A Tutorial Introduction to Designs in Unifying Theories of Programming

Jim Woodcock and Ana Cavalcanti

University of Kent
Computing Laboratory
Canterbury UK
{J.C.P.Woodcock,A.L.C.Cavalcanti}@kent.ac.uk

Abstract. In their *Unifying Theories of Programming (UTP)*, Hoare & He use the alphabetised relational calculus to give denotational semantics to a wide variety of constructs taken from different programming paradigms. A key concept in their programme is the *design*: the familiar precondition-postcondition pair that describes the contract between a programmer and a client. We give a tutorial introduction to the theory of alphabetised relations, and its sub-theory of designs. We illustrate the ideas by applying them to theories of imperative programming, including Hoare logic, weakest preconditions, and the refinement calculus.

1 Introduction

The book by Hoare & He [6] sets out a research programme to find a common basis in which to explain a wide variety of programming paradigms: unifying theories of programming (UTP). Their technique is to isolate important language features, and give them a denotational semantics. This allows different languages and paradigms to be compared.

The semantic model is an alphabetised version of Tarski's relational calculus, presented in a predicative style that is reminiscent of the schema calculus in the Z [14] notation. Each programming construct is formalised as a relation between an initial and an intermediate or final observation. The collection of these relations forms a *theory* of the paradigm being studied, and it contains three essential parts: an alphabet, a signature, and healthiness conditions.

The *alphabet* is a set of variable names that gives the vocabulary for the theory being studied. Names are chosen for any relevant external observations of behaviour. For instance, programming variables x, y, and z would be part of the alphabet. Also, theories for particular programming paradigms require the observation of extra information; some examples are a flag that says whether the program has started (*okay*); the current time (*clock*); the number of available resources (*res*); a trace of the events in the life of the program (*tr*); or a flag that says whether the program is waiting for interaction with its environment (*wait*). The *signature* gives the rules for the syntax for denoting objects of the theory. *Healthiness conditions* identify properties that characterise the theory.
Each healthiness condition embodies an important fact about the computational model for the programs being studied.

E. Boiten, J. Derrick, G. Smith (Eds.): IFM 2004, LNCS 2999, pp. 40–66, 2004.

Example 1 (Healthiness conditions).

1. The variable *clock* gives us an observation of the current time, which moves ever onwards. The predicate B specifies this.

   $$B \mathrel{\widehat{=}} clock \leq clock'$$

   If we add B to the description of some activity, then the variable *clock* describes the time observed immediately before the activity starts, whereas *clock'* describes the time observed immediately after the activity ends. If we suppose that P is a healthy program, then we must have that $P \Rightarrow B$.
2. The variable *okay* is used to record whether or not a program has started. A sensible healthiness condition is that we should not observe a program's behaviour until it has started; such programs satisfy the following equation.

   $$P = (okay \Rightarrow P)$$

   If the program has not started, its behaviour is not described. □

Healthiness conditions can often be expressed in terms of a function ϕ that makes a program healthy. There is no point in applying ϕ twice, since we cannot make a healthy program even healthier. Therefore, ϕ must be idempotent: $P = \phi(P)$; this equation characterises the healthiness condition. For example, we can turn the first healthiness condition above into an equivalent equation, $P = P \wedge B$, and then the following function on predicates $and_B \mathrel{\widehat{=}} \lambda X \bullet P \wedge B$ is the required idempotent.

The relations are used as a semantic model for unified languages of specification and programming. Specifications are distinguished from programs only by the fact that the latter use a restricted signature. As a consequence of this restriction, programs satisfy a richer set of healthiness conditions.

Unconstrained relations are too general to handle the issue of program termination; they need to be restricted by healthiness conditions. The result is the theory of designs, which is the basis for the study of the other programming paradigms in [6]. Here, we present the general relational setting, and the transition to the theory of designs.

In the next section, we present the most general theory of UTP: the alphabetised predicates. In the following section, we establish that this theory is a complete lattice. Section 4 discusses Hoare logic and weakest preconditions. Section 5 restricts the general theory to designs. Next, in Section 6, we present an alternative characterisation of the theory of designs using healthiness conditions. After that, we rework the Hoare logic and weakest preconditions definitions; we also outline a novel formalisation of Morgan's calculus based on designs. Finally, we conclude with a summary and a brief account of related work.

2 The Alphabetised Relational Calculus

The alphabetised relational calculus is similar to Z's schema calculus, except that it is untyped and rather simpler. An *alphabetised predicate* (P, Q, $\ldots$, true) is an alphabet-predicate pair, where the predicate's free variables are all members of the alphabet. Relations are predicates in which the alphabet is composed of undecorated variables (x, y, z, $\dots$) and dashed variables (x', a', $\dots$); the former represent initial observations, and the latter, observations made at a later intermediate or final point. The alphabet of an alphabetised predicate P is denoted αP, and may be divided into its before-variables ($in\alpha P$) and its after-variables ($out\alpha P$). A *homogeneous relation* has $out\alpha P = in\alpha P'$, where $in\alpha P'$ is the set of variables obtained by dashing all variable in the alphabet $in\alpha P$. A *condition* ($b, c, d, \dots, true$) has an empty output alphabet.

Standard predicate calculus operators can be used to combine alphabetised predicates. Their definitions, however, have to specify the alphabet of the combined predicate. For instance, the alphabet of a conjunction is the union of the alphabets of its components: $\alpha(P \land Q) = \alpha P \cup \alpha Q$. Of course, if a variable is mentioned in the alphabet of both P and Q, then they are both constraining the same variable.

A distinguishing feature of UTP is its concern with program development, and consequently program correctness. A significant achievement is that the notion of program correctness is the same in every paradigm in [6]: in every state, the behaviour of an implementation implies its specification.

If we suppose that $\alpha P = \{a, b, a', b'\}$, then the *universal closure* of P is simply $\forall\, a, b, a', b' \bullet P$, which is more concisely denoted as $[\,P\,]$. The correctness of a program P with respect to a specification S is denoted by $S \sqsubseteq P$ (S is refined by P), and is defined as follows.

$$S \sqsubseteq P \quad \textit{iff} \quad [\,P \Rightarrow S\,]$$

Example 2 (Refinement). Suppose we have the specification $x' > x \land y' = y$, and the implementation $x' = x + 1 \land y' = y$. The implementation's correctness is argued as follows.

$$\begin{aligned}
& x' > x \land y' = y \;\sqsubseteq\; x' = x + 1 \land y' = y && \textit{[definition of } \sqsubseteq\textit{]}\\
= {} & [\, x' = x + 1 \land y' = y \Rightarrow x' > x \land y' = y \,] && \textit{[universal one-point rule, twice]}\\
= {} & [\, x + 1 > x \land y = y \,] && \textit{[arithmetic and reflection]}\\
= {} & true &&
\end{aligned}$$

And so, the refinement is valid. □

As a first example of the definition of a programming constructor, we consider conditionals. Hoare & He use an infix syntax for the conditional operator, and define it as follows.

$$\begin{aligned}
P \lhd b \rhd Q \;&\widehat{=}\; (b \land P) \lor (\neg\, b \land Q) && \textit{if } \alpha b \subseteq \alpha P = \alpha Q\\
\alpha(P \lhd b \rhd Q) \;&\widehat{=}\; \alpha P &&
\end{aligned}$$

Informally, $P \lhd b \rhd Q$ means P if b else Q.

The presentation of conditional as an infix operator allows the formulation of many laws in a helpful way.

L1 $P \lhd b \rhd P = P$ *idempotence*

L2 $P \lhd b \rhd Q = Q \lhd \neg\, b \rhd P$ *symmetry*

L3 $(P \lhd b \rhd Q) \lhd c \rhd R = P \lhd b \wedge c \rhd (Q \lhd c \rhd R)$ *associativity*

L4 $P \lhd b \rhd (Q \lhd c \rhd R) = (P \lhd b \rhd Q) \lhd c \rhd (P \lhd b \rhd R)$ *distributivity*

L5 $P \lhd \mathit{true} \rhd Q = P = Q \lhd \mathit{false} \rhd P$ *unit*

L6 $P \lhd b \rhd (Q \lhd b \rhd R) = P \lhd b \rhd R$ *unreachable branch*

L7 $P \lhd b \rhd (P \lhd c \rhd Q) = P \lhd b \vee c \rhd Q$ *disjunction*

L8 $(P \odot Q) \lhd b \rhd (R \odot S) = (P \lhd b \rhd R) \odot (Q \lhd b \rhd S)$ *interchange*

In the Interchange Law (*L8*), the symbol $\odot$ stands for any truth-functional operator.

For each operator, Hoare & He give a definition followed by a number of algebraic laws as those above. These laws can be proved from the definition. As an example, we present the proof of the Unreachable Branch Law (*L6*).

Example 3 (Proof of Unreachable Branch (L6)).

$$\begin{aligned}
& (P \lhd b \rhd (Q \lhd b \rhd R)) && [L2] \\
={} & ((Q \lhd b \rhd R) \lhd \neg\, b \rhd P) && [L3] \\
={} & (Q \lhd b \wedge \neg\, b \rhd (R \lhd \neg\, b \rhd P)) && [\text{propositional calculus}] \\
={} & (Q \lhd \mathit{false} \rhd (R \lhd \neg\, b \rhd P)) && [L5] \\
={} & (R \lhd \neg\, b \rhd P) && [L2] \\
={} & (P \lhd b \rhd R) && \Box
\end{aligned}$$

Implication is, of course, still the basis for reasoning about the correctness of conditionals. We can, however, prove refinement laws that support a compositional reasoning technique.

Law 1 (Refinement to Conditional)

$$P \sqsubseteq (Q \lhd b \rhd R) \;=\; (\, P \sqsubseteq b \wedge Q \,) \wedge (\, P \sqsubseteq \neg\, b \wedge R \,) \qquad \Box$$

This result allows us to prove the correctness of a conditional by a case analysis on the correctness of each branch. Its proof is as follows.

Proof of Law 1.

$$\begin{aligned}
& P \;\sqsubseteq\; (Q \lhd b \rhd R) && [\text{definition of } \sqsubseteq] \\
={} & [\, (Q \lhd b \rhd R) \Rightarrow P \,] && [\text{definition of conditional}] \\
={} & [\, b \wedge Q \vee \neg\, b \wedge R \Rightarrow P \,] && [\text{propositional calculus}] \\
={} & [\, b \wedge Q \Rightarrow P \,] \wedge [\, \neg\, b \wedge R \Rightarrow P \,] && [\text{definition of } \sqsubseteq\text{, twice}] \\
={} & (\, P \sqsubseteq b \wedge Q \,) \wedge (\, P \sqsubseteq \neg\, b \wedge R \,) && \Box
\end{aligned}$$

A compositional argument is also available for conjunctions.

Law 2 (Separation of Requirements)

$$((P \wedge Q) \sqsubseteq R) = (P \sqsubseteq R) \wedge (Q \sqsubseteq R) \qquad \square$$

We can prove that an implementation satisfies a conjunction of requirements by considering each conjunct separately. The omitted proof is left as an exercise for the interested reader.

Sequence is modelled as relational composition. Two relations may be composed, providing that the output alphabet of the first is the same as the input alphabet of the second, except only for the use of dashes.

$$\begin{aligned} &P(v')\ ;\ Q(v) \ \widehat{=}\ \exists\, v_0 \bullet P(v_0) \wedge Q(v_0) && \textsf{if}\ \ out\alpha P = in\alpha Q' = \{v'\} \\ &in\alpha(P(v')\ ;\ Q(v)) \ \widehat{=}\ in\alpha P \\ &out\alpha(P(v')\ ;\ Q(v)) \ \widehat{=}\ out\alpha Q \end{aligned}$$

Composition is associative and distributes backwards through the conditional.

$$\begin{aligned} &\textsf{L1} \quad P\ ;\ (Q\ ;\ R) = (P\ ;\ Q)\ ;\ R && \textsf{associativity} \\ &\textsf{L2} \quad (P \lhd b \rhd Q)\ ;\ R = ((P\ ;\ R) \lhd b \rhd (Q\ ;\ R)) && \textsf{left distribution} \end{aligned}$$

The simple proofs of these laws, and those of a few others in the sequel, are omitted for the sake of conciseness.

The definition of assignment is basically equality; we need, however, to be careful about the alphabet. If $A = \{x, y, \ldots, z\}$ and $\alpha e \subseteq A$, where αe is the set of free variables of the expression e, the assignment $x :=_A e$ of expression e to variable x changes only x's value.

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

There is a degenerate form of assignment that changes no variable: it's called "skip", and has the following definition.

$$\begin{aligned} &I\!\!I_A \ \widehat{=}\ (v' = v) && \textsf{if}\ \ A = \{v\} \\ &\alpha I\!\!I_A \ \widehat{=}\ A \cup A' \end{aligned}$$

Skip is the identity of sequence.

$$\textsf{L5} \quad P\ ;\ I\!\!I_{\alpha P} = P = I\!\!I_{\alpha P}\ ;\ P \qquad \textsf{unit}$$

We keep the numbers of the laws presented in [6] that we reproduce here.

In theories of programming, nondeterminism may arise in one of two ways: either as the result of run-time factors, such as distributed processing; or as the under-specification of implementation choices. Either way, nondeterminism is modelled by choice; the semantics is simply disjunction.

$$\begin{aligned} &P \sqcap Q \ \widehat{=}\ P \vee Q && \textsf{if}\ \ \alpha P = \alpha Q \\ &\alpha(P \sqcap Q) \ \widehat{=}\ \alpha P \end{aligned}$$

The alphabet must be the same for both arguments.

The following law gives an important property of refinement: if P is refined by Q, then offering the choice between P and Q is immaterial; conversely, if the choice between P and Q behaves exactly like P, so that the extra possibility of choosing Q does not add any extra behaviour, then Q is a refinement of P.

Law 3 (Refinement and Nondeterminism)

$$P \sqsubseteq Q \;=\; (P \sqcap Q = P) \qquad \Box$$

Proof.

$$\begin{aligned}
& P \sqcap Q = P && \textit{[antisymmetry]}\\
&= (P \sqcap Q \sqsubseteq P) \wedge (P \sqsubseteq P \sqcap Q) && \textit{[definition of } \sqsubseteq\textit{, twice]}\\
&= [\, P \Rightarrow P \sqcap Q \,] \wedge [\, P \sqcap Q \Rightarrow P \,] && \textit{[definition of } \sqcap\textit{, twice]}\\
&= [\, P \Rightarrow P \vee Q \,] \wedge [\, P \vee Q \Rightarrow P \,] && \textit{[propositional calculus]}\\
&= \mathit{true} \wedge [\, P \vee Q \Rightarrow P \,] && \textit{[propositional calculus]}\\
&= [\, Q \Rightarrow P \,] && \textit{[definition of } \sqsubseteq\textit{]}\\
&= P \sqsubseteq Q && \Box
\end{aligned}$$

Another fundamental result is that reducing nondeterminism leads to refinement.

Law 4 (Thin Nondeterminism)

$$P \sqcap Q \sqsubseteq P \qquad \Box$$

The proof is immediate from properties of the propositional calculus.

Variable blocks are split into the commands $\mathit{var}\, x$, which declares and introduces x in scope, and $\mathit{end}\, x$, which removes x from scope. Their definitions are presented below, where A is an alphabet containing x and x'.

$$\begin{aligned}
\mathit{var}\, x \;&\mathrel{\widehat=}\; (\,\exists\, x \bullet I\!I_A\,) \qquad & \alpha(\,\mathit{var}\, x\,) \;&\mathrel{\widehat=}\; A \setminus \{x\}\\
\mathit{end}\, x \;&\mathrel{\widehat=}\; (\,\exists\, x' \bullet I\!I_A\,) \qquad & \alpha(\,\mathit{end}\, x\,) \;&\mathrel{\widehat=}\; A \setminus \{x'\}
\end{aligned}$$

The relation $\mathit{var}\, x$ is not homogeneous, since it does not include x in its alphabet, but it does include x'; similarly, $\mathit{end}\, x$ includes x, but not x'.

The results below state that following a variable declaration by a program Q makes x local in Q; similarly, preceding a variable undeclaration by a program Q makes x' local.

$$\begin{aligned}
&(\,\mathit{var}\, x \,;\, Q\,) = (\,\exists\, x \bullet Q\,)\\
&(\,Q \,;\, \mathit{end}\, x\,) = (\,\exists\, x' \bullet Q\,)
\end{aligned}$$

More interestingly, we can use $\mathit{var}\, x$ and $\mathit{end}\, x$ to specify a variable block.

$$(\,\mathit{var}\, x \,;\, Q \,;\, \mathit{end}\, x\,) = (\,\exists\, x, x' \bullet Q\,)$$

In programs, we use $\mathit{var}\, x$ and $\mathit{end}\, x$ paired in this way, but the separation is useful for reasoning.

The following laws are representative.

$$\mathit{L6} \quad (\, \mathsf{var}\, x \,;\, \mathsf{end}\, x \,) = \mathit{I\!I}$$

$$\mathit{L8} \quad (\, x := e \,;\, \mathsf{end}\, x \,) = (\, \mathsf{end}\, x \,)$$

Variable blocks introduce the possibility of writing programs and equations like that below.

$$\begin{aligned}&(\, \mathsf{var}\ x \,;\, x := 2 * y \,;\, w := 0 \,;\, \mathsf{end}\ x \,)\\ &\qquad = \ (\, \mathsf{var}\ x \,;\, x := 2 * y \,;\, \mathsf{end}\ x \,) \,;\, w := 0\end{aligned}$$

Clearly, the assignment to w may be moved out of the scope of the the declaration of x, but what is the alphabet in each of the assignments to w? If the only variables are w, x, and y, and suppose that $A = \{w, y, w', y'\}$, then the assignment on the right has the alphabet A; but the alphabet of the assignment on the left must also contain x and x', since they are in scope. There is an explicit operator for making alphabet modifications such as this: *alphabet extension.* If the right-hand assignment is $P \mathrel{\widehat{=}} w :=_A 0$, then the left-hand assignment is denoted by P_{+x}.

$$P_{+x} \mathrel{\widehat{=}} P \wedge x' = x \qquad \text{for} \quad x, x' \notin \alpha P$$

$$\alpha(P_{+x}) \mathrel{\widehat{=}} \alpha P \cup \{x, x'\}$$

If Q does not mention x, then the following laws hold.

$$\begin{array}{lrcl}\mathit{L1} & \mathsf{var}\, x \,;\, Q_{+x} \,;\, P \,;\, \mathsf{end}\, x & = & Q \,;\, \mathsf{var}\, x \,;\, P \,;\, \mathsf{end}\, x\\ \mathit{L2} & \mathsf{var}\, x \,;\, P \,;\, Q_{+x} \,;\, \mathsf{end}\, x & = & \mathsf{var}\, x \,;\, P \,;\, \mathsf{end}\, x \,;\, Q\end{array}$$

Together with the laws for variable declaration and undeclaration, the laws of alphabet extension allow for program transformations that introduce new variables and assignments to them.

3 The Complete Lattice

The refinement ordering is a partial order: reflexive, anti-symmetric, and transitive. Moreover, the set of alphabetised predicates with a particular alphabet A is a complete lattice under the refinement ordering. Its bottom element is denoted $\bot_{\mathrm{A}}$, and is the weakest predicate *true*; this is the program that aborts, and behaves quite arbitrarily. The top element is denoted $\top^{A}$, and is the strongest predicate *false*; this is the program that performs miracles and implements every specification. These properties of abort and miracle are captured in the following two laws, which hold for all P with alphabet A.

$$\mathit{L1} \quad \bot_{\mathrm{A}} \sqsubseteq P \qquad \textit{bottom element}$$

$$\mathit{L2} \quad P \sqsubseteq \top_{\mathrm{A}} \qquad \textit{top element}$$

The least upper bound is not defined in terms of the relational model, but by the law *L1* below. This law alone is enough to prove laws *L1A* and *L1B*, which are actually more useful in proofs.

$\mathsf{L1}$ $P \sqsubseteq (\sqcap S)$ *iff* $(P \sqsubseteq X$ *for all* X *in* $S)$ *unbounded nondeterminism*

$\mathsf{L1A}$ $(\sqcap S) \sqsubseteq X$ *for all* X *in* S *lower bound*

$\mathsf{L1B}$ *if* $P \sqsubseteq X$ *for all* X *in* S, *then* $P \sqsubseteq (\sqcap S)$ *greatest lower bound*

These laws characterise basic properties of least upper bounds.

A function F is *monotonic* if and only if $P \sqsubseteq Q \Rightarrow F(P) \sqsubseteq F(Q)$. Operators like conditional and sequence are monotonic; negation and conjunction are not. There is a class of operators that are all monotonic.

Example 4 (Disjunctivity and monotonicity). Suppose that $P \sqsubseteq Q$ and that $\odot$ is disjunctive, or rather, $R \odot (S \sqcap T) = (R \odot S) \sqcap (R \odot T)$. From this, we can conclude that $P \odot R$ is monotonic in its first argument.

$$
\begin{aligned}
& P \odot R && \textsf{[assumption } (P \sqsubseteq Q) \textsf{ and Law 3]} \\
={}& (P \sqcap Q) \odot R && \textsf{[assumption } (\odot \textsf{ disjunctive)]} \\
={}& (P \odot R) \sqcap (Q \odot R) && \textsf{[thin nondeterminism]} \\
\sqsubseteq{}& Q \odot R
\end{aligned}
$$

A symmetric argument shows that $P \odot Q$ is also monotonic in its other argument. In summary, disjunctive operators are always monotonic. The converse is not true: monotonic operators are not always disjunctive. □

Since alphabetised relations form a complete lattice, every construction defined solely using monotonic operators has a fixed-point. Even more, a result by Tarski says that the set of fixed-points form a complete lattice themselves. The extreme points in this lattice are often of interest; for example, $\top$ is the strongest fixed-point of $X = P \mathbin{;} X$, and $\bot$ is the weakest.

The weakest fixed-point of the function F is denoted by μF, and is simply the greatest lower bound (the *weakest*) of all the fixed-points of F.

$$\mu F \mathrel{\widehat{=}} \sqcap \{ X \mid F(X) \sqsubseteq X \}$$

The strongest fixed-point νF is the dual of the weakest fixed-point.

Hoare & He use weakest fixed-points to define recursion. They write a recursive program as $\mu X \bullet \mathcal{C}(X)$, where $\mathcal{C}(X)$ is a predicate that is constructed using monotonic operators and the variable X. As opposed to the variables in the alphabet, X stands for a predicate itself, and we call it the recursive variable. Intuitively, occurrences of X in $\mathcal{C}$ stand for recursive calls to $\mathcal{C}$ itself. The definition of recursion is as follows.

$$\mu X \bullet \mathcal{C}(X) \mathrel{\widehat{=}} \mu F \qquad \textsf{\textit{where}}\ F \mathrel{\widehat{=}} \lambda X \bullet \mathcal{C}(X)$$

The standard laws that characterise weakest fixed-points are valid.

$\mathsf{L1}$ $\mu F \sqsubseteq Y$ if $F(Y) \sqsubseteq Y$ *weakest fixed-point*

$\mathsf{L2}$ $[F(\mu F) = \mu F]$ *fixed-point*

$\mathsf{L1}$ establishes that μF is weaker than any fixed-point; $\mathsf{L2}$ states that μF is itself a fixed-point. From a programming point of view, $\mathsf{L2}$ is just the copy rule.

Proof of L1.

$$\begin{array}{ll}
\quad F(Y) \sqsubseteq Y & \textit{[set comprehension]} \\
= Y \in \{\, X \mid F(X) \sqsubseteq X \,\} & \textit{[lattice law L1A]} \\
\Rightarrow \sqcap\{\, X \mid F(X) \sqsubseteq X \,\} \sqsubseteq Y & \textit{[definition of } \mu\, F\textit{]} \\
= \mu\, F \sqsubseteq Y & \Box
\end{array}$$

Proof of L2.

$$\begin{array}{ll}
\quad \mu\, F = F(\mu\, F) & \textit{[mutual refinement]} \\
= \mu\, F \sqsubseteq F(\mu\, F) \land F(\mu\, F) \sqsubseteq \mu\, F & \textit{[fixed-point law L1]} \\
\Leftarrow F(F(\mu\, F)) \sqsubseteq F(\mu\, F) \land F(\mu\, F) \sqsubseteq \mu\, F & \textit{[F monotonic]} \\
\Leftarrow F(\mu\, F) \sqsubseteq \mu\, F & \textit{[definition]} \\
= F(\mu\, F) \sqsubseteq \sqcap\{\, X \mid F(X) \sqsubseteq X \,\} & \textit{[lattice law L1B]} \\
\Leftarrow \forall\, X \in \{\, X \mid F(X) \sqsubseteq X \,\} \bullet F(\mu\, f) \sqsubseteq X & \textit{[comprehension]} \\
= \forall\, X \bullet F(X) \sqsubseteq X \Rightarrow F(\mu\, F) \sqsubseteq X & \textit{[transitivity of } \sqsubseteq\textit{]} \\
\Leftarrow \forall\, X \bullet F(X) \sqsubseteq X \Rightarrow F(\mu\, F) \sqsubseteq F(X) & \textit{[F monotonic]} \\
\Leftarrow \forall\, X \bullet F(X) \sqsubseteq X \Rightarrow \mu\, F \sqsubseteq X & \textit{[fixed-point law L1]} \\
= \mathit{true} & \Box
\end{array}$$

The while loop is written $b * P$: while b is true, execute the program P. This can be defined in terms of the weakest fixed-point of a conditional expression.

$$b * P \;\widehat{=}\; \mu\, X \bullet (\, (P \,;\, X) \lhd b \rhd \mathit{I\!I} \,)$$

Example 5 (Non-termination). If b always remains true, then obviously the loop $b * P$ never terminates, but what is the semantics for this non-termination? The simplest example of such an iteration is $\mathit{true} * \mathit{I\!I}$, which has the semantics $\mu\, X \bullet X$.

$$\begin{array}{ll}
\quad \mu\, X \bullet X & \textit{[definition of least fixed-point]} \\
= \sqcap\{\, Y \mid (\lambda\, X \bullet X)(Y) \sqsubseteq Y \,\} & \textit{[function application]} \\
= \sqcap\{\, Y \mid Y \sqsubseteq Y \,\} & \textit{[relexivity of } \sqsubseteq\textit{]} \\
= \sqcap\{\, Y \mid \mathit{true} \,\} & \textit{[property of } \sqcap\textit{]} \\
= \bot & \Box
\end{array}$$

A surprising, but simple, consequence of Example 5 is that a program can recover from a non-terminating loop!

Example 6 (Aborting loop). Suppose that the sole state variable is x and that c is a constant.

$$
\begin{array}{ll}
\quad (b * P);\ x := c & \textsf{[Example 5]} \\
= \bot;\ x := c & \textsf{[definition of } \bot\textsf{]} \\
= \textsf{true};\ x := c & \textsf{[definition of assignment]} \\
= \textsf{true};\ x' = c & \textsf{[definition of composition]} \\
= \exists\, x_0 \bullet \textsf{true} \wedge x' = c & \textsf{[predicate calculus]} \\
= x' = c & \textsf{[definition of assignment]} \\
= x := c & \square
\end{array}
$$

Example 6 is rather disconcerting: in ordinary programming, there is no recovery from a non-terminating loop. It is the purpose of *designs* to overcome this deficiency in the programming model; we return to this in Section 5.

4 Theories of Program Correctness

In this section, we apply the theory of alphabetised relations to two key ideas in imperative programming: Hoare logic and the weakest precondition calculus.

4.1 Hoare Logic

Hoare logic provides a way to decompose the correctness argument for a program. The *Hoare triple* $p \left\{ Q \right\} r$ asserts the correctness of program Q against the specification with precondition p and postcondition r:

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

The logical rules for Hoare logic are very famous. We reproduce some below.

$$
\begin{array}{lll}
\textsf{L1} & \textsf{if } p \left\{ Q \right\} r \textsf{ and } p \left\{ Q \right\} s \textsf{ then } p \left\{ Q \right\} (r \wedge s) & \\
\textsf{L3} & \textsf{if } \ p \left\{ Q \right\} r \ \textsf{ then } \ (p \wedge q) \left\{ Q \right\} (r \vee s) & \\
\textsf{L4} & r(e) \left\{ x := e \right\} r(x) & \textsf{assignment} \\
\textsf{L6} & \textsf{if } p \left\{ Q_1 \right\} s \textsf{ and } s \left\{ Q_2 \right\} r \textsf{ then } p \left\{ \, Q_1 \ ; \ Q_2 \, \right\} r & \textsf{sequence} \\
\textsf{L8} & \textsf{if } (b \wedge c) \left\{ Q \right\} c \textsf{ then } c \left\{ \nu X \bullet (Q \ ; X) \lhd b \rhd I\!I \right\} (\neg\, b \wedge c) & \textsf{iteration} \\
\textsf{L9} & \mathit{false} \left\{ Q \right\} r \textsf{ and } p \left\{ Q \right\} \mathit{true} \textsf{ and } p \left\{ \textsf{false} \right\} \mathit{false} \textsf{ and } p \left\{ I\!I \right\} p &
\end{array}
$$

The proof rule for iteration uses *strongest fixed-points*. The implications of this are explained below. First, we present a proof for the rule.

Proof of L8. Suppose that $(b \wedge c) \left\{ Q \right\} c$, and let Y be the overall specification, so that $Y \mathrel{\widehat{=}} c \Rightarrow \neg\, b' \wedge c'$.

$$
\begin{array}{ll}
\quad c \left\{ \nu X \bullet (Q \ ; X) \lhd b \rhd I\!I \right\} (\neg\, b \wedge c) & \textsf{[definition of Hoare triple]} \\
= \ Y \sqsubseteq \nu X \bullet (Q \ ; X) \lhd b \rhd I\!I & \textsf{[strongest fixed-point L1]}
\end{array}
$$

$$\begin{aligned}
&\Leftarrow Y \sqsubseteq (Q \,;\, Y) \lhd b \rhd \mathbb{I} && \textit{[refinement to conditional (Example 1)]}\\
&= (\, Y \sqsubseteq (b \wedge Q) \,;\, Y \,) \wedge (\, Y \sqsubseteq \neg\, b \wedge \mathbb{I} \,) && \textit{[definition of } \sqsubseteq\textit{]}\\
&= (\, Y \sqsubseteq (b \wedge Q) \,;\, Y \,) \wedge [\neg\, b \wedge \mathbb{I} \Rightarrow (c \Rightarrow \neg\, b' \wedge c')\,]\\
& && \textit{[propositional calculus \& definition of } \mathbb{I}\textit{]}\\
&= (\, Y \sqsubseteq (b \wedge Q) \,;\, Y \,) \wedge \mathit{true} && \textit{[definition of Hoare triple]}\\
&= c \left\{\, b \wedge Q \,;\, Y \,\right\} (\neg\, b \wedge c) && \textit{[sequential composition (Hoare L6)]}\\
&\Leftarrow (\, c \left\{\, b \wedge Q \,\right\} c \,) \wedge (\, c \left\{\, c \Rightarrow \neg\, b' \wedge c' \,\right\} \neg\, b \wedge c \,)\\
& && \textit{[assumption and predicate calculus]}\\
&= \mathit{true} && \Box
\end{aligned}$$

This simple proof is the advantage in defining the semantics of a loop using the strongest fixed-point. The next example shows its disadvantage.

Example 7 (Non-termination and Hoare logic).

$$\begin{aligned}
&\quad p \left\{\mathit{true} * \mathbb{I}\right\} q && \textit{[strongest fixed-point semantics]}\\
&= p \left\{\nu X \bullet (\mathbb{I} \,;\, X) \lhd \mathit{true} \rhd \mathbb{I}\right\} q && \textit{[strongest fixed-point]}\\
&= p \left\{\top\right\} q && \textit{[definition of Hoare triple]}\\
&= ((p \Rightarrow q') \;\sqsubseteq\; \top) && \textit{[top element]}\\
&= \mathit{true} && \Box
\end{aligned}$$

This shows that a non-terminating loop is identified with miracle, and so implements any specification. This drawback is the motivation for choosing weakest fixed-points as the semantics of recursion. We have already seen, however, that this also leads to problems.

An example on the use of Hoare logic is presented below.

Example 8 (Hoare logic proof for Swap). Consider the little program that swaps two numbers, using a temporary register.

$$\mathit{Swap} \;\widehat{=}\; t := a;\; a := b;\; b := t$$

A simple specification for *Swap* names the initial values of a and b, and then requires that they be swapped. The correctness assertion is therefore given by the Hoare triple below.

$$a = A \wedge b = B \left\{t := a;\; a := b;\; b := t\right\} a = B \wedge b = A$$

This assertion can be discharged using the rules of Hoare logic. First, we apply the rule for sequence *L6* to decompose the problem into two parts corresponding to the two sub-programs $t := a;\; a := b$ and $b := t$. This involves inventing an assertion for the state that exists between these two programs. Our choice is $a = B \wedge t = A$ to reflect the fact that a now has the value of b, and t holds the original value of a.

$$a = A \wedge b = B \left\{t := a;\; a := b;\; b := t\right\} a = B \wedge b = A$$

$$\Leftarrow \begin{pmatrix} a = A \wedge b = B \left\{ t := a;\ a := b \right\} a = B \wedge t = A & \textit{(i)} \\ \wedge & \\ a = B \wedge t = A \left\{ b := t \right\} a = B \wedge b = A & \textit{(ii)} \end{pmatrix}$$

Now we use *L6* again; this time to decompose the first sub-program *(i)*.

$$\textit{(i)} \Leftarrow \begin{pmatrix} a = A \wedge b = B \left\{ t := a \right\} b = B \wedge t = A & \textit{(iii)} \\ \wedge & \\ b = B \wedge t = A \left\{ a := b \right\} a = B \wedge t = A & \textit{(iv)} \end{pmatrix}$$

Each of the remaining assertions *(ii–iv)* is discharged by an application of the rule for assignment, *L4*. □

This example shows how the correctness argument is structured by the application of each rule. Another way of using the rules is to assert only the postcondition; the precondition may then be *calculated* using the rules. We address this topic below.

4.2 Weakest Precondition Calculus

If we fix the program and the postcondition, then we can calculate an appropriate precondition to form a valid Hoare triple. As there will typically be many such preconditions, it is useful to find just one that can lead us to the others. From Hoare Logic Law *L3*, we have that if $p \left\{ Q \right\} r$, then $(p \wedge q) \left\{ Q \right\} r$. If we find the *weakest precondition* w that satisfies the Hoare triple $w \left\{ Q \right\} r$, then this law states that every stronger precondition must also satisfy the assertion.

To find w, we must manipulate the assertion to constrain the precondition to be at least as strong as some other condition. We parametrise p, Q, and r to make their alphabets explicit. The derivation expands the definition of the triple and of refinement, so that the precondition $p(v)$ can be pushed into the antecedent of an implication. The rest of the derivation is simply tidying up.

$$\begin{aligned}
& p(v) \left\{ Q(v, v') \right\} r(v) && \textit{[definition of Hoare triple]} \\
= {} & (p(v) \Rightarrow r(v')) \sqsubseteq Q(v, v') && \textit{[definition of } \sqsubseteq\textit{]} \\
= {} & [\, Q(v, v') \Rightarrow (\, p(v) \Rightarrow r(v') \,)\,] && \textit{[trading antecedents]} \\
= {} & [\, p(v) \Rightarrow (\, Q(v, v') \Rightarrow r(v') \,)\,] && \textit{[restricting the quantification of } v'\textit{]} \\
= {} & [\, p(v) \Rightarrow (\forall\, v' \bullet Q(v, v') \Rightarrow r' \,)\,] && \textit{[De Morgan's quantifier rule]} \\
= {} & [\, p(v) \Rightarrow \neg\, (\, \exists\, v' \bullet Q(v, v') \wedge \neg\, r(v') \,)\,] && \textit{[change of variable]} \\
= {} & [\, p(v) \Rightarrow \neg\, (\, \exists\, v_0 \bullet Q(v, v_0) \wedge \neg\, r(v_0) \,)\,] && \textit{[definition of sequence]} \\
= {} & [\, p(v) \Rightarrow \neg\, (\, Q(v, v')\ ;\ \neg\, r(v) \,)\,]
\end{aligned}$$

This says that if p holds, then it is impossible for Q to arrive in a state where r fails to hold. *Every* precondition must have this property; including, of course, $\neg\, (Q\ ;\ \neg\, r)$ itself. We can summarise this derivation as follows.

if $w = \neg\, (\, Q\ ;\ \neg\, r \,)$ *then* $w \left\{ Q \right\} r$

The condition w is the *weakest solution* for the precondition for program Q to be guaranteed to achieve postcondition r. This useful result motivates and justifies the definition of weakest precondition.

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

The laws below state the standard weakest precondition semantics for the programming operators.

$$\begin{array}{lll} \mathit{L1} & ((x := e)\,\mathit{wp}\,r(x)) = r(e) & \mathit{assignment} \\ \mathit{L2} & ((P \,;\, Q)\,\mathit{wp}\,r) = (P\,\mathit{wp}(Q\,\mathit{wp}\,r)) & \mathit{sequential\ composition} \\ \mathit{L3} & ((P \lhd b \rhd Q)\,\mathit{wp}\,r) = ((P\,\mathit{wp}\,r) \lhd b \rhd (Q\,\mathit{wp}\,r)) & \mathit{conditional} \\ \mathit{L4} & ((P \sqcap Q)\,\mathit{wp}\,r) = (P\,\mathit{wp}\,r) \wedge (Q\,\mathit{wp}\,r) & \mathit{nondeterministic\ choice} \end{array}$$

Weakest precondition and Hoare logic, however, do not solve the pending issue of non-termination, to which we turn our attention now.

5 Designs

The problem pointed out in Section 2 can be explained as the failure of general alphabetised predicates P to satisfy the equation below.

$$\mathit{true} \,;\, P \;=\; \mathit{true}$$

In particular, in Example 6 we presented a non-terminating loop which, when followed by an assignment, behaves like the assignment. Operationally, it is as though the non-terminating loop could be ignored.

The solution is to consider a subset of the alphabetised predicates in which a particular observational variable, called *okay*, is used to record information about the start and termination of programs. The above equation holds for predicates P in this set. As an aside, we observe that *false* cannot possibly belong to this set, since $\mathit{false} = \mathit{false} \,;\, \mathit{true}$.

The predicates in this set are called designs. They can be split into precondition-postcondition pairs, and are in the same spirit as specification statements used in refinement calculi. As such, they are a basis for unifying languages and methods like B [1], VDM [7], Z, and refinement calculi [8,2,9].

In designs, *okay* records that the program has started, and okay' records that it has terminated. These are auxiliary variables, in the sense that they appear in a design's alphabet, but they never appear in code or in preconditions and postconditions.

In implementing a design, we are allowed to assume that the precondition holds, but we have to fulfill the postcondition. In addition, we can rely on the program being started, but we must ensure that the program terminates. If the precondition does not hold, or the program does not start, we are not committed to establish the postcondition nor even to make the program terminate.

A design with precondition P and postcondition Q, for predicates P and Q not containing *okay* or *okay′*, is written $(\, P \vdash Q \,)$. It is defined as follows.

$$(\, P \vdash Q \,) \;\widehat{=}\; (\, okay \land P \Rightarrow okay' \land Q \,)$$

If the program starts in a state satisfying P, then it will terminate, and on termination Q will be true.

Abort and miracle are defined as designs in the following examples. Abort has precondition *false* and is never guaranteed to terminate.

Example 9 (Abort).

$$\begin{array}{ll}
\; \mathit{false} \vdash \mathit{false} & \text{[definition of design]} \\
= okay \land \mathit{false} \Rightarrow okay' \land \mathit{false} & \text{[false zero for conjunction]} \\
= \mathit{false} \Rightarrow okay' \land \mathit{false} & \text{[vacuous implication]} \\
= \mathit{true} & \text{[vacuous impliciation]} \\
= \mathit{false} \Rightarrow okay' \land \mathit{true} & \text{[false zero for conjunction]} \\
= okay \land \mathit{false} \Rightarrow okay' \land \mathit{true} & \text{[definition of design]} \\
= \mathit{false} \vdash \mathit{true} & \square
\end{array}$$

Miracle has precondition *true*, and establishes the impossible: *false*.

Example 10 (Miracle).

$$\begin{array}{ll}
\; \mathit{true} \vdash \mathit{false} & \text{[definition of design]} \\
= okay \land \mathit{true} \Rightarrow okay' \land \mathit{false} & \text{[true unit for conjunction]} \\
= okay \Rightarrow \mathit{false} & \text{[contradiction]} \\
= \neg\, okay & \square
\end{array}$$

A reassuring result about a design is the fact that refinement amounts to either weakening the precondition, or strengthening the postcondition in the presence of the precondition. This is established by the result below.

Law 5 *Refinement of Designs*

$$P_1 \vdash Q_1 \sqsubseteq P_2 \vdash Q_2 \;=\; [\, P_1 \land Q_2 \Rightarrow Q_1 \,] \land [\, P_1 \Rightarrow P_2 \,] \qquad \square$$

Proof.

$$\begin{array}{ll}
\; P_1 \vdash Q_1 \;\sqsubseteq\; P_2 \vdash Q_2 & \text{[definition of } \sqsubseteq\text{]} \\
= [\, (\, P_2 \vdash Q_2 \,) \Rightarrow (\, P_1 \vdash Q_1 \,) \,] & \text{[definition of design, twice]} \\
= [\, (\, okay \land P_2 \Rightarrow okay' \land Q_2 \,) \Rightarrow (\, okay \land P_1 \Rightarrow okay' \land Q_1 \,) \,] & \\
 & \text{[case analysis on } okay\text{]} \\
= [\, (\, P_2 \Rightarrow okay' \land Q_2 \,) \Rightarrow (\, P_1 \Rightarrow okay' \land Q_1 \,) \,] & \text{[case analysis on } okay'\text{]}
\end{array}$$

$$\begin{aligned}
&= [\,(\,(\,P_2 \Rightarrow Q_2\,) \Rightarrow (\,P_1 \Rightarrow Q_1\,)\,) \wedge (\,\neg\, P_2 \Rightarrow \neg\, P_1\,)\,] && \textsf{\textit{[propositional calculus]}}\\
&= [\,(\,(\,P_2 \Rightarrow Q_2\,) \Rightarrow (\,P_1 \Rightarrow Q_1\,)\,) \wedge (\,P_1 \Rightarrow P_2\,)\,] && \textsf{\textit{[predicate calculus]}}\\
&= [\,P_1 \wedge Q_2 \Rightarrow Q_1\,] \wedge [\,P_1 \Rightarrow P_2\,] && \square
\end{aligned}$$

The most important result, however, is that abort is a zero for sequence. This was, after all, the whole point for the introduction of designs.

$$\textsf{\textit{L1}} \quad \textsf{\textit{true}} \;;\; (P \vdash Q) = \textsf{\textit{true}} \qquad\qquad \textsf{\textit{left-zero}}$$

Proof.

$$\begin{aligned}
&\quad \textsf{\textit{true}} \;;\; (P \vdash Q) && \textsf{\textit{[property of sequential composition]}}\\
&= \exists\, okay_0 \bullet \textsf{\textit{true}} \;;\; (P \vdash Q)[okay_0/okay] && \textsf{\textit{[case analysis]}}\\
&= (\,\textsf{\textit{true}} \;;\; (P \vdash Q)[true/okay]\,) \vee (\,\textsf{\textit{true}} \;;\; (P \vdash Q)[false/okay]\,)\\
& && \textsf{\textit{[property of design]}}\\
&= (\,\textsf{\textit{true}} \;;\; (P \vdash Q)[true/okay]\,) \vee (\,\textsf{\textit{true}} \;;\; \textsf{\textit{true}}\,) && \textsf{\textit{[relational calculus]}}\\
&= (\,\textsf{\textit{true}} \;;\; (P \vdash Q)[true/okay]\,) \vee \textsf{\textit{true}} && \textsf{\textit{[propositional calculus]}}\\
&= \textsf{\textit{true}} && \square
\end{aligned}$$

In this new setting, it is necessary to redefine assignment and skip, as those introduced previously are not designs.

$$\begin{aligned}
&(x := e) \;\widehat{=}\; (\,\textsf{\textit{true}} \vdash x' = e \wedge y' = y \wedge \cdots \wedge z' = z\,)\\
&\mathit{I\!I}_D \;\widehat{=}\; (\,\textsf{\textit{true}} \vdash \mathit{I\!I}\,)
\end{aligned}$$

Their existing laws hold, but it is necessary to prove them again, as their definitions changed.

$$\begin{aligned}
&\textsf{\textit{L2}} \quad (v := e \;;\; v := f(v)) = (v := f(e))\\
&\textsf{\textit{L3}} \quad (v := e \;;\; (P \lhd b(v) \rhd Q)) = ((v := e \;;\; P) \lhd b(e) \rhd (v := e \;;\; Q))\\
&\textsf{\textit{L4}} \quad (\mathit{I\!I}_D \;;\; (P \vdash Q)) = (P \vdash Q)
\end{aligned}$$

As as an example, we present the proof of *L2*.

Proof of L2.

$$\begin{aligned}
&\quad v := e \;;\; v := f(v) && \textsf{\textit{[definition of assignment, twice]}}\\
&= (\,\textsf{\textit{true}} \vdash v' = e\,) \;;\; (\,\textsf{\textit{true}} \vdash v' = f(v)\,) && \textsf{\textit{[case analysis on }} okay_0\textsf{\textit{]}}\\
&= (\,(\,\textsf{\textit{true}} \vdash v' = e\,)[true/okay'] \;;\; (\,\textsf{\textit{true}} \vdash v' = f(v)\,)[true/okay]\,) \vee\\
&\quad \neg\, okay \;;\; \textsf{\textit{true}} && \textsf{\textit{[definition of design]}}\\
&= (\,(\,okay \Rightarrow v' = e\,) \;;\; (\,okay' \wedge v' = f(v)\,)\,) \vee \neg\, okay && \textsf{\textit{[relational calculus]}}\\
&= okay \Rightarrow (\,v' = e \;;\; (\,okay' \wedge v' = f(v)\,)\,) && \textsf{\textit{[assignment composition]}}
\end{aligned}$$

$$= okay \Rightarrow okay' \land v' = f(e) \qquad \textsf{[definition of design]}$$
$$= (\, \textsf{true} \vdash v' = f(e)\,) \qquad \textsf{[definition of assignment]}$$
$$= v := f(e) \qquad \square$$

If any of the program operators are applied to designs, then the result is also a design. This follows from the laws below, for choice, conditional, sequence, and recursion. The choice between two designs is guaranteed to terminate when they both are; since either of them may be chosen, then either postcondition may be established.

$$\textsf{T1} \quad (\,(P_1 \vdash Q_1) \sqcap (P_2 \vdash Q_2)\,) = (\,P_1 \land P_2 \vdash Q_1 \lor Q_2\,)$$

If the choice between two designs depends on a condition b, then so do the precondition and the postcondition of the resulting design.

$$\begin{aligned}\textsf{T2} \quad & (\,(P_1 \vdash Q_1) \lhd b \rhd (P_2 \vdash Q_2)\,) \\ & = ((P_1 \lhd b \rhd P_2) \vdash (\,Q_1 \lhd b \rhd Q_2)\,)\end{aligned}$$

A sequence of designs $(P_1 \vdash Q_1)$ and $(P_2 \vdash Q_2)$ terminates when P_1 holds, and Q_1 is guaranteed to establish P_2. On termination, the sequence establishes the composition of the postconditions.

$$\begin{aligned}\textsf{T3} \quad & (\,(P_1 \vdash Q_1)\,;\,(P_2 \vdash Q_2)\,) \\ & = (\,(\neg\,(\neg\,P_1\,;\,\textsf{true}) \land (Q_1\,\textsf{wp}\,P_2)) \vdash (Q_1\,;\,Q_2)\,)\end{aligned}$$

Preconditions can be relations, and this fact complicates the statement of Law *T3*; if the P_1 is a condition instead, then the law is simplified as follows.

$$\textsf{T3}' \quad (\,(p_1 \vdash Q_1)\,;\,(P_2 \vdash Q_2)\,) = (\,p_1 \land (Q_1\,\textsf{wp}\,P_2)) \vdash (Q_1\,;\,Q_2)\,)$$

A recursively defined design has as its body a function on designs; as such, it can be seen as a function on precondition-postcondition pairs (X, Y). Moreover, since the result of the function is itself a design, it can be written in terms of a pair of functions F and G, one for the precondition and one for the postcondition.

As the recursive design is executed, the precondition F is required to hold over and over again. The strongest recursive precondition so obtained has to be satisfied, if we are to guarantee that the recursion terminates. Similarly, the postcondition is established over and over again, in the context of the precondition. The weakest result that can possibly be obtained is that which can be guaranteed by the recursion.

$$\textsf{T4} \quad (\,\mu\,X, Y \bullet (\,F(X, Y) \vdash G(X, Y)\,)\,) = (\,P(Q) \vdash Q\,)$$
$$\textsf{where } P(Y) = (\,\nu X \bullet F(X, Y)\,) \textsf{ and } Q = (\,\mu\,Y \bullet P(Y) \Rightarrow G(P(Y), Y)\,)$$

Further intuition comes from the realisation that we want the least refined fixed-point of the pair of functions. That comes from taking the strongest precondition, since the precondition of every refinement must be weaker, and the weakest postcondition, since the postcondition of every refinement must be stronger.

Like the set of general alphabetised predicates, designs form a complete lattice. We have already presented the top and the bottom (miracle and abort).

$$\begin{aligned} \top_{D} &\mathrel{\widehat=} (\, \mathsf{true} \vdash \mathsf{false} \,) = \neg\, okay \\ \bot_{D} &\mathrel{\widehat=} (\, \mathsf{false} \vdash \mathsf{true} \,) = \mathsf{true} \end{aligned}$$

The least upper bound and the greatest lower bound are established in the following theorem.

Theorem 1. *Meets and joins*

$$\begin{aligned} &\textstyle\bigsqcap_i (\, P_i \vdash Q_i \,) = (\, \bigwedge_i P_i \,) \vdash (\, \bigvee_i Q_i \,) \\ &\textstyle\bigsqcup_i (\, P_i \vdash Q_i \,) = (\, \bigvee_i P_i \,) \vdash (\, \bigwedge_i P_i \Rightarrow Q_i \,) \end{aligned}$$

As with the binary choice, the choice $\bigsqcap_i (\, P_i \vdash Q_i \,)$ terminates when all the designs do, and it establishes one of the possible postconditions. The least upper bound models a form of choice that is conditioned by termination: only the terminating designs can be chosen. The choice terminates if any of the designs does, and the postcondition established is that of any of the terminating designs.

6 Healthiness Conditions

Another way of characterising the set of designs is by imposing healthiness conditions on the alphabetised predicates. Hoare & He identify four healthiness conditions that they consider of interest: $\mathsf{H1}$ to $\mathsf{H4}$. We discuss each of them.

6.1 H1: Unpredictability

A relation R is $\mathsf{H1}$ healthy if and only if $R = (okay \Rightarrow R)$. This means that observations cannot be made before the program has started. A consequence is that R satisfies the left-zero and unit laws below.

$$\mathsf{true} \,;\, R = \mathsf{true} \quad \text{and} \quad \mathbb{I}_{D} \,;\, R = R$$

We now present a proof of these results.

Designs with left-units and left-zeros are $\mathsf{H1}$*.*

$$\begin{array}{lr} R & [\textsf{assumption } (\mathbb{I}_D \textsf{ is left-unit})] \\ = \mathbb{I}_{D} \,;\, R & [\mathbb{I}_D \textsf{ definition}] \\ = (\, \mathsf{true} \vdash \mathbb{I}_{D} \,) \,;\, R & [\textsf{design definition}] \\ = (\, okay \Rightarrow okay' \wedge \mathbb{I} \,) \,;\, R & [\textsf{relational calculus}] \\ = (\, \neg\, okay \,;\, R \,) \vee (\, \mathbb{I} \,;\, R \,) & [\textsf{relational calculus}] \\ = (\, \neg\, okay \,;\, \mathsf{true} \,;\, R \,) \vee (\, \mathbb{I} \,;\, R \,) & [\textsf{assumption (true is left-zero)}] \\ = \neg\, okay \vee (\, \mathbb{I} \,;\, R \,) & [\textsf{assumption } (\mathbb{I} \textsf{ is left-unit})] \\ = \neg\, okay \vee R & [\textsf{relational calculus}] \\ = okay \Rightarrow R & \Box \end{array}$$

H1 designs have a left-zero.

$$\begin{array}{ll}
\quad \mathit{true} \mathbin{;} R & \text{[assumption (R is H1)]} \\
= \mathit{true} \mathbin{;} (\, okay \Rightarrow R \,) & \text{[relational calculus]} \\
= (\, \mathit{true} \mathbin{;} \neg\, okay \,) \vee (\, \mathit{true} \mathbin{;} R \,) & \text{[relational calculus]} \\
= \mathit{true} \vee (\, \mathit{true} \mathbin{;} R \,) & \text{[relational calculus]} \\
= \mathit{true} & \square
\end{array}$$

H1 designs have a left-unit.

$$\begin{array}{ll}
\mathbb{I}_D \mathbin{;} R & \text{[definition of $\mathbb{I}_D$]} \\
= (\, \mathit{true} \vdash \mathbb{I}_D \,) \mathbin{;} R & \text{[definition of design]} \\
= (\, okay \Rightarrow okay' \wedge \mathbb{I} \,) \mathbin{;} R & \text{[relational calculus]} \\
= (\, \neg\, okay \mathbin{;} R \,) \vee (\, okay \wedge R \,) & \text{[relational calculus]} \\
= (\, \neg\, okay \mathbin{;} \mathit{true} \mathbin{;} R \,) \vee (\, okay \wedge R \,) & \text{[true is left-zero]} \\
= (\, \neg\, okay \mathbin{;} \mathit{true} \,) \vee (\, okay \wedge R \,) & \text{[relational calculus]} \\
= \neg\, okay \vee (\, okay \wedge R \,) & \text{[relational calculus]} \\
= okay \Rightarrow R & \text{[R is H1]} \\
= R & \square
\end{array}$$

This means that we could use the left-zero and unit laws to characterise *H1*.

6.2 H2: Possible Termination

The second healthiness condition is $[\, R[\mathit{false}/okay'] \Rightarrow R[\mathit{true}/okay'] \,]$. This means that if R is satisfied when $okay'$ is *false*, it is also satisfied then $okay'$ is *true*. In other words, R cannot *require* nontermination, so that it is always possible to terminate.

The designs are exactly those relations that are *H1* and *H2* healthy. First we present a proof that relations that are *H1* and *H2* healthy are designs.

H1 and H2 healthy relations are designs. Let $R^f = R[\mathit{false}/okay']$ and $R^t = R[\mathit{true}/okay']$.

$$\begin{array}{ll}
\quad R & \text{[assumption (R is H1)]} \\
= okay \Rightarrow R & \text{[propositional calculus]} \\
= okay \Rightarrow (\, \neg\, okay' \wedge R^f \,) \vee (\, okay' \wedge R^t \,) & \text{[assumption (R is H2)]} \\
= okay \Rightarrow (\, \neg\, okay' \wedge R^f \wedge R^t \,) \vee (\, okay' \wedge R^t \,) & \text{[propositional calculus]} \\
= okay \Rightarrow (\, (\, (\, \neg\, okay' \wedge R^f \,) \vee okay' \,) \wedge R^t \,) & \text{[propositional calculus]} \\
= okay \Rightarrow (\, (\, R^f \vee okay' \,) \wedge R^t \,) & \text{[propositional calculus]} \\
= okay \Rightarrow (\, R^f \wedge R^t \,) \vee (\, okay' \wedge R^t \,) & \text{[assumption (R is H2)]}
\end{array}$$

$$
\begin{aligned}
&= okay \Rightarrow R^f \lor (\, okay' \land R^t \,) && \textit{[propositional calculus]}\\
&= okay \land \neg\, R^f \Rightarrow okay' \land R^t && \textit{[design definition]}\\
&= \neg\, R^f \vdash R^t && \Box
\end{aligned}
$$

It is very simple to prove that designs are *H1* healthy; we present the proof that designs are *H2* healthy.

Designs are H2.

$$
\begin{aligned}
&(\, P \vdash Q \,)[false/okay'] && \textit{[definition of design]}\\
&= (\, okay \land P \Rightarrow \mathit{false} \,) && \textit{[propositional calculus]}\\
&\Rightarrow (\, okay \land P \Rightarrow Q \,) && \textit{[definition of design]}\\
&= (\, P \vdash Q \,)[true/okay'] && \Box
\end{aligned}
$$

While *H1* characterises the rôle of *okay*, *H2* characterises *okay'*. Therefore, it should not be a surprise that, together, they identify the designs.

6.3 H3: Dischargeable Assumptions

The healthiness condition *H3* is specified as an algebraic law: $R = R \,;\, \mathbb{I}_D$. A design satisfies *H3* exactly when its precondition is a condition. This is a very desirable property, since restrictions imposed on dashed variables in a precondition can never be discharged by previous or successive components. For example, $x' = 2 \vdash true$ is a design that can either terminate and give an arbitrary value to x, or it can give the value 2 to x, in which case it is not required to terminate. This is a rather bizarre behaviour.

A design is H3 iff its assumption is a condition.

$$
\begin{aligned}
&(\,(\, P \vdash Q \,) = (\,(\, P \vdash Q \,) \,;\, \mathbb{I}_D \,)\,) && \textit{[definition of design-skip]}\\
&= (\,(\, P \vdash Q \,) = (\,(\, P \vdash Q \,) \,;\, (\, \mathit{true} \vdash \mathbb{I}_D \,)\,)\,) && \textit{[sequence of designs]}\\
&= (\,(\, P \vdash Q \,) = (\, \neg\,(\, \neg\, P \,;\, \mathit{true} \,) \land \neg\,(\, Q \,;\, \neg\, \mathit{true} \,) \vdash Q \,;\, \mathbb{I}_D \,)\,) && \textit{[skip unit]}\\
&= (\,(\, P \vdash Q \,) = (\, \neg\,(\, \neg\, P \,;\, \mathit{true} \,) \vdash Q \,)\,) && \textit{[design equality]}\\
&= (\, \neg\, P = \neg\, P \,;\, \mathit{true} \,) && \textit{[propositional calculus]}\\
&= (\, P = P \,;\, \mathit{true} \,) && \Box
\end{aligned}
$$

The final line of this proof states that $P = \exists\, v' \bullet P$, where v' is the output alphabet of P. Thus, none of the after-variables' values are relevant: P is a condition only on the before-variables.

6.4 H4: Feasibility

The final healthiness condition is also algebraic: $R \,;\, \mathit{true} = \mathit{true}$. Using the definition of sequence, we can establish that this is equivalent to $\exists\, v' \bullet R$, where

v' is the output alphabet of R. In words, this means that for *every* initial value of the observational variables on the input alphabet, there exist final values for the variables of the output alphabet: more concisely, establishing a final state is feasible. The design $\top_D$ is not *H4* healthy, since miracles are not feasible.

7 Theories of Program Correctness Revisited

In this section, we reconsider our theories of program correctness in the light of the theory of designs. We start with assertional reasoning, which we postponed until we had an adequate treatment of termination. We review Hoare logic and weakest preconditions, before introducing the refinement calculus.

7.1 Assertional Reasoning

A well-established reasoning technique for correctness is that of assertional reasoning. It uses assumptions and assertions to annotate programs: write conditions that must, or are expected to, hold in several points of the program. If the conditions do hold, assumptions and assertions do not affect the behaviour of the program; they are comments. If the condition of an assumption does not hold, the program becomes miraculous; if the condition of an assertion does not hold, the program aborts.

$$\begin{aligned} c^\top &\mathrel{\widehat=} \boldsymbol{I\!I}_D \lhd c \rhd \top_D && \textit{[assumption]} \\ c_\bot &\mathrel{\widehat=} \boldsymbol{I\!I}_D \lhd c \rhd \bot_D && \textit{[assertion]} \end{aligned}$$

For simplicity, we ignore the alphabets in these definitions.

The following law establishes that a sequence of assertions can be joined.

Law 6 (Composition of Assertions)

$$b_\bot \; ; c_\bot = (b \land c)_\bot \qquad \square$$

Proof.

$$\begin{aligned} & b_\bot \; ; c_\bot && \textit{[definition of assertion]} \\ &= (\boldsymbol{I\!I}_D \lhd b \rhd \bot_D) \; ; c_\bot && \textit{[composition left-distribution over conditional]} \\ &= ((\boldsymbol{I\!I}_D \; ; c_\bot) \lhd b \rhd (\bot_D \; ; c_\bot)) && \textit{[left-unit for sequence (L4)]} \\ &= (c_\bot \lhd b \rhd (\bot_D \; ; c_\bot)) && \textit{[}\bot_D \textit{ left-zero for sequence (L1)]} \\ &= (c_\bot \lhd b \rhd \bot_D) && \textit{[definition of assertion]} \\ &= ((\boldsymbol{I\!I}_D \lhd c \rhd \bot_D) \lhd b \rhd \bot_D) && \textit{[conditional associativity (L3)]} \\ &= (\boldsymbol{I\!I}_D \lhd b \land c \rhd \bot_D) && \textit{[definition of assertion]} \\ &= (b \land c)_\bot && \square \end{aligned}$$

Reasoning with assertions often involves distributing them through a program. For example, we can move an assertion over an assignment.

Law 7 (Assertions and Assignments)

$$(c(e)_\bot \mathbin{;} x := e) \;=\; (x := e;\; c(x)_\bot) \qquad \Box$$

Proof.

$$\begin{array}{ll}
\quad c(e)_\bot \mathbin{;} x := e & \textit{[definition of assertion]} \\
= (\mathit{I\!I}_D \lhd c(e) \rhd \bot_D) \mathbin{;} x := e & \textit{[sequence L2]} \\
= \mathit{I\!I}_D \mathbin{;} x := e \lhd c(e) \rhd \bot_D \mathbin{;} x := e & \textit{[}x := e \textit{ is a design (H1)]} \\
= x := e \lhd c(e) \rhd \bot_D & \textit{[}x := e \textit{ is H3 and H4]} \\
= x := e \mathbin{;} \mathit{I\!I}_D \lhd c(e) \rhd x := e \mathbin{;} \bot_D & \textit{[design assignment Law L3]} \\
= x := e \mathbin{;} (\mathit{I\!I}_D \lhd c(x) \rhd \bot_D) & \textit{[definition of assertion]} \\
= x := e;\; c(x)_\bot & \Box
\end{array}$$

Finally, we present below a law for distributing assertions through a conditional.

Law 8 (Assertions and Conditionals)

$$c_\bot \mathbin{;} (P \lhd b \rhd Q) \;=\; ((b \wedge c)_\bot \mathbin{;} P \lhd b \rhd (\neg\, b \wedge c)_\bot \mathbin{;} Q) \qquad \Box$$

We leave the proof of this law as as exercise.

7.2 Hoare Logic

In Section 4, we define the Hoare triple for relations as follows.

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

The next two examples show that this is not appropriate for designs. First, we consider which specifications are satisfied by an aborting program.

Example 11 (Abort).

$$\begin{array}{ll}
\quad p \left\{ \bot_D \right\}_D r & \textit{[definition of } \bot_D\textit{]} \\
= p \left\{ \mathit{false} \vdash \mathit{true} \right\} r & \textit{[definition of design]} \\
= p \left\{ \mathit{okay} \wedge \mathit{false} \Rightarrow \mathit{okay}' \wedge \mathit{true} \right\} r & \textit{[propositional calculus]} \\
= p \left\{ \mathit{true} \right\}_D r & \textit{[definition of Hoare triple]} \\
= [\, \mathit{true} \Rightarrow (p \Rightarrow r') \,] & \textit{[propositional calculus]} \\
= [\, p \Rightarrow r' \,] & \Box
\end{array}$$

This is simply wrong, since it establishes the validity of

$$\mathit{true} \left\{ \bot_D \right\} \mathit{true}$$

Here the requirement is for the program to terminate in *every* state—which abort clearly does fails to do. Next, we consider which specifications are satisfied by a miraculous program.

Example 12 (Miracle).

$$\begin{array}{lr}
\ p \left\{ \top_{D} \right\}_{D} r & [\textit{definition of } \top_{D}] \\
= p \left\{ \mathit{true} \vdash \mathit{false} \right\} r & [\textit{definition of design}] \\
= p \left\{ \mathit{okay} \land \mathit{true} \Rightarrow \mathit{okay}' \land \mathit{false} \right\} r & [\textit{propositional calculus}] \\
= p \left\{ \neg\, \mathit{okay} \right\} r & [\textit{definition of Hoare triple}] \\
= [\, \neg\, \mathit{okay} \Rightarrow (p \Rightarrow r') \,] & [\textit{case analysis}] \\
= [\, \neg\, \mathit{true} \Rightarrow (p \Rightarrow r') \,] \land [\, \neg\, \mathit{false} \Rightarrow (p \Rightarrow r') \,] & [\textit{propositional calculus}] \\
= [\, p \Rightarrow r' \,] & [\textit{from Example 11}] \\
= p \left\{ \bot_{D} \right\}_{D} r & \Box
\end{array}$$

Again,this is simply wrong, since it is the same result as before—and a miracle is surely different from an aborting program! So, we conclude that we need to adjust the definition of Hoare triple for designs. For any design Q, we define the Hoare triple as follows.

$$p \left\{ Q \right\}_{D} r \ \ \widehat{=}\ \ (p \vdash r') \sqsubseteq Q$$

If we replay our two examples, we get the expected results. First, what specifications are satisfied by an aborting program?

Example 13 (Abortive implementation).

$$\begin{array}{lr}
\ p \left\{ \bot_{D} \right\}_{D} r & [\textit{definition of } \bot_{D}] \\
= p \left\{ \mathit{false} \vdash \mathit{true} \right\} r & [\textit{definition of Hoare triple}] \\
= [\, (\, \mathit{false} \vdash \mathit{true} \,) \Rightarrow (\, p \vdash r' \,) \,] & [\textit{definition of design, twice}] \\
= [\, (\, p \Rightarrow \mathit{false} \,) \land (\, p \land \mathit{true} \Rightarrow r \,) \,] & [\textit{propositional calculus}] \\
= [\, \neg\, p \land (\, p \Rightarrow r' \,) \,] & [\textit{propositional calculus}] \\
= [\, \neg\, p \,] & \Box
\end{array}$$

The answer is that the precondition must be a contradiction. Next, what specifications are satisfied by a miraculous program?

Example 14 (Miraculous implementation).

$$\begin{array}{lr}
\ p \left\{ \top_{D} \right\}_{D} r & [\textit{definition of } \top_{D}] \\
= p \left\{ \mathit{true} \vdash \mathit{false} \right\} r & [\textit{definition of Hoare triple}] \\
= [\, (\, \mathit{true} \vdash \mathit{false} \,) \Rightarrow (\, p \vdash r' \,) \,] & [\textit{definition of design, twice}] \\
= [\, (\, p \Rightarrow \mathit{true} \,) \land (\, p \land \mathit{false} \Rightarrow r' \,) \,] & [\textit{propositional calculus}] \\
= [\, \mathit{true} \,] & [\textit{predicate calculus}] \\
= \mathit{true} & \Box
\end{array}$$

The answer is that a miracle satisfies *every* specification.
We now prove that Hoare logic rule *L1* holds for the new definition.

$$
\begin{aligned}
& (p \left\{ Q \right\}_{D} r) \wedge (p \left\{ Q \right\}_{D} s) && \textit{[definition of design Hoare triple]} \\
={} & ((p \vdash r') \sqsubseteq Q) \wedge ((p \vdash s') \sqsubseteq Q) && \textit{[separation of requirements]} \\
={} & (p \vdash r') \wedge (p \vdash s') \ \sqsubseteq \ Q && \textit{[definition of designs]} \\
={} & (okay \wedge p \Rightarrow okay' \wedge r') \wedge (okay \wedge p \Rightarrow okay' \wedge s') \ \sqsubseteq \ Q && \\
& && \textit{[propositional calculus]} \\
={} & (okay \wedge p \Rightarrow okay' \wedge okay' \wedge r' \wedge s') \ \sqsubseteq \ Q && \textit{[definition of design]} \\
={} & (p \vdash r' \wedge s') \ \sqsubseteq \ Q && \textit{[definition of design Hoare triple]} \\
={} & p \left\{ Q \right\}_{D} r \wedge s && \Box
\end{aligned}
$$

Other rules may be proved in a similar way.

7.3 Weakest Precondition

Once more, we can use our definition of a Hoare triple to derive an expression for the weakest precondition of *H3* healthy designs.

$$
\begin{aligned}
& p \left\{ P \vdash Q \right\}_{D} r && \textit{[definition of Hoare triple]} \\
={} & (p \vdash r') \sqsubseteq (P \vdash Q) && \textit{[refinement of designs]} \\
={} & [\,(p \Rightarrow P) \wedge (p \wedge Q \Rightarrow r')\,] && \textit{[propositional calculus]} \\
={} & [\,p \Rightarrow P \wedge (Q \Rightarrow r')\,] && \textit{[predicate calculus]} \\
={} & [\,\forall\, v' \bullet p \Rightarrow P \wedge (Q \Rightarrow r')\,] && \textit{[since } v' \textit{ not free in } a\textit{]} \\
={} & [\,p \Rightarrow \forall\, v' \bullet P \wedge (Q \Rightarrow r')\,] && \textit{[assumption (} P \vdash Q \textit{ is H3): } v' \textit{ not free in } P\textit{]} \\
={} & [\,p \Rightarrow P \wedge (\forall\, v' \bullet Q \Rightarrow r')\,] && \textit{[De Morgan's quantifier law]} \\
={} & [\,p \Rightarrow P \wedge \neg\,(\exists\, v' \bullet \neg\,(Q \Rightarrow r'))\,] && \textit{[propositional calculus]} \\
={} & [\,p \Rightarrow P \wedge \neg\,(\exists\, v' \bullet Q \wedge \neg\, r')\,] && \textit{[definition of sequential composition]} \\
={} & [\,p \Rightarrow P \wedge \neg\,(Q \,;\, \neg\, r)\,] && \Box
\end{aligned}
$$

This motivates our new definition for the weakest precondition for a design.

$$(\, P \vdash Q \,)\ \mathit{wp}_{D}\ r = (\, P \wedge (Q\ \mathit{wp}\ r)\,)$$

This new definition uses the *wp* operator introduced before.

7.4 Specification Statements

Our final theory of program correctness is Morgan's refinement calculus [8]. There, a *specification statement* is a kind of design. The syntax is as follows.

$$frame \ : \ [\, precondition,\ postcondition \,]$$

The *frame* describes the variables that are allowed to change, and the *precondition* and *postcondition* are the same as those in a design.

For example, the specification statement $x : [\, y \neq 0, x' = x \,\mathsf{div}\, y \,]$ is represented by the design $y \neq 0 \vdash (\, x' = x \,\mathsf{div}\, y \,)_{+y}$, providing that the only program variables are x and y.
The refinement law for assignment introduction is as shown below.

Law 9 *Assignment Introduction in the Refinement Calculus*

$$\textit{providing that} \quad [\, p \Rightarrow Q[e/w'] \,]$$
$$w, x : [\, p, Q \,] \;\sqsubseteq\; w := e \qquad \square$$

Proof.

$$\begin{aligned}
& w := e && \textit{[definition of assignment]}\\
&= \textit{true} \vdash (\, w' = e \,)_{+\{x,y\}} && \textit{[weaken precondition]}\\
&\sqsupseteq p \vdash (\, w' = e \,)_{+\{x,y\}} && \textit{[assumption]}\\
&= p \vdash (\, Q[e/w'] \wedge w' = e \,)_{+\{x,y\}} && \textit{[Leibniz]}\\
&= p \vdash (\, Q \wedge w' = e \,)_{+\{x,y\}} && \textit{[alphabet extension]}\\
&= p \vdash (\, Q \wedge w' = e \wedge x' = x \,)_{+y} && \textit{[strengthen postcondition]}\\
&\sqsupseteq p \vdash Q_{+y} && \textit{[definition of specification statement]}\\
&= w, x : [\, p, Q \,] && \square
\end{aligned}$$

Another important law allows the calculation of conditionals.

Law 10 *Conditional Introduction in the Refinement Calculus*

$$\textit{providing that} \quad [\, p \Rightarrow \bigvee i \bullet g_i \,]$$

$$w : [\, p, Q \,] \;\sqsubseteq\; \left\{ \begin{array}{l}
\textit{if } g_1 \longrightarrow w : [\, g_1 \wedge p, Q \,]\\
\square\; g_2 \longrightarrow w : [\, g_2 \wedge p, Q \,]\\
\vdots\\
\square\; g_n \longrightarrow w : [\, g_n \wedge p, Q \,]\\
\textit{fi}
\end{array}\right.$$

$\square$

This law uses a generalised form of conditional, present in Dijkstra's language of guarded commands [4] and in Morgan's calculus. The conditions are called guards, and the choice of branch to execute is nondeterministic among those whose guards are true. The definition of this guarded conditional is not difficult, but here we consider just the conditional operator we have presented before.

Proof of binary case : $w : [\, p, Q \,] \sqsubseteq (w : [\, g \wedge p, Q \,] \lhd g \rhd w : [\, \neg\, g \wedge p, Q \,])$.

In order to prove this refinement, we can resort to Law 1 and proceed by case analysis. In this case, we need to prove $w : [\, p, Q \,] \sqsubseteq g \wedge w : [\, g \wedge p, Q \,]$ and $w : [\, p, Q \,] \sqsubseteq \neg\, g \wedge w : [\, \neg\, g \wedge p, Q \,]$. Below, we prove the first case; the second case is similar.

$$\begin{aligned}
& g \wedge w : [\, g \wedge p, Q \,] && \textsf{\textit{[definition of specification statement]}}\\
=\ & g \wedge (\, g \wedge p \vdash Q_{+x} \,) && \textsf{\textit{[definition of alphabet extension]}}\\
=\ & g \wedge (\, g \wedge p \vdash Q \wedge x' = x \,) && \textsf{\textit{[definition of design]}}\\
=\ & g \wedge (\, \mathit{okay} \wedge g \wedge p \Rightarrow \mathit{okay}' \wedge Q \wedge x' = x \,) && \textsf{\textit{[propositional calculus]}}\\
=\ & g \wedge (\, \mathit{okay} \wedge p \Rightarrow \mathit{okay}' \wedge Q \wedge x' = x \,) && \textsf{\textit{[propositional calculus]}}\\
\Rightarrow\ & (\, \mathit{okay} \wedge p \Rightarrow \mathit{okay}' \wedge Q \wedge x' = x \,) && \textsf{\textit{[definition of design]}}\\
=\ & p \vdash Q_{+x} && \textsf{\textit{[definition of specification statement]}}\\
=\ & w : [\, p, Q \,] && \Box
\end{aligned}$$

Next, we present an example of the application of the refinement calculus. The problem is to calculate the maximum and minimum of two numbers.

Example 15 (Finding the maximum and the minimum). The problem has a simple specification: $x, y : [\, \mathit{true}, x' = \mathit{max}(x, y) \wedge y' = \mathit{min}(x, y) \,]$. Our first step is to use Law 10 to introduce a conditional statement that checks the order of the two variables.

$$\begin{aligned}
& x, y : [\, \mathit{true}, x' = \mathit{max}(x, y) \wedge y' = \mathit{min}(x, y) \,]\\
\sqsubseteq\ & \left(\begin{array}{c}
x, y : [\, x < y, x' = \mathit{max}(x, y) \wedge y' = \mathit{min}(x, y) \,] \quad \textsf{\textit{(i)}}\\
\lhd\ x < y\ \rhd\\
x, y : [\, x \geq y, x' = \mathit{max}(x, y) \wedge y' = \mathit{min}(x, y) \,] \quad \textsf{\textit{(ii)}}
\end{array}\right)
\end{aligned}$$

The 'then' case is easily implemented using a multiple assignment, a generalised assignment that updates a list of variables in parallel. Its semantics and properties are similar to those of the single assignment; in particular, Law 9 holds.

$$\textsf{\textit{(i)}} \ \sqsubseteq \ x, y := y, x$$

providing that

$$[\, x < y \Rightarrow (\, x' = \mathit{max}(x, y) \wedge y' = \mathit{min}(x, y) \,)[y, x / x', y'] \,]$$

which follows from substitution and properties of the *max* and *min* functions. The 'else' case is even simpler, since the variables are already in the right order.

$$(ii) \ \sqsubseteq \ \textsf{\textit{skip}}$$

providing that

$$[\, x \geq y \Rightarrow (\, x' = \mathit{max}(x, y) \wedge y' = \mathit{min}(x, y) \,)[x, y / x', y'] \,]$$

The development may be summarised as the following refinement.

$$
\begin{array}{ll}
& x, y : [\, true, x' = max(x, y) \land y' = min(x, y) \,] \\
\sqsubseteq & \\
& \mathsf{\textit{if}}\ x < y\ \mathsf{\textit{then}}\ x, y := y, x\ \mathsf{\textit{else\ skip\ fi}} \\
= & \\
& \mathsf{\textit{if}}\ x < y\ \mathsf{\textit{then}}\ x, y := y, x\ \mathsf{\textit{fi}}
\end{array}
\qquad \Box
$$

There are many other laws in the refinement calculus; we omit them for the sake of conciseness.

8 Conclusions

Through a series of examples, we have presented the alphabetised relational calculus and its sub-theory of designs. In this framework, we have presented the formalisation of four different techniques for reasoning about program correctness. The assertional technique, the Hoare logic, and the weakest preconditions are presented in [6]; our original contribution is a recasting of Hoare logic and weakest preconditions in the theory of designs, and an outline of the formalisation of Morgan's calculus.

We hope to have given a didactic and accessible account of this basic foundation of the unifying theories of programming. We have left out, however, most of the more elaborate programming constructs contemplated in [6]. These include theories for concurrency, communication, and functional, logic, and higher-order programming. We also have not discussed their account of algebraic and operational semantics, nor the correctness of compilers.

In our recent work, we have used the theory of communication and concurrency to provide a semantics for *Circus* [13], an integration of Z and CSP [11] aimed at supporting the development of reactive concurrent systems. We have used the semantics to justify a refinement strategy for *Circus* based on calculational laws in the style of Morgan [3].

In [10], UTP is also used to give a semantics to another integration of Z and CSP, which also includes object-oriented features. In [12], UTP is extended with constructs to capture real-time properties as a first step towards a semantic model for a timed version of *Circus*. In [5], a theory of general correctness is characterised as an alternative to designs; instead of *H1* and *H2*, a different healthiness condition is adopted to restrict general relations.

Currently, we are collaborating with colleagues to extend UTP to capture mobility, synchronicity, and object orientation. We hope to contribute to the development of a theory that can support all the major concepts available in modern programming languages.

References

1. J-R. Abrial. *The B-Book: Assigning Progams to Meanings*. Cambridge University Press, 1996.
2. R. J. R. Back and J. Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.
3. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2 – 3):146 – 181, 2003.
4. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
5. S. Dunne. Recasting Hoare and He's Unifying Theories of Programs in the Context of General Correctness. In A. Butterfield and C. Pahl, editors, *IWFM'01: 5th Irish Workshop in Formal Methods*, BCS Electronic Workshops in Computing, Dublin, Ireland, July 2001.
6. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.
7. C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 1986.
8. C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.
9. J. M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9(3):287 – 306, 1987.
10. S. Qin, J. S. Dong, and W. N. Chin. A Semantic Foundation for TCOZ in Unifying Theories of Programming. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 321 – 340, 2003.
11. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.
12. A. Sherif and He Jifeng. Towards a Time Model for *Circus*. In *International Conference in Formal Engineering Methods*, pages 613 – 624, 2002.
13. J. C. P. Woodcock and A. L. C. Cavalcanti. The Semantics of *Circus*. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184—203. Springer-Verlag, 2002.
14. J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof.* Prentice-Hall, 1996.