherif. Constructing property-oriented models for verification. In *[30]*, pages 85–100, 2006.
43. Jifeng He and Jeff W. Sanders. Unifying probability. In *[30]*, pages 173–199, 2006.
44. Eric C. R. Hehner. Retrospective and prospective for unifying theories of programming. In *[30]*, pages 1–17, 2006.
45. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1998.
46. C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 1986.
47. Alistair A. McEwan and Jim Woodcock. Unifying theories of interrupts. In *[11]*, pages 122–141, 2008.
48. Carroll Morgan. *Programming from Specifications*. Prentice-Hall International, 2nd edition, 1994.
49. J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.
50. Gift Nuka and Jim Woodcock. Mechanising a unifying theory. In *[30]*, pages 217–235, 2006.
51. Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. Unifying Theories in ProofPower-Z. In *[30]*, pages 123–140, 2006.
52. Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. A denotational semantics for Circus. *Electr. Notes Theor. Comput. Sci.*, 187:107–123, 2007.
53. Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. A UTP semantics for Circus. *Formal Asp. Comput.*, 21(1–2):3–32, 2009.
54. Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. Unifying theories in ProofPower-Z. *Formal Asp. Comput.*, 25(1):133–158, 2013.
55. Juan Ignacio Perna and Jim Woodcock. A denotational semantics for Handel-C hardware compilation. In Michael Butler, Michael G. Hinchey, and María M. Larrondo-Petrie, editors, *Formal Methods and Software Engineering, 9th International Conference on Formal Engineering Methods, ICFEM 2007, Boca Raton,*

14–15 September 2007, volume 4789 of *Lecture Notes in Computer Science*, pages 266–285. Springer, 2007.
56. Juan Ignacio Perna and Jim Woodcock. UTP semantics for Handel-C. In *[11]*, pages 142–160, 2008.
57. Shengchao Qin, editor. *Unifying Theories of Programming - Third International Symposium, UTP 2010, Shanghai, 15–16 November 2010*, volume 6445 of *Lecture Notes in Computer Science*. Springer, 2010.
58. Pedro Ribeiro and Ana Cavalcanti. Designs with angelic nondeterminism. In *Seventh International Symposium on Theoretical Aspects of Software Engineering, TASE 2013, 1–3 July 2013, Birmingham*, pages 71–78. IEEE, 2013.
59. Thiago L. V. L. Santos, Ana Cavalcanti, and Augusto Sampaio. Object-orientation in the UTP. In *[30]*, pages 18–37, 2006.
60. Adnan Sherif, Ana Cavalcanti, Jifeng He, and Augusto Sampaio. A process algebraic framework for specification and validation of real-time systems. *Formal Asp. Comput.*, 22(2):153–191, 2010.
61. Adnan Sherif and Jifeng He. Towards a time model for Circus. In Chris George and Huaikou Miao, editors, *Formal Methods and Software Engineering, 4th International Conference on Formal Engineering Methods, ICFEM 2002 Shanghai, 21–25 October 2002*, volume 2495 of *Lecture Notes in Computer Science*, pages 613–624. Springer, 2002.
62. Adnan Sherif, Jifeng He, Ana Cavalcanti, and Augusto Sampaio. A framework for specification and validation of real-time systems using Circus actions. In Zhiming Liu and Keijiro Araki, editors, *Theoretical Aspects of Computing — ICTAC 2004, First International Colloquium, Guiyang, 20–24 September 2004, Revised Selected Papers*, volume 3407 of *Lecture Notes in Computer Science*, pages 478–493. Springer, 2005.
63. Michael Anthony Smith and Jeremy Gibbons. Unifying theories of locations. In *[11]*, pages 161–180, 2008.
64. Bill Stoddart and Pete Bell. Probabilistic choice, reversibility, loops, and miracles. In *[57]*, pages 253–270, 2010.
65. Bill Stoddart, Frank Zeyda, and Robert Lynas. A design-based model of reversible computation. In *[30]*, pages 63–83, 2006.
66. Kun Wei, Jim Woodcock, and Ana Cavalcanti. Circus Time with reactive designs. In *[68]*, pages 68–87, 2012.
67. Martin Weiglhofer and Bernhard K. Aichernig. Unifying input output conformance. In *[11]*, pages 181–201, 2008.
68. Burkhart Wolff, Marie-Claude Gaudel, and Abderrahmane Feliachi, editors. *Unifying Theories of Programming, 4th International Symposium, UTP 2012, Paris, 27–28 August 2012, Revised Selected Papers*, volume 7681 of *Lecture Notes in Computer Science*. Springer, 2013.
69. Jim Woodcock. The miracle of reactive programming. In *[11]*, pages 202–217, 2008.
70. Jim Woodcock. Engineering UToPiA — formal semantics for CML. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods — 19th International Symposium, Singapore, 12–16 May 2014*, volume 8442 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2014.
71. Jim Woodcock and Victor Bandur. Unifying theories of undefinedness in UTP. In *[68]*, pages 1–22, 2012.
72. Jim Woodcock and Ana Cavalcanti. A tutorial introduction to designs in Unifying Theories of Programming. In Eerke A. Boiten, John Derrick, and Graeme Smith,

editors, *Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, 4-7 April 2004*, volume 2999 of *Lecture Notes in Computer Science*, pages 40–66. Springer, 2004.
73. Jim Woodcock, Ana Cavalcanti, John S. Fitzgerald, Peter Gorm Larsen, Alvaro Miyazawa, and S. Perry. Features of CML: A formal modelling language for Systems of Systems. In *7th International Conference on System of Systems Engineering, SoSE 2012, Genova, 16–19 July 2012*, pages 445–450. IEEE, 2012.
74. Frank Zeyda and Ana Cavalcanti. Encoding Circus programs in ProofPowerZ. In *[11]*, pages 218–237, 2008.
75. Frank Zeyda and Ana Cavalcanti. Higher-order UTP for a theory of methods. In *[68]*, pages 204–223, 2012.
76. Naijun Zhan, Eun-Young Kang, and Zhiming Liu. Component publications and compositions. In *[11]*, pages 238–257, 2008.
77. Huibiao Zhu, Jifeng He, Xiaoqing Peng, and Naiyong Jin. Denotational approach to an event-driven system-level language. In *[11]*, pages 258–278, 2008.
78. Huibiao Zhu, Peng Liu, Jifeng He, and Shengchao Qin. Mechanical approach to linking operational semantics and algebraic semantics for Verilog using Maude. In *[68]*, pages 164–185, 2012.
79. Huibiao Zhu, Jeff W. Sanders, Jifeng He, and Shengchao Qin. Denotational semantics for a probabilistic timed shared-variable language. In *[68]*, pages 224–247, 2012.
80. Huibiao Zhu, Fan Yang, and Jifeng He. Generating denotational semantics from algebraic semantics for event-driven system-level language. In *[57]*, pages 286–308, 2010.