Mechanised Theory Engineering in Isabelle

Simon FOSTER and Jim WOODCOCK

Department of Computer Science, University of York, UK

Abstract

This is an introduction to mechanised theory engineering in Isabelle, an LCF-style interactive theorem prover. We introduce an embedding of Hoare & He’s Unifying Theories of Programming (UTP) in Isabelle (named Isabelle/UTP) and show how to mechanise two key theories: relations and designs. These theories are sufficient to give an account of the partial and total correctness of nondeterministic sequential programs and of networks of reactive processes. A tutorial introduction to each theory is interspersed with its formalisation and with mechanised proofs of relevant properties and theorems. The work described here underpins specification languages such as Circus, which combines state-rich imperative operations, communication and concurrency, object orientation, references and pointers, real time, and process mobility, all with denotational, axiomatic, algebraic, and operational semantics.

Keywords. Unifying Theories of Programming (UTP), Denotational Semantics, Laws of Programming, Isabelle, Interactive Theorem Proving.

Preliminaries

Unifying Theories of Programming (UTP), is originally the work of Hoare & He [47]; it is a long-term research agenda that can be summarised as follows: researchers have proposed many different programming theories and practitioners have proposed many different pragmatic programming paradigms; how do we understand the relationship between them?

UTP has been used to describe a wide variety of programming theories. In [47], 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 [23, 24, 65]; event-driven programming [50, 90, 92]; object-oriented programming [19, 67, 72]; pointer-based programming [38]; real-time programming [39, 43]; timed reactive programming [69–71, 76, 82]; and transaction programming [40, 41]. Individual programming languages have been

1Corresponding Author: jim.woodcock@york.ac.uk
given semantics in UTP. This includes the hardware description languages Handel-C [61,62] and Verilog [91]; the multi-paradigm languages Circus [14,58,59,76,88] and CML [83,87]; Safety-Critical Java [15,21,22,25,60]; and Simulink [18]. A wide variety of programming theories have been formalised in UTP, including confidentiality [7,8]; theories of testing [16,17,78]; and theories of undefinedness [6,84]. These are complemented by a collection of meta-theory, including work on higher-order UTP [89]; UTP and temporal-logic model checking [2]; and CSP as a retract of CCS [42].

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 [11,13,26,32,36,55,57,88]. 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 [35].

UTP has its origins in the work on predicative programming, which was started by Hehner; see [44] for a summary. It gives three principal ways to study the relationships between different programming paradigms: by computational paradigm; by level of abstraction; and by method of presentation.

**Computational Paradigms** How many programming languages are there? It has been estimated that there are more than 8,500, but what are the differences and commonalities amongst this great diversity? UTP groups programming languages according to a classification by computational model; for example, structured, object-oriented, functional, or logical. The technique is to identify common concepts and deal separately with additions and variations using two fundamental scientific principles: simplicity of presentation and separation of concerns.

**Abstraction** Orthogonal to this organisation by computational paradigm, languages could be categorised by their level of abstraction within a particular paradigm. For example, the lowest level of abstraction may be the platform-specific technology of an implementation. At the other end of the spectrum, there might be a very high-level description of overall requirements and how they are captured and analysed. In between, there will be descriptions of components and descriptions of how they will be organised into architectures. Each of these levels will have interfaces specified by contracts of some kind. UTP gives ways of mapping between these levels based on a formal notion of refinement that provides guarantees of correctness all the way from requirements to code.

**Presentation** The third classification is by the method chosen to present a language definition. There are three widely used scientific methods: (i) *Denotational*, in which each syntactic phrase is given a single mathematical meaning, a specification is just a set of denotations, and refinement is a simple correctness criterion of inclusion: every program behaviour is also a specification behaviour. (ii) *Algebraic*, where no direct meaning is given to the language, but instead equalities relate different programs with the same meaning. (iii) *Operational* where programs are defined by how they execute on an idealised abstract mathematical machine, giving a useful guide for compilation, debugging, and testing. As Hoare & He point out [47], a comprehensive account of a programming theory needs all three kinds of presentation, and the UTP technique allows us to study differences
and mutual embeddings, and to derive each from the others by mathematical definition, calculation, and proof.

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.

A key concept in UTP is the design: the familiar precondition-postcondition pair that describes the contract between a programmer and a client. Great use of this construct is made in the semantics of the Circus family of languages [59,80], where reactive processes are given a precondition-postcondition semantics that is then useful in assertional reasoning about state-rich reactive behaviour. Parts of this introduction are adapted from [86], and the tutorial is an extension of our previous tutorial on the UTP in Isabelle [36].

In Section 1, we introduce the basic concepts of UTP: alphabets, signatures, and healthiness conditions, and in Section 2 we outline the idea of theory mechanisation in Isabelle and show why it is so important to practical theory development in the UTP. In Section 3 we introduce our mechanisation of the UTP semantic framework, Isabelle/UTP [35]. In Section 4, we introduce a basic nondeterministic programming language and its laws of programming. In Section 5, we complete the initial presentation of UTP by describing the organisation of UTP theories into complete lattices. Sections 6 and 7 show how Hoare logic and the weakest precondition calculus can be defined in UTP. Section 8 introduces the UTP theory of designs that capture the notion of total correctness using assumptions and commitments. Section 9 introduces the design healthiness conditions and show how they characterise the theory of designs. We end with a discussion of related work (Section 10) and some conclusions including directions for future work (Section 11).

1. Introduction to UTP

Hoare & He's technique is to isolate important language features, and give them a denotational semantics; algebraic, axiomatic, and operational semantics can then be proved sound against this model. 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 [73,81] 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, a program with variables $x$, $y$, and $z$ would contain these names in its alphabet. Theories for particular programming paradigms require the observation of extra information; some examples are: a flag that says whether the program has...
started (ok); the current time (clock); the number of available resources (res); a
trace of the events in the life of the program (tr); a set of refused events (ref); or
a flag that says whether the program is waiting for interaction with its environ-
ment (wait). The signature gives the rules for the syntax for denoting objects
of the theory. For instance, in a theory of imperative programming this would
include operators like sequential composition, assignment, if-then-else, and iter-
ation. Healthiness conditions identify properties that characterise the predicates
of the theory. Each healthiness condition embodies an important fact about the
computational model for the programs being studied.

Example 1 (Healthiness conditions (Hoare & He))

1. The variable clock gives us an observation of the current time, which moves
ever onwards. The predicate \( c \) specifies this: \( C \subseteq \text{clock} \leq \text{clock}' \). If we add
\( C \) 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 \( P \Rightarrow C \).
2. The variable ok 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:
\( P = (\text{ok} \Rightarrow P) \). If the program has not started, its behaviour is not described.

Healthiness conditions can often be expressed as a function \( \phi \) that makes a program
healthy. There is no point in applying \( \phi \) twice, since we cannot make a healthy
program even healthier, so \( \phi \) must be idempotent: \( P = \phi(\phi(P)) \); this equation
characterises the healthiness condition. We can turn the first healthiness condition
above into an equivalent equation, \( P = P \land C \), and then the following function
on predicates and \( C \) is the required idempotent.

Example 2 (Boyle’s Law) Consider a simple theory to model the behaviour of a
gas with regard to varying temperature and pressure. The physical phenomenon
of the behaviour of the gas is subject to Boyle’s Law: “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)”. The alphabet of our theory
contains the three mathematical variables described in Boyle’s Law: \( k, p, \) and \( V \).
The model’s observations correspond to real-world observations in what we might
term the model-based agenda: the variables \( k, p, \) and \( V \) are shared with the real
world. We now need to describe the syntax used to denote objects of the theory.
There is a requirement that temperature remains constant, so, to use our model
to simulate the effects of Boyle’s law, we need just two operations, one to change
the pressure and one change the volume. We know the observations we can make
of our theory and the two operations we can use to change these observations. We
now need to define some healthiness conditions as a way of determining member-
ship of the theory. We are interested only in gases that obey Boyle’s law, which
states that \( p \ast V = k \) must be invariant. Healthiness conditions determine the
correct states of the system, and here we need both static and dynamic invariants:

- The equation \( p \ast V = k \) is a static invariant: it applies to a state.
We also require \( k \) to be constant. If we start in the state \((k, p, V)\), where \( p * V = k \), then transit to the state \((k', p', V')\), where \( p' * V' = k' \), then we must have that \( k' = k \). This is a dynamic invariant: it applies to a relation.

Suppose we have \( \alpha(\phi) = \{p, V, k\} \); then define \( B(\phi) = (\exists k \cdot \phi) \land (k = p * V) \). Now, regardless of whether \( \phi \) is healthy or not, \( B(\phi) \) certainly is. For example:

\[
\phi = (p = 10) \land (V = 5) \land (k = 100) \\
B(\phi) = (\exists k \cdot \phi) \land (k = p * V) = (p = 10) \land (V = 5) \land (k = 50)
\]

Notice that \( B(B(\phi)) = B(\phi) \). This is known as idempotence: taking the medicine twice leaves you healthy, no more and no less so than taking the medicine only once. This gives us a simple test for healthiness: \( \phi \) is already healthy if applying \( B \) leaves it unchanged. That is, if it satisfies the equation \( \phi = B(\phi) \). In this sense, \( \phi \) is a fixed point of the idempotent function \( B \).

Consider another observation, that the pressure is between 10 and 20 Pa:

\[
\psi = (p \in [10, 20]) \land (V = 5).
\]

Clearly, \( \phi \Rightarrow \psi \). If we make both \( \phi \) and \( \psi \) healthy, we discover another fact: \( B(\phi) \Rightarrow B(\psi) \). In fact, \( B \) is monotonic in the sense that \( \forall \phi, \psi \cdot (\phi \Rightarrow \psi) \Rightarrow (B(\phi) \Rightarrow B(\psi)) \). The most useful healthiness conditions are monotonic idempotent functions, which leads to some very important mathematical properties concerning complete lattices and Galois connections.

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.

An important application of UTP is to build tool chains: tools are linked by unifying theories. Tool chains can be either longitudinal or transverse; a longitudinal tool chain involves a series of tools where the output from one tool is used as the input to another. An example is Isabelle’s sledgehammer tool (cf. Section 2), which invokes a number of other proof tools. The outputs from these tools are proofs that must be interpreted as Isabelle proofs. A transverse toolchain provides a collection of tools for a particular language. This is illustrated in Fig. 1.

For a particular language, we might like to provide a compiler, an interpreter, a model checker, a refinement calculator, and a theorem prover. The diagram shows the UTP approach to building this collection. Modern languages have heterogeneous semantics, so at the base of the diagram there is the mathematical semantics: the alphabetised relational calculus. The UTP theories for each semantic paradigm are built on top of this and linked together to form the gold standard for the language definition: its denotational semantics. Next, the operational semantics and axiomatic semantics for the language are derived from the denotational semantics so that they are consistent and complementary. The operational semantics can then be used as the basis for tools such as a compiler (code generator), an interpreter (simulation engine), and a model checker. The axiomatic semantics forms the basis for verification tools and refinement calculators.

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 [47]. Here, we present the general relational setting, and the transition to the theory of designs. Moreover we show how this semantic foundation has been mechanised in Isabelle/UTP and how this can be used as a basis for rigorous engineering of theories of programming.

In the next section, we introduce the foundational proof assistant of our mechanisation, Isabelle, and briefly demonstrate its key proof facilities. We then present the most general theory of UTP: the alphabetised predicates, and then how this is mechanised in Isabelle/UTP. In the following section, we establish that the theory of alphabetised predicates forms a complete lattice. Section 8 restricts the general theory to designs. Next, in Section 9, we present an alternative characterisation of the theory of designs using healthiness conditions. Finally, we conclude with a summary and a brief account of related work.

2. Theory Mechanisation

We have mechanised UTP in the interactive theorem prover Isabelle [54]. This allows the laws of programming to be mechanically verified, and makes them available for use in mechanical program derivation, verification, and refinement. In this section we give a brief overview of Isabelle; for a more indepth treatment please see the excellent documentation provided on the Isabelle website.\(^2\)

Interactive theorem provers (ITPs) have been built as an aid to software engineers and theoreticians who wish to prove properties of their models, specifications, and programs, for example correctness or refinement. A typical ITP enables a user to define abstract mathematical models using a variety of notions, such as datatypes, recursive functions, and inductive predicates. For example we could model the abstract syntax tree of a language like C or Java using a collection of datatypes, and then define a compilation function which translates a given program into a byte-code or machine-code representation for execution. We can then attempt to mechanically prove properties about the compile function, for example that it preserves type safety in a strongly typed language. We can prove properties about a given program, such as that no memory leaks exist.

Isabelle is a generic interactive theorem prover. Rather than simply being an implementation of a single logic, it supports various logic images which are built as axiomatic extensions to the Isabelle core. The most well used logic is HOL [37]

\(^2\)isabelle.in.tum.de/documentation.html.
(Higher Order Logic), which is based on an ML-style functional programming language and has been designed for software and hardware verification. Isabelle/HOL then refers to Isabelle instantiated with the HOL object logic, which consists of approximately twenty axioms. The remainder of HOL is constructed definitionally from this small axiomatic core, making it highly mathematically principled.

One of the defining principles of the Isabelle system is the LCF architecture, which ensures that inconsistencies cannot be introduced in a proof. In Isabelle, proofs consist of a sequence of calculations which transform the assumptions into the goal. The commands used in this transformation are called proof tactics, which help the programmer with varying degrees of automation. The LCF architecture ensures that tactics can only make steps which are grounded in the core axioms of the object logic. This then ensures that Isabelle proofs are dependable and correct with respect to the axioms. This, along with Isabelle’s highly extensible nature at all levels, makes it an ideal theorem prover for implementing the UTP.

We now give some practical examples of proof in Isabelle/HOL. We may wish to prove the simple property $\exists x. x > 6$. In Isabelle/HOL we can formalise such a property and proof in the following manner:

```isar
theorem greater_than_six: "\exists x::nat. x > 6"
  apply (rule_tac x="7" in exI)
  apply (simp)
  done
```

The `theorem` command creates a new proof goal for the given logical formula with an optional name, in this case `greater_than_six`. Logical formulae like that above are always delimited by speech-marks, which sets them apart as terms of the logic, also known as “inner-syntax”, as opposed to commands like `theorem` and `done` which are part of the “outer-syntax”. Mathematical symbols like $\exists$ can be entered into Isabelle either by typing the \LaTeX-style \texttt{\exists} or alternatively by using the Isabelle symbol catalogue.

The style of proof is called the “apply-style” proof: we execute a sequence of tactics which variously divide and conquer the proof in an effort to discharge all goals and subgoals. There are two steps to this simple proof. We first invoke a rule called `exI` which performs existential introduction: we explicitly supply a value for $x$ for which the property holds, in this case $7$. This leaves us with the proof goal $7 > 6$, which can be dispatched by simple arithmetic properties, so we use Isabelle’s built in simplifier tactic `simp` to finish the proof. Isabelle then gives the message “No subgoals!”, which means the proof is complete and we can type `done`. At this point the property `greater_than_six` is entered into the Isabelle fact database for us to use in future proofs.

Such mechanised proofs greatly increase the confidence that a given property is true. If we try to prove something which is not true, Isabelle prevents it. For instance we can try and prove that all numbers are greater than six:

```isar
theorem all_greater_than_six: "\forall x::nat. x > 6"
  apply (rule_tac allI)
  done
```

The `theorem` command creates a new proof goal for the given logical formula with an optional name, in this case `all_greater_than_six`. Logical formulae like that above are always delimited by speech-marks, which sets them apart as terms of the logic, also known as “inner-syntax”, as opposed to commands like `theorem` and `done` which are part of the “outer-syntax”. Mathematical symbols like $\forall$ can be entered into Isabelle either by typing the \LaTeX-style \texttt{\forall} or alternatively by using the Isabelle symbol catalogue.

The style of proof is called the “apply-style” proof: we execute a sequence of tactics which variously divide and conquer the proof in an effort to discharge all goals and subgoals. There are two steps to this simple proof. We first invoke a rule called `exI` which performs existential introduction: we explicitly supply a value for $x$ for which the property holds, in this case $7$. This leaves us with the proof goal $7 > 6$, which can be dispatched by simple arithmetic properties, so we use Isabelle’s built in simplifier tactic `simp` to finish the proof. Isabelle then gives the message “No subgoals!”, which means the proof is complete and we can type `done`. At this point the property `all_greater_than_six` is entered into the Isabelle fact database for us to use in future proofs.

Such mechanised proofs greatly increase the confidence that a given property is true. If we try to prove something which is not true, Isabelle prevents it. For instance we can try and prove that all numbers are greater than six:
We cannot make much progress with such a proof – there just isn’t a tactic to perform this proof as it is unprovable. For such situations, Isabelle also contains a helpful counterexample generator called nitpick [10] which can be used to see if a property can be refuted.

```plaintext
theorem all_greater_than_six: "∀ x::nat. x > 6"
  nitpick
```

When we run this command Isabelle returns “Nitpick found a counterexample: x = 6”, which clearly shows why this proof is impossible. We therefore terminate our proof attempt by typing `oops`, which disposes of the proof goal. So Isabelle acts as a theoretician’s conscience, requiring that properties be comprehensively discharged. Due to the LCF architecture, all proofs in Isabelle are constructed with respect to the axiomatice core, even those originating from automated proof tactics. This means that proofs are not predicated on the correctness of the tools and tactics, but only on the correctness of the underlying axioms which makes Isabelle proofs trustworthy.

Such proofs can, however, be tedious for a theoretician to construct manually and therefore Isabelle provides a number of automated proof tactics to aid in proof. For instance the `greater_than_six` theorem can be proved in one step by application of Isabelle’s main automated proof method `auto`. The `auto` tactic performs introduction/elimination style classical deduction and simplification in an effort to prove a goal. The user can also extend `auto` by adding additional rules which it can make use of, increasing the scope of problems which it can deal with.

Additionally, a more recent development is the addition of the `sledgehammer` [9] tool. Sledgehammer makes use of external first-order automated theorem provers. An automated theorem prover (ATP) is a system which can provide solutions to a certain subclass of logical problems. Sledgehammer can make use of a large number of ATPs, such as E [68], Vampire [64], SPASS [77], Waldmeister [45] and Z3 [29]. During a proof the user can invoke `sledgehammer` which causes the current goal, along with relevant assumptions, to be submitted to the ATPs which attempt a proof. Upon success, a proof command is returned, which the user can insert to complete the proof.

For instance, we may wish to prove that for any given number there is an even number greater than it. We can prove this by calling `sledgehammer`:

```plaintext
theorem greater_than_y_even: "∀ y::nat. ∃ x > y. (x mod 2 = 0)"
  sledgehammer
```

In this case, sledgehammer successfully returns with ostensible proofs from four of the ATPs. We can select one of these proofs to see if it works:

```plaintext
theorem greater_than_y_even: "∀ y::nat. ∃ x > y. (x mod 2 = 0)"
  by (metis Suc_1 even_Suc even_nat_mod_two_eq_zero lessI less_SucI
    numeral_1_eq_Suc_0 numeral_One)
```

The proof command successfully discharges the goal, using seven laws from Isabelle’s standard library. `Sledgehammer` does not trust the external ATPs to re-
turn sound results, but instead reconstructs them using Isabelle’s axioms and the verified prover *metis*.

Sledgehammer works particularly well with Isabelle’s natural language proof script language *Isar*. Isar allows proof to be written in a calculational style. More information about Isar proofs can be found on the Isabelle website. The majority of proofs in tutorial are presented using Isar, as exemplified in Section 4.

3. Isabelle/UTP

3.1. Overview

Isabelle/UTP [35] is a semantic embedding of the alphabetised relational calculus. A semantic embedding mechanises a client logic by describing it in terms of another, previously mechanised host logic. This has the advantage that the client logic can reuse automated proof facilities of the host logic. We use HOL’s constructs to construct a definitional model of alphabetised predicates and concretise the notions of types and values within alphabetised predicates.

1. We define a polymorphic *value model* using an Isabelle type class that requires a HOL datatype to represent values, and a type relation of values. All our types are polymorphic over the underlying value model specified by the type parameter ‘a.

2. We define a datatype to represent variables—‘a *uvar*, which consists of a string and dashes and subscripts, and a type associated with the underlying value model.

3. We define a type of *bindings*—‘a *binding*—total mappings from variables to typed values.

4. We define the type of *core predicates*—‘a *upred*—which consists of the set of bindings that satisfy the predicate. In the predicate \( x > 7 \), \( x \mapsto 8 \) is a binding, whilst \( x \mapsto 3 \) is not.

5. We define the type of *alphabetised predicates*—‘a *uapred*—which consists of an alphabet (a finite set of variables) and a core predicate that may talk only about alphabetic variables.

Our mechanisation is thus purely semantic—at no point do we require a fixed syntax tree. Programming operators can be added incrementally by defining as and when required. We cannot directly talk about the free variables of a predicate \( P (fv(P)) \); instead we introduce a weaker semantic notion *unrestriction* over core predicates. The core predicate \( P \) is unrestricted by variable set \( xs \), written \( xs \nsubseteq P \) if \( P \) does not depend on any \( x \in xs \) for its valuation. For example if \( x \neq y \) then \( \{ y \} \nsubseteq (x > 7) \) since this predicate valuation is unaffected by \( y \). Perhaps surprisingly it is also the case that \( \{ y \} \nsubseteq (y > 7 \lor \text{true}) \)—this is because this predicate is always true no matter the value of \( y \). Thus unrestriction is strictly weaker than the free variables, but is adequate in such a purely semantic basis. An alphabetised

---

3See [www.cs.york.ac.uk/~simonf/utp-isabelle/](http://www.cs.york.ac.uk/~simonf/utp-isabelle/).
predicate pairs an alphabet $\alpha(P)$ with a core predicate $P$ that possesses the property $\alpha(P) \not\exists P$, that is $P$ can only constrain variables in its alphabet.

Similarly we introduce a semantic notion of substitution, written as usual as $P[x/e]$, for expression $v$, which behaves similarly to syntactic substitution. Expressions $(\alpha, \lambda m \ pexpr$ are also specified semantically as a function of type $(\alpha, \lambda m \ pvar$ where $\lambda m$ is the value model type and $\alpha$ is the return type. UTP expressions are therefore subject to the Isabelle type system, meaning that only well-typed constructions are permitted. Similarly, we derived a typed notion of variable—$(\alpha, \lambda m \ pvar$—that also exhibits type information.

### 3.2. Syntax matters

The syntax of Isabelle/UTP follows the standard syntax of UTP as presented throughout these notes. In order to distinguish UTP predicates from HOL predicates Isabelle/UTP has its own syntactic categories. To enter a UTP core predicate we provide backtick quotation, e.g. $P^Q$, and to enter an alphabetised predicate we provide double backtick quotation, e.g. $P_{Q}^R$. Expressions are similarly mainly written as expected; the parser supports standard numeric, list, and set constants and functions. References to variables in expressions are usually preceeded by a dollar sign, e.g. $x + 5$. A boolean typed UTP expression can be converted into a predicate using banana brackets $L M$, e.g. $L($y - $z) < 9 M$.

### 3.3. Typing

As stated, our implementation of the UTP subjects predicates to HOL typing. Thus variables and expressions carry type data meaning they cannot be composed improperly. For example attempting to enter $(\$x > 10) \land (\$x + \$y = true)$ will result in a type error. Conversely, if we simply enter $\$x = true$, Isabelle will correctly infer that $x$ is a boolean UTP variable of type $(bool, \lambda m \ pvar$.

Harnessing HOL typing in this way is useful, as it reduces the proof burden by freeing the user from discharging typing proof obligations. Nevertheless it can also be a hindrance, for example when we want to compare things of a different type. For example we might have a law of the form

$$(x := u ; y := v) = (y := v ; x := u) \quad \text{if } x \neq y \text{ and } \{x\} \not\exists v \text{ and } \{y\} \not\exists u$$

which allows us to commute two assignments. We need to compare $x$ and $y$. If they have different types, HOL equality does not permit their comparison.

To overcome this, we introduce type erasure, written $\nu \downarrow$, which removes the HOL type data from a variable or expression. If we have a variable $x$ of type $(bool, \lambda m \ pvar$ then $x \downarrow$ has type $(\lambda m \ uvar$, a UTP variable with encapsulated type data. The latter variable is not type less, the type data is still encoded within, only HOL does not “see” this type data. Thus we can compare two variables of different types by writing $x \downarrow \neq y \downarrow$, for example. Likewise we can place them in a common alphabet, $\{x \downarrow, y \downarrow, z \downarrow\}$, which has type $(\lambda m \ uvar \ set$.
3.4. Proof

Our approach to proof is to maximise reuse of HOL laws and proof tactics through proof by transfer, where we transfer a proof goal from a domain in which proof is difficult to one where automated proof is well supported. Proof about UTP core predicates can be automated by transferring goals to HOL predicates, which have a large number of associated laws. The main tactics are:

- `utp-poly-tac`—predicate calculus reasoning
- `utp-prel-tac`—relational calculus reasoning
- `utp-subst-tac`—execute substitution
- `utp-closure-tac`—discharge closure side conditions
- `utp-unrest-tac`—discharge unrestriction side conditions
- `utp-alpha-tac`—split an alphabetised predicate goal into an alphabet goal and core predicate goal
- `utp-solve`—`utp-alpha-tac + utp-poly-tac`

Many of these tactics also have a version in which `auto` is called after interpretation, for instance `utp-poly-auto-tac` is simply `utp-poly-tac` followed by `auto`. Using these tactics, proof of many goals is simplified. For example, we can solve a simple core predicate conjecture using `utp-poly-tac` as follows:

```
lemma my-lem1: "m = 5 ∧ n = 7 ⇒ m + n = 12"
by (utp-poly-auto-tac)
```

Similarly, we can discharge a substitution goal by application of the `utp-subst-tac`, which recursively applies all the substitution laws that Isabelle/UTP knows.

```
lemma my-lem2: "∀ y. x ∧ y)[true/x] = "∀ y. (true ∧ y)"
by (utp-subst-tac)
```

where variables `x` and `y` are of type boolean. With a combination of tactics we can show that this is `false`, since there is a false value for `y`:

```
lemma my-lem2': "∀ y. x ∧ y)[true/x] = 'false'
by (utp-subst-tac, utp-poly-auto-tac)
```

These tactics work on core predicates rather than alphabetised predicates because proof at the alphabetised level shows equivalence of alphabets and the property on the underlying core predicates. We make use of this property in the following proof of the law of excluded middle for alphabetised predicates:

```
lemma my-lem3: "(P ∨ ¬ P)" = "trueα(P)"
apply (utp-alpha-tac)
apply (utp-poly-tac)
done
```

We first use `utp-alpha-tac` to show that `α(P ∨ ¬ P) = α(trueα(P)) = α(P)`. Once this is discharged, we use `utp-poly-tac` to show that the underlying core predicate are equivalent. Alternatively we could just have solved this goal using `utp-solve` which combines the tactics. These tactics allow us to easily establish the basic laws of the UTP predicate and relational calculi.
Example 3 (Selection of basic predicate and relational calculus laws)

**theorem** AndA-assoc: \[ P \land (Q \land R) \equiv (P \land Q) \land R \]
by (utp-solve)

**theorem** AndA-comm: \[ P \land Q \equiv Q \land P \]
by (utp-solve)

**theorem** AndA-OrA-distr: \[ (P \lor Q) \land R \equiv (P \land R) \lor (Q \land R) \]
by (utp-solve)

**theorem** AndA-contra: \[ P \land \neg P \equiv \text{false} \]
by (utp-solve)

**theorem** ImpliesA-export: \[ P \Rightarrow Q \equiv P \Rightarrow P \land Q \]
by (utp-solve)

**theorem** SemiA-assoc: \[ P ; \text{false}_\alpha(P) \equiv \text{false}_\alpha(P) \]
by (utp-alpha-tac, simp add: alphabet-split)

Using the tactics we have constructed a large library of algebraic laws for propositional logic and relation algebra. These laws are most easily applied by application of sledgehammer, which will find the most appropriate rules to complete the step of proof. Sledgehammer works particularly well with Isabelle’s natural language proof script language Isar, which allows proof to be written in a calculational style.

4. Laws of Programming

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 [47]; 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 given simply as \( \forall a, b, a', b' \cdot P \), which is more concisely denoted as \([P]\). The correctness of a program \( P \) with respect to a specification \( S \) is denoted by \( S \subseteq P \) (\( S \) is refined by \( P \)), and is defined as \( S \subseteq P \iff \[P \Rightarrow S\] \).

In the following sections, we introduce the definitions of the constructs of a nondeterministic sequential programming language, together with their laws of programming. Each law can be proved correct as a theorem involving the denotational semantics given by its definition. The constructs are: conditional choice; sequential composition; assignment; nondeterminism; and variable blocks.

4.1. Conditional

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

\[ P \iff b \iff (b \land P) \lor (\neg b \land Q) \quad \text{iff} \quad \alpha b \subseteq \alpha P = \alpha Q \]
\[ \alpha(P \triangleleft b \triangleright Q) \equiv \alpha P \]

Informally, \( P \triangleleft b \triangleright Q \) means \( P \) if \( b \) else \( Q \). The presentation of conditional as an infix operator allows the formulation of many laws in a helpful way. In the Interchange Law, 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.

Example 4 (Proof of Unreachable Branch)

\[
\begin{align*}
(P \triangleleft b \triangleright (Q \triangleleft b \triangleright R)) & \quad \text{[L2]} \\
= ((Q \triangleleft b \triangleright R) \triangleleft \neg b \triangleright P) & \quad \text{[L3]} \\
= (Q \triangleleft b \land \neg b \triangleright (R \triangleleft \neg b \triangleright P)) & \quad \text{[propositional calculus]} \\
= (Q \triangleleft \false \triangleright (R \triangleleft \neg b \triangleright P)) & \quad \text{[L5]} \\
= (R \triangleleft \neg b \triangleright P) & \quad \text{[L2]} \\
= (P \triangleleft b \triangleright R) & \quad \text{\( \square \)}
\end{align*}
\]

This proof can be mechanised in Isar using the same sequence:

Example 5 (Isar Proof of Unreachable Branch)

\[
\begin{proof}
\begin{align*}
\text{have } \frac{\text{?lhs} = 'Q \triangleleft b \triangleright (R \triangleleft \neg b \triangleright P)' \quad \text{by (metis CondR-sym)}}{\text{also have } ... = 'Q \triangleleft b \land \neg b \triangleright (R \triangleleft \neg b \triangleright P)' \quad \text{by (metis CondR-assoc)}} \\
\text{also have } ... = 'Q \triangleleft \false \triangleright (R \triangleleft \neg b \triangleright P)' \quad \text{by (utp-poly-tac)}} \\
\text{also have } ... = \frac{\text{?rhs} \quad \text{by (metis CondR-false)}}{\text{also have } ... = 'R \triangleleft \neg b \triangleright P' \quad \text{by (metis CondR-sym)}} \\
\text{finally show } \frac{\text{?thesis} \quad \text{by (metis CondR-sym)}}{\text{qed}}
\end{align*}
\end{proof}
\]

Isar allows proof in a natural style mimicking the “pen-and-paper” proof above. The \texttt{proof} command opens an Isar proof environment for a goal, and \texttt{have} is used to create a subgoal to act as lemma for the overall goal, which must be followed by a proof, usually using the \texttt{by} command. In a calculational proof we transitively compose the previous subgoal with the next, which is done by prefixing the subgoal with \texttt{also}. Furthermore Isar provides the “…” variable which contains the right-hand side of the previous subgoal. Once all steps of the proof are complete the \texttt{finally} command collects all the subgoals together, and \texttt{show} is used to prove the overall goal. In the case that no further proof is needed the user can simply type “,” to finish. A completed proof environment is terminated with \texttt{qed}.

In this case, the proof proceeds by application of \texttt{sledgehammer} for each line where an algebraic law is applied, and by \texttt{utp-poly-tac} when propositional calculus is needed. As it happens this proof need not be manually proved, as our tactic is powerful enough to discharge it:

\[
\begin{proof}
\text{theorem } \text{CondR-unreach-branch: '}(P \triangleleft b \triangleright (Q \triangleleft b \triangleright R))' = 'P \triangleleft b \triangleright R' \quad \text{by (utp-poly-auto-tac)}
\end{proof}
\]
Explicit proofs are nevertheless valuable to giving the intuition behind a property.

4.2. Sequential composition

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.

\[
P(v') : Q(v) \cong \exists v_0 \bullet P(v_0) \land Q(v_0) \iff \text{out} P = \text{in} P' = \{v'\}
\]

\[
\text{in} P(v') : Q(v) \cong \text{in} P \quad \text{out} (P(v') : Q(v)) \cong \text{out} P
\]

4.3. Assignment

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 \( \alpha e \) 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.

\[
x :=_A e \cong (x' = e \land y' = y \land \cdots \land z' = z) \quad \alpha(x :=_A e) \cong A \cup A'
\]

There is a degenerate form of assignment that changes no variable: it is called “skip”.

4.4. Nondeterminism

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 underspecification of implementation choices. Either way, nondeterminism is modelled by choice; the semantics is simply disjunction.

\[
P \cap Q \cong P \lor Q \quad \text{if } \alpha P = \alpha Q
\]

\[
\alpha(P \cap Q) \cong \alpha P
\]

The alphabet must be the same for both arguments.

5. The complete lattice

A lattice is a partially ordered set where all non-empty finite subsets have both a least upper-bound (join) and a greatest lower-bound (meet). A complete lattice additionally requires all subsets have both a join and a meet.

**Example 6 (Complete lattice: Powerset)** The powerset of any set \( S \), ordered by inclusion, forms a complete lattice. The empty set is the least element and \( S \) itself is the greatest element. Set union is the join operation and set intersection is the meet. The powerset of \( \{0, 1, 2, 3\} \) ordered by subset, is illustrated in Figure 2. □
Example 7 (Complete lattice: Divisible natural numbers) Natural numbers ordered by divisibility form a complete lattice, where \( n \) is exactly divisible by \( m \), providing \( n \) is an exact multiple of \( m \). This gives us the following partial order: \( m \subseteq n \iff (\exists k \cdot k \times m = n) \). In this ordering, \( 1 \) is the bottom element (it exactly divides every other number) and \( 0 \) is the top element (it can be divided exactly by every other number). For example, if we restrict our attention to the numbers between \( 0 \) and \( 1 \), we obtain the lattice illustrated in Figure 2.

Let \((S, \subseteq)\) be a complete lattice. A function \( f : S \to S \) is monotonic (or order-preserving) with respect to \((S, \subseteq)\) if, for each \( x, y \in S \), whenever we have \( x \subseteq y \), then we also have \( f(x) \subseteq f(y) \). Since a complete lattice cannot be empty, it has at least a bottom element, \( \bot \). This implies that, since \( f \) is monotonic, \( f \) has at least one fixed point: whatever \( f(\bot) \) is, we must have \( \bot \subseteq f(\bot) \), and by mononicity, \( f(\bot) \subseteq f(f(\bot)) \), and so \( f(\bot) \) is a fixed point of \( f \). Fixed-point theory is used to give a semantics to recursion, so now that the existence of fixed points is guaranteed by monotonicity, what else can we say about the fixed points that do exist? We discuss this issue below.

Isabelle/HOL contains a comprehensive mechanised theory of complete lattices and fixed-points, which we directly make use of in Isabelle/UTP. We omit details of these proofs’ mechanisation; the reader can refer to the HOL library.

5.1. Lattice Operators

The refinement ordering is partial: reflexive, anti-symmetric, and transitive. The set of alphabetised predicates with a particular alphabet \( A \) is a complete lattice under the refinement ordering. Its bottom element is denoted \( \bot_A \), and is the weakest predicate \textit{true}; this is the program that aborts, and behaves quite arbitrarily. The top element is denoted \( \top_A \), and is the strongest predicate \textit{false}; this is the program that performs miracles and implements every specification.

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.

\[
L1 \quad P \subseteq (\bigcap S) \iff (P \subseteq X \text{ for all } X \text{ in } S) \quad \text{unbounded nondeterminism}
\]
Figure 3. Complete lattice of fixed points of the function \( f(s) = s \cup \{0\} \) (right).

5.2. Recursion

Since alphabetised relations form a complete lattice, every construction defined solely using monotonic operators has at least one fixed point. Even more, a result by Knaster and Tarski, described below, says that the set of fixed points form a complete lattice themselves. The extreme points in this lattice are interesting; \( \top \) is the strongest fixed-point of \( X = P \times X \), and \( \bot \) is the weakest.

**Example 8 (Complete lattice of fixed points)** Consider the function \( f(s) = s \cup \{0\} \) restricted to the domain comprising the powerset of \( \{0, 1, 2\} \). The complete lattice of fixed points for \( f \) is illustrated in Fig 3. \( \square \)

The Knaster-Tarski theorem tells us more about what the fixed points look like. First, some definitions:

- A fixed point of \( F : S \to S \) is an element \( x \in S \) such that \( F(x) = x \).
- A pre-fixed point of \( F \) is an element \( X \) such that \( X \subseteq F(X) \).
- A post-fixed point of \( F \) is an element \( x \) such that \( F(X) \subseteq X \).

**Theorem 1 (Knaster-Tarski)** Let \((S, \subseteq)\) be a complete lattice and \( F : S \to S \) be a monotonic function on \((S, \subseteq)\); then

- The function \( F \) has at least one fixed point.
- The least fixed-point of \( F \) coincides with the glb of its set of post-fixed points.
- The greatest fixed-point of \( F \) coincides with the lub of its pre-fixed points.
- The set of fixed points of \( F \) is a complete lattice under the \( \subseteq \) relation.

**Fig. 4** represents the prefixed, postfixed, and fixed points of \( F \).
Figure 4. Complete lattice of fixed points

The weakest fixed-point of the function $F$ is denoted by $\mu F$, and is defined simply as the greatest lower bound (the weakest) of all the prefixed points of $F$.

$$\mu F \equiv \bigcap \{ X \mid F(X) \sqsubseteq X \}$$

The strongest fixed-point $\nu F$ is the dual of the weakest fixed-point.

We use weakest fixed-points to define recursion, writing a recursive program as $\mu X \cdot C(X)$, where $C(X)$ is a predicate that is constructed using monotonic operators and the variable $X$. $X$ stands for a predicate itself, called the recursive variable. Intuitively, occurrences of $X$ in $C$ stand for recursive calls to $C$ itself.

$$\mu X \cdot C(X) \equiv \mu F \quad \text{where} \quad F \equiv \lambda X \cdot C(X)$$

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

- **L1** \quad $\mu F \sqsubseteq Y$ if $F(Y) \sqsubseteq Y$ \hspace{1cm} \text{weakest fixed-point}
- **L2** \quad $F(\mu F) = \mu F$ \hspace{1cm} \text{fixed-point}

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

### 5.3. Iteration

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

$$b \ast P \equiv \mu X \cdot ((P ; X) \triangleleft b \triangleright \top)$$
Example 9 (Non-termination) If $b$ always remains true, then obviously the loop $b \Rightarrow P$ never terminates, but what is the semantics for this non-termination? The simplest example of such an iteration is $true \Rightarrow II$, which has the semantics $\mu X \cdot X$.

$$
\begin{align*}
\mu X \cdot X & \quad \text{[definition of least fixed-point]} \\
= \bigcap \{ Y \mid (\lambda X \cdot X)(Y) \subseteq Y \} & \quad \text{[function application]} \\
= \bigcap \{ Y \mid Y \subseteq Y \} & \quad \text{[reflexivity of $\subseteq$]} \\
= \bigcap \{ Y \mid true \} & \quad \text{[property of $\cap$]} \\
= \bot & \\
\end{align*}
$$

A surprising, but simple, consequence of Example 9 is that a program can recover from a non-terminating loop!

Example 10 (Aborting loop) Suppose that the sole state variable is $x$ and that $c$ is a constant.

$$
\begin{align*}
(b \Rightarrow P); x := c & \quad \text{[Example 9]} \\
= \bot; x := c & \quad \text{[definition of $\bot$]} \\
= true; x := c & \quad \text{[definition of assignment, composition]} \\
= \exists x_0 \cdot true \land x' = c & \quad \text{[definition of assignment]} \\
= x' = c & \quad \text{[predicate calculus]} \\
= x := c & \quad \Box
\end{align*}
$$

Example 10 is rather disconcerting: in ordinary programming, there is no recovery from a non-terminating loop. An alternative solution would be to use the strongest fixed-point to define iteration:

$$
\begin{align*}
while b \ do \ P & \equiv \nu X \cdot \left( (P \land X) \triangleleft b \triangleright II \right) \\
\end{align*}
$$

However, with this definition non-termination while $true \ do \ P$ instead of $true$ becomes $false$, the miraculous specification, since $\nu X \cdot P; X = false$. Thus we cannot reason about non-termination in this setting, only finite iteration. It is the purpose of designs to overcome this deficiency in the programming model; we return to this in Section 8.

Nevertheless, our strongest-fixed point while loop does have some interesting properties. For example, it corresponds to the following Kleene-star-based loop:

$$
\begin{align*}
while b \ do \ P & \equiv (b \land P)^+ \land (\neg b') \\
\end{align*}
$$

Thus we can use the laws of Kleene algebra [27, 49] to reason about iteration, in particular since this algebraic hierarchy has been formalised in Isabelle [4, 34].

$$
\begin{align*}
W1 \quad & while \ true \ do \ P = false \\
W2 \quad & while \ false \ do \ P = II \\
W3 \quad & while \ b \ do \ P = P \land (while \ b \ do \ P) \triangleleft b \triangleright II \\
\end{align*}
$$
We prove **W3**, the unfolding law, in Isabelle/UTP as follows:

**theorem IterP-unfold:**
assumes \( b \in \text{COND} \quad P \in \text{REL} \)
shows \( \text{‘while } b \text{ do } P \text{ od} = (P \land b) \lor (P \land \neg b) \)’
**proof** –

also have ‘while \( b \) do \( P \) od = (while \( b \) do \( P \) od \( \land b \)) \lor (while \( b \) do \( P \) od \( \land \neg b \))’
by (metis AndP-comm WF-PREDICATE-cases)

also have ‘(\( P \land b \)) ; \( \text{while } b \text{ do } P \text{ od} \)’
by (metis IterP-cond-false IterP-cond-true assms)

also have ‘\( (P \land b) ; \text{while } b \text{ do } P \text{ od} \) < b \gg II’
by (metis CondR-def)

finally show \( ?\text{thesis} \).

qed

6. Hoare Logic

The Hoare triple \( \{ p \} Q \{ r \} \) is a specification of the correctness of a program \( Q \). Here, \( p \) and \( r \) are assertions and \( Q \) is a command. This is partial correctness in the sense that the assertions do not require \( Q \) to terminate. Instead, the correctness statement is that, if \( Q \) is started in a state satisfying \( p \), then, if it does terminate, it will finish in a state satisfying \( r \).

\[
\{ p \} Q \{ r \} \equiv [p \land Q \Rightarrow r']
\]

This is a correctness assertion that can be expressed as the refinement assertion \( (p \Rightarrow r') \sqsubseteq Q \). The Axioms of Hoare Logic can be proved from this definition:

\[ L1 \text{ if } \{ p \} Q \{ r \} \land \{ p \} Q \{ s \} \text{ then } \{ p \} Q \{ (r \land s) \} \]

\[ L2 \text{ if } \{ p \} Q \{ r \} \land \{ q \} Q \{ r \} \text{ then } \{ (p \lor q) \} Q \{ r \} \]

\[ L3 \text{ if } \{ p \} Q \{ r \} \text{ then } \{ (p \land q) \} Q \{ (r \lor s) \} \]

\[ L4 \text{ } r(x) \{ x := e \} r(x) \]

\[ L5 \text{ if } \{ (p \land b) \} Q_1 \{ r \} \land \{ (p \land \neg b) \} Q_2 \{ r \} \text{ then } \{ p \} Q_1 \ll b\gg Q_2 \{ r \} \]

\[ L6 \text{ if } \{ p \} Q_1 \{ s \} \land \{ s \} Q_2 \{ r \} \text{ then } \{ p \} Q_1 ; Q_2 \{ r \} \]

\[ L7 \text{ if } \{ p \} Q_1 \{ r \} \land \{ p \} Q_2 \{ r \} \text{ then } \{ p \} Q_1 \cap Q_2 \{ r \} \]

\[ L8 \text{ if } \{ (b \land c) \} Q \{ c \} \text{ then } \{ c \} \nu X \bullet (Q ; X) \ll b\gg (\neg b \land c) \]

\[ L9 \{ false \} Q \{ r \} \land \{ p \} Q \{ true \} \land \{ p \} false \{ false \} \]
We have implemented the Hoare triple in Isabelle/UTP and proved the laws above, which allows us to mechanically verify simple programs. For this purpose we have written some proof tactics inspired by algebraic program verification [3] which allows the semi-automated verification of program fragments. For example, consider the following Hoare triple:

\[
\{ \text{true} \} \text{ if } (x \leq y) \text{ then } z := x \text{ else } z := y \{ z = \max(x, z) \}
\]

Discharging this requires application of the conditional law (L5) and the assign law (L4). We have two tactics, hoare-cond and hoare-assign which correspond to these and also discharge associated side conditions.

**lemma Hoare-example-1:**

```
\{ true \} if (x \leq y) then z := x else z := y \{ z = \max(x, z) \}
```

The first application of hoare-cond splits this into the two assignment cases:

\[
\{ x \geq y \} z := x \{ z = \max(x, y) \}
\]
\[
\{ (x \geq y) \} z := y \{ z = \max(x, y) \}
\]

Both of these cases can easily be discharged by the assignment law. We next prove a program verification classic, the greatest common divisor.

**lemma Hoare-example-2:**

```
assumes X > 0 Y > 0
shows ' \{ \text{true} \} \{ x = \gcd(X, Y) \}
```

We have two integer constants, \(X\) and \(Y\), which are both greater than zero, and want to calculate their greatest common divisor through an iterative algorithm.
We create two program variables, x and y, to represent these values. The algorithm
iterates whilst \( x \neq y \), subtracting the lesser of x and y from the greater. When
the two become the same we have found the greatest common divisor.

To prove this we first need a loop invariant, which in this case annotates
the loop. The invariant states that in every step x and y remain positive, and
that their greatest common divisor equals that of X and Y. Once we have this
invariant we use the additional tactic, \texttt{hoare-iter}, to invoke the loop law (L8) and
then the conditional and assignment laws. The side condition of the refinement
law cannot be discharged automatically, and so we invoke sledgehammer for both
assignments, which gives two Z3 SMT proofs. Basically we are required to show
that \( gcd(x - y, y) = gcd(x, y - x) = gcd(X, Y) \).

7. Weakest Preconditions

A Hoare triple involves three variables: a precondition, a postcondition, and a
command. If we fix two of these variables, then we can calculate an extreme
solution for the third. For example, if we fix the command and the precondition,
then we calculate the strongest postcondition. Alternatively, we could fix the
command and the postcondition and calculate the weakest precondition, and that
is what we do here. We start with some relational calculus to obtain an implication
with the precondition assertion as the antecedent of an implication of the form:
\([p \Rightarrow R] \). If we fix R, then there are perhaps many solutions for p that satisfy
this inequality. Of all the possibilities, the weakest must actually be equal to R.

\[
\begin{align*}
\{ p(v) \} & \quad Q(v, v') \quad \{ r(v) \} \\
& = [ Q(v, v') \Rightarrow ( p(v) \Rightarrow r(v') ) ] \\
& = [ p(v) \Rightarrow ( Q(v, v') \Rightarrow r(v') ) ] \\
& = [ p(v) \Rightarrow ( \forall v' . Q(v, v') \Rightarrow r(v') ) ] \\
& = [ p(v) \Rightarrow \neg ( \exists v' . Q(v, v') \land \neg r(v') ) ] \\
& = [ p(v) \Rightarrow \neg ( \exists v_0 . Q(v, v_0) \land \neg r(v_0) ) ] \\
& = [ p(v) \Rightarrow \neg ( Q(v, v') ; \neg r(v) ) ]
\end{align*}
\]

Taking \( W(v) = \neg ( Q(v, v') ; \neg r(v) ) \), the Hoare triple is valid:

\[
\{ W(v) \} \quad Q(v, v') \quad \{ r(v) \}
\]

Here, \( W \) is the weakest solution for the precondition for \( Q \) to be guaranteed to
achieve \( r \).

We define the predicate transformer \( \texttt{wp} \) as a relation between \( Q \) and \( r \) as follows:

\[
Q \; \texttt{wp} \; r \; \equiv \; \neg ( Q ; \neg r )
\]

The laws for the weakest precondition operator are as follows:
A representative selection of Isabelle proofs for these rules is shown below. Most of them can be proved automatically.

**Theorem SemiR-wp**: 
\[(P ; Q) \text{ wp } R' = \text{'}P \text{ wp } (Q \text{ wp } R)\]
by (utp-rel-auto-tac)

**Theorem AssignR-wp**: 
\[x \in D \triangleleft v, x D_1 \triangleleft v R \in \text{REL}\]
\[\text{shows } (x := R v) \text{ wp } R = R[v/x]\]
using assms
by (simp add: WeakPrecondP-def AssignR-SemiR-left usubst)

**Theorem ChoiceP-wp**: 
\[(P \cap Q) \text{ wp } R' = (P \text{ wp } R) \wedge (Q \text{ wp } R)\]
by (utp-rel-auto-tac)

**Theorem ImpliesP-precond-wp**: 
\[[(R \implies S)]' \implies [(Q \text{ wp } R) \implies (Q \text{ wp } S)]'\]
by (metis ConjP-wp RefP-AndP RefP-def less-eq-upred-def)

**Theorem FalseP-wp**: 
\[Q ; \text{true}' = \text{true}' \implies Q \text{ wp } \text{false}' = \text{false}'\]
by (simp add: WeakPrecondP-def)

8. Designs

The problem pointed out in Section 5—that the relational model does not capture the semantics of nonterminating programs—can be explained as the failure of general alphabetised predicates \(P\) to satisfy the equation below.

\[true ; P = \text{true}\]

In particular, in Example 10 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 \(ok\), 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 \textit{false} cannot possibly belong to this set, since \textbf{true} : \textit{false} = \textit{false}.

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 [48], Z [81], and refinement calculi [5,51,52].

In designs, \textit{ok} records that the program has started, and \textit{ok}^0 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 \textit{ok} or \textit{ok}^0, is written $(P \vdash Q)$. It is defined as follows.

$$(P \vdash Q) \equiv (\text{ok} \land P \Rightarrow \text{ok}^0 \land Q)$$

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

### 8.1. Lattice Operators

Abort and miracle are defined as designs in the following examples. Abort has precondition \textit{false} and is never guaranteed to terminate. It is denoted by $\bot_D$. Miracle has precondition \textbf{true}, and establishes the impossible: \textit{false}. It is denoted by $\top_D$.

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

$$\top_D \equiv (\textbf{true} \vdash \textit{false}) = \neg \text{ok}$$

$$\bot_D \equiv (\textit{false} \vdash \textbf{true}) = \textbf{true}$$

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

**Theorem 2** Meets and joins

$$\cap_i(P_i \vdash Q_i) = (\land_i P_i) \vdash (\lor_i Q_i)$$

$$\cup_i(P_i \vdash Q_i) = (\lor_i P_i) \vdash (\land_i P_i \Rightarrow Q_i)$$

As with the binary choice, the choice $\cap_i(P_i \vdash Q_i)$ 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 do, and the postcondition established is that of any of the terminating designs.
8.2. Refinement of Designs

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 8.1 (Refinement of designs)

\[ P_1 \vdash Q_1 \subseteq P_2 \vdash Q_2 = [P_1 \land Q_2 \Rightarrow Q_1] \land [P_1 \Rightarrow P_2] \]

8.3. Nontermination

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.

\( L1 \) true ; \((P \vdash Q) = true\) \hspace{1cm} \text{left-zero}

8.4. Assignment

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

\( (x := e) \equiv (\text{true} \vdash x' = e \land y' = y \land \cdots \land z' = z) \)

\( \Pi_D \equiv (\text{true} \vdash \Pi) \)

8.5. Closure under the Program Combinators

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 terminate; since either of them may be chosen, then either postcondition may be established.

\( T1 \) \((P_1 \vdash Q_1) \cap (P_2 \vdash Q_2) = (P_1 \land P_2 \vdash Q_1 \lor Q_2)\)

If the choice between two designs depends on a condition \( b \), then so do the pre-condition and the postcondition of the resulting design.

\( T2 \) \((P_1 \vdash Q_1) \langle b \rangle (P_2 \vdash Q_2) \)
\[ = ((P_1 \langle b \rangle P_2) \vdash (Q_1 \langle b \rangle Q_2)) \]

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.

\( T3 \) \((P_1 \vdash Q_1) ; (P_2 \vdash Q_2) \)
\[ = ((\neg (\neg P_1 ; \text{true}) \land (Q_1 \text{wp } P_2)) \vdash (Q_1 ; Q_2)) \]
where \( Q_1 \text{	extit{wp}} P_2 \) is the weakest precondition under which execution of \( Q_1 \) is guaranteed to achieve the postcondition \( P_2 \). It is defined in [47] as

\[
Q \text{\textit{wp}} P = \neg (Q ; \neg P)
\]

The Isabelle proof of \( \textbf{T3} \) is difficult, but rewarding:

**Example 11 (Isar Proof of Design Composition)**

**Theorem** DesignD-composition:

**Assumes**

\[
(P_1 \in \text{REL}) \land (P_2 \in \text{REL}) \land (Q_1 \in \text{REL}) \land (Q_2 \in \text{REL})
\]

\[
\{ \text{ok} \} \not\subseteq P_1 \{ \text{ok} \}, \not\subseteq P_2 \{ \text{ok} \} \not\subseteq Q_1 \{ \text{ok} \}, \not\subseteq Q_2
\]

**Shows** \( (P_1 \vdash Q_1) ; (P_2 \vdash Q_2) = \neg (\neg P_1 ; \text{true} \land (Q_1 \text{\textit{wp}} P_2) \lor (Q_1 ; Q_2)) \)

**Proof**

\[
\begin{align*}
\text{have} & \ (P_1 \vdash Q_1) ; (P_2 \vdash Q_2) \\
& = \exists \ \text{ok} \quad \left( (P_1 \vdash Q_1)[\text{false}/\text{ok}] ; (P_2 \vdash Q_2)[\text{false}/\text{ok}] \right) \\
& \text{by (rule SemiR-extract-variable-ty, simp-all add: closure typing unrest assms)}
\end{align*}
\]

\[
\begin{align*}
\text{also have} & \ (P_1 \vdash Q_1)[\text{false}/\text{ok}] ; (P_2 \vdash Q_2)[\text{false}/\text{ok}] \\
& \vee (P_1 \vdash Q_1)[\text{true}/\text{ok}] ; (P_2 \vdash Q_2)[\text{true}/\text{ok}] \\
& \text{by (simp add: cases typing usubst defined closure unrest DesignD-def assms erasure inju SubstP-VarP-single-UNREST)}
\end{align*}
\]

\[
\begin{align*}
\text{also from assms} \& \ (P_1 \vdash Q_1) ; (P_2 \vdash Q_2) \land \neg (Q_1 ; P_2) \\
& \text{by simp add: typing usubst defined unrest DesignD-def OrP-comm SubstP-VarP-single-UNREST}
\end{align*}
\]

\[
\begin{align*}
\text{also have} & \ (\neg (Q_1 \land P_1) ; (P_2 \Rightarrow \text{false} \lor Q_2)) \lor (\neg (Q_1 \land P_1)) ; (P_2) \Rightarrow \text{true} \\
& \lor (\neg Q_1) ; (P_2) \Rightarrow \text{false} \lor Q_2 \\
& \text{by (simp add: OrP-comm SemiR-OrP-distr ImpliesP-def)}
\end{align*}
\]

\[
\begin{align*}
\text{also have} & \ (\neg (Q_1) ; (P_2) \Rightarrow \text{false} \lor Q_2) \\
& \text{by (simp add: OrP-distl utp-pred-simps(9))}
\end{align*}
\]

\[
\begin{align*}
\text{also have} & \ (\neg \text{ok} ; \text{false} \lor \neg P_1 ; \text{true} \lor (Q_1) ; \neg P_2) \lor (\neg \text{ok} \land (Q_1 ; Q_2)) \\
& \text{by (simp add: ImpliesP-def SemiR-OrP-distl AndP-comm SemiR-AndP-right-postcond closure)}
\end{align*}
\]

\[
\text{thus \textit{thesis} by (simp OrP-assoc SemiR-OrP-distr demorgan2)}
\]

\[
\text{qed}
\]

\[
\begin{align*}
\text{also have} & \ (\neg (\neg P_1) ; \text{true} \land (Q_1) ; \neg P_2) \\
& \text{by (simp add: SemiR-TrueP-precond closure)}
\end{align*}
\]

\[
\text{thus \textit{thesis} by (simp DesignD-def ImpliesP-def OrP-assoc demorgan2 demorgan3)}
\]
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.

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

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 post-condition 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.

$$T4 \quad (\mu X, Y \bullet (F(X, Y) \vdash G(X, Y))) = (P(Q) \vdash Q)$$

where $P(Y) = (\nu X \bullet F(X, Y))$ 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.

## 9. Healthiness conditions

We characterise the lattice of design relations: syntactically, as $P \vdash Q$; semantically, as $ok \land P \Rightarrow ok' \land Q$; and with closure conditions over programs. We could also impose healthiness conditions on alphabetised predicates. Hoare & He [47] identify four conditions of interest: $H1$ to $H4$. We discuss each of them in turn, although the first two completely characterise the lattice of designs.

### 9.1. $H1$: unpredictability

A relation $R$ is $H1$ healthy if and only if $R = (ok \Rightarrow R)$. This means that observations cannot be made before the program has started. This healthiness condition is idempotent. If $R$ is $H1$-healthy, then $R$ also satisfies the left-zero and unit laws below.
true \; R = true \quad \text{and} \quad \Pi_D \; R = R

We now present a proof of these results. First, we prove that the algebraic unit and zero properties guarantee $\Pi_1$-healthiness.

Designs with left-units and left-zeros are $\Pi_1$

\[ R = \Pi_D \; R \quad \text{[assumption (} \Pi_D \text{ is left-unit)]} \]

\[ = (\text{true} \; \Pi_D) \; R \quad \text{[design definition]} \]

\[ = (\neg \text{ok} \; R) \lor (\Pi \; R) \quad \text{[relational calculus]} \]

\[ = (\neg \text{ok} \; \text{true} \; R) \lor (\Pi \; R) \quad \text{[assumption (} \text{true} \text{ is left-zero)]} \]

\[ = \neg \text{ok} \lor (\Pi \; R) \quad \text{[assumption (} \Pi \text{ is left-unit)]} \]

\[ = \neg \text{ok} \Rightarrow R \quad \text{[relational calculus]} \]

The Isabelle proof has a few more steps, but follows a similar line of reasoning. We require that $P$ be a well-formed relation, consisting of only undashed and dashed variables. We also prefer the use of the simplifier, executed by simp, to discharge each of the steps.

**Theorem $H_1$-algebraic-intro:**

**Assumes**

'$true \; R' = 'true'$

'$\Pi_D \; R' = 'R'$

**Shows** $R$ is $H_1$

**Proof**

Let $\pi_{\forall} = \text{REL-VAR} - \text{OKAY}$

have $R = '\Pi_D \; R'$ by (simp add: assms)

also have $... = ('true \; \Pi_{\forall}) \; R'$

by (simp add:skipDA-def)

also have $... = (\forall ok \Rightarrow (\forall ok \land \Pi_{\forall})) \; R'$

by (simp add:designD-def)

also have $... = (\forall ok \Rightarrow (\forall ok \land \Pi_{\forall})) \; R'$

by (metis (hide-lams, no-types) ImpliesP-export)

also have $... = (\forall ok \Rightarrow (\forall ok \land \Pi_{\forall})) \; R'$

by (utp-poly-auto-tac)

also have $... = (\forall ok \Rightarrow \Pi) \; R'$

by (simp add:skipRA-unfold[THEN sym] skipR-as-SkipRA ImpliesP-export[THEN sym] erasure closure typing)

also have $... = ((\neg \forall ok) \; R \lor R)'$

by (simp add:ImpliesP-def SemiR-OrP-distr)

also have $... = ((\neg \forall ok) \; true \; R \lor R)'$

by (simp add:SemiR-TrueP-precond closure)

also have $... = ((\neg \forall ok) \; true \lor R)'$

by (simp add:SemiR-assoc[THEN sym] assms)

also have $... = \forall ok \Rightarrow R'$

by (simp add:SemiR-TrueP-precond closure ImpliesP-def)

finally show $?thesis$ by (simp add:is-healthy-def H1-def)

**Qed**
Next, we prove the implication the other way around: that $H1$-healthy predicates have the unit and zero properties.

$H1$ predicates have a left-zero

\[
\begin{align*}
\text{true} ; R &= \text{true} ; (\text{ok} \Rightarrow R) \quad \text{[assumption (R is H1)]} \\
&= \text{true} \quad \text{[relational calculus]} \\
&= \text{true} \quad \square
\end{align*}
\]

...and the same in Isabelle:

**theorem** $H1$-left-zero:

assumes $P$ is $H1$

shows $\text{true} ; P' = \text{true}'$

**proof** —
from *assms* have $\text{true} ; P' = \text{true} ; (\text{ok} \Rightarrow P)'$
by (utp-poly-tac)
also have ... = $\text{true} ; (\neg \text{ok} \lor P)'$
by (simp add:ImpliesP-def)
also have ... = $(\text{true}; \neg \text{ok}) \lor (\text{true} ; P)'$
by (metis SemiR-OrP-distl)
also from *assms* have ... = $\text{true} \lor (\text{true} ; P)'$
by (simp add:SemiR-precond-left-zero closure)
finally show ?thesis by simp
qed

$H1$ predicates have a left-unit

\[
\begin{align*}
\Pi_0 ; R &= (\text{true} \vdash \Pi_0) ; R \quad \text{[definition of \Pi_0]} \\
&= (\text{ok} \Rightarrow \text{ok}' \land \Pi_0) ; R \quad \text{[definition of design]} \\
&= (\neg \text{ok} ; R) \lor (\text{ok} \land R) \quad \text{[relational calculus]} \\
&= (\neg \text{ok} ; \text{true} ; R) \lor (\text{ok} \land R) \quad \text{[true is left-zero]} \\
&= (\neg \text{ok} \lor \text{true} \lor (\text{ok} \land R) \quad \text{[relational calculus]} \\
&= \neg \text{ok} \lor (\text{ok} \land R) \quad \text{[relational calculus]} \\
&= \text{ok} \Rightarrow R \quad [R \text{ is } H1] \\
&= R \quad \square
\end{align*}
\]

...and the same in Isabelle:
**Theorem H1-left-unit:**

assumes \( P \) is H1

shows \( \Pi_D \vdash P' = 'P' \)

**Proof** –

let \( 
\begin{align*}
?\text{vs} & = \text{REL-VAR - OKAY} \\
\end{align*}
\)

have \( \Pi_D \vdash P' = 'true; \Pi \vdash P' \) by (simp add:SkipDA-def)

also have ... = \( ($ok \Rightarrow $ok' \wedge \Pi \vdash P') \) by (simp add:DesignD-def)

also have ... = \( ($ok \Rightarrow $ok' \wedge \Pi \vdash P') \) by (utp-poly-tac)

also have ... = \( ($ok \Rightarrow $ok' \wedge \Pi \vdash P') \) by (simp add:VarP-EqualP-Prop aux erasure typing closure, utp-pred-tac)

also have ... = \( ($ok \Rightarrow \Pi) \vdash P' \) by (simp add:SkipR-as-SkipRA SkipRA-unfold[of ok]

ImpliesP-export THEN sym closure)

also have ... = \( 'true; \Pi \vdash P' \) by (simp add:ImpliesP-def SemiR-OrP-distr)

also have ... = \( 'true; \Pi \vdash P' \) by (simp add:SemiR-TrueP-precond closure)

also have ... = \( ($ok \Rightarrow P') \) by (metis SemiR-assoc)

also from assms have ... = \( ($ok \Rightarrow P') \) by (simp add:H1-left-zero ImpliesP-def SemiR-TrueP-precond closure)

finally show \( \text{?thesis using assms} \)

by (simp add:H1-def is-healthy-def)

qed

This means that we can use the left-zero and unit laws to exactly characterise H1 healthiness. We can assert this equivalence property in Isabelle by combining the three theorems:

**Theorem H1-algebraic:**

\( P \) is H1 \( \iff 'true; P' = 'true' \) \( \wedge (\Pi_D \vdash P' = 'P') \)

by (metis H1-algebraic-intro H1-left-unit H1-left-zero assms)

The design identity is the obvious lifting of the relational identity to a design; that is, it has precondition true and the postcondition is the relational identity. There’s a simple relationship between them: H1.

**Law 9.1 (Relational and design identities)**

\[ \Pi_D = H1(\Pi) \]

**Proof:**

\[
\begin{align*}
\Pi_D &= (\begin{array}{c}
\text{true } \vdash \Pi \\
\text{design}
\end{array}) \quad \left[\Pi_D\right] \\
&= (\begin{array}{c}
ok \Rightarrow ok' \wedge \Pi \\
\text{II, prop calculus}
\end{array}) \quad \left[\Pi, \text{prop calculus}\right] \\
&= (\begin{array}{c}
ok \Rightarrow \Pi \\
\text{H1}
\end{array}) \quad \left[H1\right] \\
&= H1(\Pi) \quad \square
\end{align*}
\]
9.2. \( \text{H2} \): possible termination

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

The healthiness condition \( \text{H2} \) is not obviously characterised by a monotonic idempotent function. We now define the idempotent \( J \) for alphabet \( \{ok, ok', v, v'\} \), and use this in an alternative definition of \( \text{H2} \).

\[ J \cong (ok \Rightarrow ok') \land v' = v \]

The most interesting property of \( J \) is the following algebraic law that allows a relation to be split into two complementary parts, one that definitely aborts and one that does not. Note the asymmetry between the two parts.

**Law 9.2 (J-split)** For all relations with \( ok \) and \( ok' \) in their alphabet,

\[ P ; J = P' \lor (P' \land ok') \]

Likewise this proof can be mechanised in Isabelle, though a little more detailed is required. In particular, we treat the equalities of each sides of the disjunction separately in the final step.

**theorem** J-split:

\[ 'P ; J' = 'P' \lor (P' \land $ok')' \]

**proof** –

let \(?vs = (REL-VAR – OKAY)\)

have \( 'P ; J' = 'P ; (($ok \Rightarrow $ok') \land \_?vs)' \)

by (simp add:JA-pred-def)

also have \( ... = 'P ; ((\neg $ok \lor ($ok \land $ok')) \land \_?vs)' \)

by (metis ImpliesP-export)

also have \( ... = 'P ; ((\neg $ok \land ($ok \land $ok')) \land \_?vs)' \)

by (utp-rel-auto-tac)

also have \( ... = (P ; (\neg $ok \land \_?vs)) \lor (P ; ($ok \land (\_?vs \land $ok'))) \)

by (smt AndP-OrP-distr AndP-assoc AndP-comm SemiR-OrP-distl)

also have \( ... = 'P' \lor (P' \land $ok')' \)

proof –

from assms have \( '(P ; (\neg $ok \land \_?vs))' = 'P' \)

by (simp add: VarP-NotP-EqualP-aux SemiR-left-one-point closure typing defined unrest urename ubsub SemiR-SkipRA-right var-dist erasure)

moreover have \( '(P ; ($ok \land (\_?vs \land $ok'))) = (P' \land $ok')' \)

proof –

have \( '(P ; ($ok \land (\_?vs \land $ok'))) = (P ; ($ok \land \_?vs)) \land $ok' \)

by (insert SemiR-TrueP-postcond[OF VarP-precond-closure[ok],simplified], simp add:AndP-assoc SemiR-AndP-right-UNDASHED unrest closure)
moreover from assms have \( '(P : (\$ok \land H_{\mathbb{R}_{\mathbb{R}}}))' = 'P' \)
by (simp add: VarP-EqualP-aux SemiR-left-one-point closure typing defined unrest rename usubst SemiR-SkipRA-right var-dist erasure)

finally show \(?thesis \).
qed

ultimately show \(?thesis by (simp)
qed

finally show \(?thesis .
qed

The two characterisations of \( H_2 \) are equivalent.

Law 9.3 (\( H_2 \) equivalence)

\[
(P = P ; J) = [P^f \Rightarrow P^t]
\]

Proof:

\[
\begin{align*}
(P = P ; J) &= (P = P^f \lor (P^t \land \$ok')) \\
&= (P = P^f \lor (P^t \land \$ok')) \land (P = P^f \lor (P^t \land \$ok'))^t \\
&= (P^f = P^t \lor (P^t \land false)) \land (P^t = P^f \lor (P^t \land true)) \\
&= (P^f = P^t) \land (P^t = P^f \lor P^t) \\
&= (P^t = P^f \lor P^t) \quad \text{[predicate calculus]}
\end{align*}
\]

\[
= [P^t \Rightarrow P^t]
\]

... and in Isabelle:

\[\text{theorem } H2\text{-equivalence:}\]
\[P \text{ is } H2 \iff 'P^f \Rightarrow P^t;\]
\[\text{proof –}\]
\[\text{have '}[P \equiv (P ; J)]' = '[(P \equiv (P^f \lor (P^t \land \$ok')))']\]
\[\text{by (simp add:J-split)}\]
\[\text{also have ... = '}'[(P \equiv P^f \lor P^t \land \$ok')] \land (P \equiv P^f \lor P^t \land \$ok')']\]
\[\text{by (simp add: ucases erasure)}\]
\[\text{also have ... = '}'[(P^f \equiv P^f) \land (P^t \equiv P^f \lor P^t)]'}\]
\[\text{by (simp add: usubst closure typing defined erasure)}\]
\[\text{also have ... = '}'[P^t \equiv (P^f \lor P^t)]'}\]
\[\text{by (utp-pred-tac)}\]
\[\text{ultimately show } \equiv\text{thesis}\]
\[\text{by (utp-pred-auto-tac)}\]
\[\text{qed}\]

\( J \) itself is \( H_2 \) healthy.
Law 9.4 \((J \text{ is } H2)\)

\[ J = H2(J) \]

**Proof:**

\[
\begin{align*}
H2(J) & = J^f \lor (J^f \land ok') \quad [J]\text{-split} \\
& = (\neg ok' \land v' = v) \lor (ok' \land v' = v) \quad \text{[propositional calculus]} \\
& = (\neg ok' \lor ok') \land v' = v \quad \text{[propositional calculus]} \\
& = (ok \Rightarrow ok') \land v' = v \quad [J] \\
& = J
\end{align*}
\]

\ldots\text{and in Isabelle:}

```isabelle

given \(\text{theoem} J\text{-is-H2:} H2(J) = J\)

given \(\text{proof -}\)

given \(\text{let } ?\gamma = (REL\text{VAR} - OKAY)\)

given \(\text{have 'H2(J)'} = 'J^f \lor (J^f \land ?ok')'\)

given \(\text{by (metis H2-def J-split)}\)

given \(\text{also have ... = '(?ok \land ?_test) \lor ?_test \land (?ok')'}}\)

given \(\text{by (simp add:JA-pred-def subst typing defined closure)}\)

given \(\text{also have ... = '(?ok \lor ?ok') \land ?_test'}\)

given \(\text{by (utp-poly-auto-tac)}\)

given \(\text{also have ... = '(!ok \Rightarrow ?ok') \land ?_test'}\)

given \(\text{by (utp-poly-tac)}\)

given \(\text{finally show } \text{?thesis}\)

given \(\text{by (metis JA-pred-def)}\)

given \(\text{qed}\)

\(J\text{ is idempotent.}\)

Law 9.5 \((H2\text{-idempotent})\)

\[ H2 \circ H2 = H2 \]

**Proof:**

\[
\begin{align*}
H2 \circ H2(P) & = (P ; J) ; J \quad [H2] \\
& = P ; (J ; J) \quad \text{[associativity]} \\
& = P ; H2(J) \quad [J \text{ } H2 \text{ healthy}] \\
& = P ; J \quad [H2] \\
& = P
\end{align*}
\]

\ldots\text{and in Isabelle:}
Theorem H2-idempotent:

\[ H2(H2(R)) = H2(R) \]

Proof:

1. Have \( H2(H2(R)) = (R ; J) \)
   - By (metis H2-def)
2. Also have \( R ; (J ; J) \)
   - By (metis SemiR-assoc)
3. Also have \( H2(J) \)
   - By (metis H2-def)
4. Also have \( R ; J \)
   - By (metis J-is-H2)
5. Also have \( (R ; J) = (R ; J ; J) \)
   - By (metis H2-def)
6. Finally show \( \text{thesis} \)

qed

Any predicate that insists on proper termination is healthy.

Example 12 (Example: H2-substitution)

\( ok' \land (x' = 0) \) is H2

Proof:

\[
\begin{align*}
(ok' \land (x' = 0))^i &\Rightarrow (ok' \land (x' = 0))^i \\
= (false \land (x' = 0) & true \land (x' = 0)) \\
= (false \Rightarrow (x') = 0)) \\
= true
\end{align*}
\]

The proof could equally well be done with the alternative characterisation of H2.

Example 13  Example: H2-J

\( ok' \land (x' = 0) \) is H2

Proof:

\[
\begin{align*}
ok' \land (x' = 0) ; J &\quad \text{[J-splitting]} \\
= (ok' \land (x' = 0))^i \lor ((ok' \land (x' = 0))^i \land ok') &\quad \text{[subst.]} \\
= (false \land (x' = 0)) \lor (true \land (x' = 0) \land ok') &\quad \text{[prop. calculus]} \\
= false \lor ((x' = 0) \land ok') &\quad \text{[propositional calculus]} \\
= ok' \land (x' = 0) \\
\end{align*}
\]

\( \Box \)

If a relation is both H1 and H2 healthy, then it is a design. We prove this by showing that the relation can be expressed syntactically as a design.
Law 9.6 (\textit{H1-H2} relations are designs)

\[
\begin{align*}
P & \Rightarrow \text{\textit{ok}} & \quad \text{[assumption: \textit{P} is \textit{H1}]} \\
ok & \Rightarrow P & \quad \text{[assumption: \textit{P} is \textit{H2}]} \\
ok & \Rightarrow P \, \triangledown \, J & \quad \text{[\textit{J-splitting}]} \\
ok & \Rightarrow P \, \lor \, (P^t \land \text{\textit{ok}'}) & \quad \text{[propositional calculus]} \\
ok \land \neg P^t & \Rightarrow \text{\textit{ok}' \land P^t} & \quad \text{[design]} \\
\neg P^t & \triangledown P^t & \quad \text{[design]}
\end{align*}
\]

Likewise this proof can be formalised in Isabelle:

\begin{verbatim}
theorem H1-H2-is-DesignD:
  assumes \textit{P} is \textit{H1} \textit{P} is \textit{H2}
  shows \textit{P} = \neg P^t \triangledown P^t,
proof 
  from \textit{assms} have \textit{P} = \textit{\textit{ok} \Rightarrow P^t}
    by (utp-poly-tac)
  also from \textit{assms} have ... = \textit{\textit{ok} \Rightarrow H2(P)^t}
    by (utp-poly-tac)
  also have ... = \textit{\textit{ok} \Rightarrow (P^t \lor (P^t \land \textit{ok}'))^t}
    by (metis H2-split \textit{assms})
  also have ... = \textit{\textit{ok} \land \neg P^t} \Rightarrow \textit{\textit{ok}' \land P^t}^t
    by (utp-poly-auto-tac)
  also have ... = \textit{\textit{\neg P^t} \triangledown P^t}^t
    by (metis DesignD-def)
  finally show \textit{?thesis}.
qed
\end{verbatim}

Designs are obviously \textit{H1}; we now show that they must also be \textit{H2}. These two results complete the proof that \textit{H1} and \textit{H2} together exactly characterise designs.

Law 9.7 \textit{Designs are H2}

\[
\begin{align*}
(\text{\textit{P} \triangledown \textit{Q}})^t & \quad \text{[definition of design]} \\
\Rightarrow (\text{\textit{ok} \land P} \Rightarrow \textit{false}) & \quad \text{[propositional calculus]} \\
\Rightarrow (\text{\textit{ok} \land P} \Rightarrow \textit{Q}) & \quad \text{[definition of design]} \\
\Rightarrow (\text{\textit{P} \triangledown \textit{Q}})^t & \quad \text{[definition of design]}
\end{align*}
\]

Miracle, even though it does not mention \textit{ok}', is \textit{H2}-healthy.

Example 14 (Miracle is \textit{H2})

\[
\begin{align*}
\neg \text{\textit{ok}} & \quad \text{[miracle]} \\
\text{\textit{true} \triangledown \textit{false}} & \quad \text{[designs are \textit{H2}]} \\
\text{\textit{H2}(true \triangledown false)} & \quad \text{[miracle]} \\
\text{\textit{H2}(\neg \text{\textit{ok}})} & \quad \text{[miracle]}
\end{align*}
\]
The final thing to prove is that it does not matter in which order we apply \(H_1\) and \(H_2\); the key point is that a design requires both properties.

**Law 9.8 (\(H_1\)-\(H_2\) commute)**

\[
\begin{align*}
H_1 \circ H_2(P) & = \text{ok} \Rightarrow P \wedge J & \text{[propositional calculus]} \\
& = \neg \text{ok} \vee P \wedge J & \text{[miracle is \(H_2\)]} \\
& = H_2(\neg \text{ok}) \vee P \wedge J & \text{[relational calculus]} \\
& = (\neg \text{ok} \vee P) \wedge J & \text{[propositional calculus]} \\
& = (\text{ok} \Rightarrow P) \wedge J & \text{[\(H_1\), \(H_2\)]} \\
& = H_2 \circ H_1(P) & \\
\end{align*}
\]

\[\Box\]

9.3. **\(H_3\): dischargeable assumptions**

The healthiness condition \(H_3\) is specified as an algebraic law: \(R = R \circ II_D\). A design satisfies \(H_3\) 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, \(x' = 2 \vdash \text{true}\) 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 \(H_3\) iff its assumption is a condition

\[
\begin{align*}
( (P \vdash Q) = ( (P \vdash Q) \circ II_D )) & = ( (P \vdash Q) = ( (P \vdash Q) \circ (\text{true} \vdash II_D) ) ) & \text{[definition of design-skip]} \\
= ( (P \vdash Q) = ( (P \vdash Q) \circ \neg \text{false} \wedge \neg (Q \circ \text{true} \vdash Q \circ II_D) ) ) & = ( (P \vdash Q) = ( \neg (P \vdash \text{true} \vdash Q) ) ) & \text{[sequence of designs]} \\
& = ( (P \vdash Q) = ( \neg (P \vdash \text{true} \vdash Q) ) ) & \text{[skip unit]} \\
& = (P = \neg P \circ \text{true} ) & \text{[propositional calculus]} \\
& = (P = P \circ \text{true} ) & \Box
\end{align*}
\]

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.

Showing that \(H_3\) is idempotent reduces to showing that \(II_D\) is idempotent, which we prove below.

**Theorem** SkipD-idempotent:

\[II_D; II_D' = II_D'\]

**Proof**

- have \(II_D; II_D' = (\text{ok} \Rightarrow II) \circ (\neg \text{ok} \Rightarrow II)'\)
  - by (metis SkipD-form)
- also have \(= \neg (\text{ok} \Rightarrow II) \circ (\neg \text{ok} \Rightarrow II)'\)
  - by (metis (no-types) ImpliesP-def SemiR-OrP-distl)
also have ... = ‘(\$(ok) \Rightarrow II \land \neg \$ok’) \lor (\$(ok) \Rightarrow II)’
by (utp-xrel-auto-tac)
also have ... = ‘((\$(ok) \Rightarrow II \land \$ok = \$ok’ \land \neg \$ok’) \lor (\$ok \Rightarrow II)’
by (utp-xrel-auto-tac)
also have ... = ‘(false ; true) \lor (\$ok \Rightarrow II)’
by (utp-poly-auto-tac)
also have ... = ‘(\$ok \Rightarrow II)’
by simp
also have ... = ‘II_p’
by (metis SkipD-form)
finally show \?thesis .
qed

Proof of \textbf{H3}’s idempotence then follows easily.

\textbf{Theorem} \textit{H3-idempotent}:
\[ H3 (H3 p) = H3 p \]
by (metis (no-types) H3-def SemiR-assoc SkipD-idempotent)

\textbf{9.4. H4: feasibility}

The final healthiness condition is also algebraic: \( R : true = true \). Using the definition of sequence, we can establish that this is equivalent to \( \exists v’ : R \), 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 \( \tau_D \) is not \textbf{H4} healthy, since miracles are not feasible. \textbf{H4} in this form is not an idempotent function, but it can be expressed as \( H4(R) = (R ; true) \Rightarrow R \).

The intuition behind this alternative definition is that \( R \) is \textbf{H4} if it remains the same when guarded by its feasibility precondition \( R ; true \). We below prove in Isabelle that these two characterisations are equivalent.

\textbf{Theorem} \textit{H4-form}:
\[ \text{assumes } P \text{ is } H4 \]
\[ \text{shows } ‘P ; true’ = ‘true’ \]

\textbf{proof} –
\[ \text{have } ‘P’ = ‘(P ; true) \Rightarrow P’ \]
\[ \text{by (metis H4-def Healthy-elim assms)} \]
\[ \text{moreover have } ‘((P ; true) \Rightarrow P) ; true’ = ‘true’ \]
\[ \text{by (utp-rel-auto-tac)} \]
\[ \text{ultimately show } ?thesis \]
\[ \text{by (simp)} \]
\[ \text{qed} \]

\textbf{Theorem} \textit{H4-equiv}:
\[ P \text{ is } H4 \quad \iff 
\quad ‘P ; true’ = ‘true’ \]
\[ \text{by (auto simp add:H4-def H4-form is-healthy-def)} \]

Moreover we can show that \textbf{H4} is idempotent, through a mixture of predicate and relational calculus manipulation.
theorem H4-idempotent:
\[ H_4(H_4(P)) = H_4(P) \]
proof
  have \( H_4(H_4(P)) = 'H_4((P ; true) ⇒ P)' \)
    by (metis H4-def)
  also have ... = \( (P ; true ⇒ P) ; true ⇒ P' \)
    by (metis AndP-comm SemiR-AndP-left-DASHED SemiR-TrueP-pre)
  also have ... = \( (P ; true ∧ P) ; true ⇒ P' \)
    by (metis ImpliesP-uncurry)
  also have ... = \( (P ; true ∧ (P ; true ⇒ P)) ; true ⇒ P' \)
    by (metis AndP-ImpliesP)
  also have ... = \( 'H_4(P)' \)
    by (metis H4-def AndP-idem)
  finally show \(?thesis\)
qed

However, unlike H1-H3, H4 is not monotonic. Consider that \( H_4(false) \) evaluates to \( true \), an abort, whilst \( H_4(x := 3) \) evaluates to \( x := 3 \), as any assignment is feasible. Similarly it follows that \( H_4(\bot_D) = true \).

theorem H4-false: \( 'H_4(false)' = 'true' \)
proof
  have \( 'H_4(false)' = 'false ⇒ false' \)
    by (metis H4-def)
  also have ... = \( false ⇒ false' \)
    by (metis SemiR-FalseP-left)
  finally show \(?thesis\)
    by simp
qed

theorem H4-TopD: \( H_4(\top_D) = true \)
by (simp add:H4-def SemiR-TrueP-precond closure)

theorem H4-assign: \( 'x := 3' \) is H4
proof
  have \( 'x := 3 ; true' = 'true' \)
    by (atp-prel-auto-tac)
  thus \(?thesis\)
    by (metis H4-equiv)
qed

Thus even though \( x := 3 \ ⊆ false \), it is not the case that \( H_4(x := 3) \ ⊆ H_4(false) \) and so H4 is not monotonic. Moreover the set of H4 predicates does not form a lattice, since for example we know that both \( x := 3 \) and \( x := 4 \) are feasible, but their conjunction is \( false \), since \( x \) cannot become both 3 and 4 simultaneously.
10. Related Work

Our mechanisation of UTP theories of relations and of designs and our future mechanisation of the theory of reactive processes form the basis for reasoning about a family of modern multi-paradigm modelling languages. This family contains both Circus [80, 85] and CML [87]: Circus combines Z [73] and CSP [46], whilst CML combines VDM [48] and CSP. Both languages are based firmly on the notion of refinement [66] and have a variety of extensions with additional computational paradigms, including real-time [70, 76], object orientation [20], synchronicity [14], and process mobility [74, 75]. Further information on Circus may be found at www.cs.york.ac.uk/circus. CML is being developed as part of the European Framework 7 COMPASS project on Comprehensive Modelling for Advanced Systems of Systems (grant Agreement: 287829); see www.compass-research.eu.

Our implementation of UTP in Isabelle is a natural extension of the work in Oliveira's PhD thesis [56], which is extended in [60], where UTP is embedded in ProofPowerZ, an extension of ProofPower/HOL supporting the Z notation, to mechanise the definition of Circus. Feliachi et al. [31] have developed a machine-checked, formal semantics based on ashaw embedding of Circus in Isabelle/Circus, a semantic theory of UTP also based on Isabelle/HOL. The definitions of Circus are based on those in [59], which are in turn based on those in [80]. Feliachi et al. derive proof rules from this semantics and implement tactic support for proofs of refinement for Circus processes involving both data and behavioral aspects. This proof environment supports a syntax for the semantic definitions that is close to textbook presentations of Circus.

Isabelle/UTP differs from the work of Feliachi et al. by being a deeper embedding of the UTP. In particular, we provide an explicit type for variables, where [31] does not, relying instead on HOL functions to represent them. Isabelle/UTP supports alphabet comparison and the specification of laws which rely on semantic properties like variable freshness. We also integrate our model of variables and expressions with the Isabelle type system, meaning we gain a significant level of proof automation. A detailed comparison of the approaches to semantic embeddings of the UTP is given in [35].

A complementary approach is given by Nipkow and Klein [53], based on denotational semantics, operational semantics, and Hoare logic. Many of the proof techniques there can be applied either directly or indirectly to Isabelle/UTP, for instance through our interpretation based proof tactics.

11. Conclusion

We have mechanised UTP, including alphabetised predicates, relations, the operators of imperative programming, and the theory of designs. All the proofs contained in this paper have been mechanised within our library, including those where the proofs have been omitted. Thus far, we have mechanised over 500 laws about the basic operators and over 60 laws about the theory of designs. The UTP library can therefore be used to perform basic static analysis of imperative programs. The proof automation level is high due to the inclusion of our proof tactics and the power of sledgehammer combined with the algebraic laws of the UTP.
Isabelle/UTP has been used as the basis for mechanising the semantics of CML [83], a formal modelling language for Systems of Systems, based in a combination of VDM and Circus. CML forms a vital element of the COMPASS project\(^4\), which supports formal engineering of Systems of Systems. So far we have mechanised much of the theory up to CSP, though work continues on the timed version of the semantics. The mechanised semantic model has also been combined with a CML expression model to provide theorem proving support for CML [28]. Our CML development environment, Symphony\(^5\), produces proof obligations which support the validity of a CML model, and we then use Isabelle-based tactics to discharge them. Moreover, Isabelle/UTP provides the basis to verify the soundness and completeness of these proof obligations with respect to the mechanised semantic model, though this remains a future work.

There are several additional directions for future work. We aim to mechanise additional theories, such as real-time and mobility, and provide links between them enabling a comprehensive approach to heterogeneous semantics. Our overall aim is to decomposed a language into its constituent theories, and then use these to provide links between languages and associated tools [33]. We also plan an online proof database for UTP theories and laws, which will enable a standard approach to proof for theory engineers. Moreover there are several portions of the UTP book which need to be mechanised, including the refinement calculus, associated laws, and in particular the total correctness law for recursion. In this vein we also plan to provide better automated support for program verification using the Hoare and weak precondition calculi. Finally, UTP also provides an account for operational semantics, and the future we wish to prove soundness of CML’s operational semantics through a link to the denotational semantics.

References


\(^4\)See www.compassresearch.eu for more information
\(^5\)Available at symphonytool.org/


[67] Thiago L. V. L. Santos, Ana Cavalcanti, and Augusto Sampaio. Object-orientation in the


