Article:
Foster, Simon David, Baxter, James, Cavalcanti, Ana Lucia Caneca et al. (2020) Unifying Semantic Foundations for Automated Verification Tools in Isabelle/UTP. Science of Computer Programming. 102510. ISSN: 0167-6423

https://doi.org/10.1016/j.scico.2020.102510

Unifying Semantic Foundations for Automated Verification Tools in Isabelle/UTP

Simon Foster*, James Baxter, Ana Cavalcanti, Jim Woodcock, Frank Zeyda

Department of Computer Science, University of York, Deramore Lane, Heslington, York YO10 5GH, United Kingdom

Abstract

The growing complexity and diversity of models used for engineering dependable systems implies that a variety of formal methods, across differing abstractions, paradigms, and presentations, must be integrated. Such an integration requires unified semantic foundations for the various notations, and co-ordination of a variety of automated verification tools. The contribution of this paper is Isabelle/UTP, an implementation of Hoare and He's Unifying Theories of Programming, a framework for unification of formal semantics. Isabelle/UTP permits the mechanisation of computational theories for diverse paradigms, and their use in constructing formalised semantics. These can be further applied in the development of verification tools, harnessing Isabelle's proof automation facilities. Several layers of mathematical foundations are developed, including lenses to model variables and state spaces as algebraic objects, alphabetised predicates and relations to model programs, algebraic and axiomatic semantics, proof tools for Hoare logic and refinement calculus, and UTP theories to encode computational paradigms.

Keywords: Theorem Proving, Lenses, Unifying Theories of Programming, Hoare Logic, Isabelle/HOL

1. Introduction

Unifying Theories of Programming [57] (UTP) is a framework for capturing, unifying, and integrating formal semantics using predicate and relational calculus. It aims to provide a "coherent structure for computer science" [57], by characterising the various programming languages it has produced, their foundational computational paradigms, and different semantic flavours. Along one axis, UTP allows us to study computational paradigms, such as functional, imperative, concurrent [57, 67], real-time [74], and hybrid dynamical systems [38, 37, 33]. In another axis, it allows us to characterise and link different presentations of semantics, such as axiomatic semantics [54, 61, 6, 71], with operational semantics and also algebraic semantics. UTP thus unites a diverse range of notations and paradigms, and so, with adequate tool support, it allows us to answer the substantial challenge of integrating formal methods [68, 44, 16].

The contribution of this paper is the theoretical and practical foundations of a tool for building UTP-based verification tools, called Isabelle/UTP, which draws from the strongest characteristics of previous implementations [66, 26, 79, 42, 80]. Our framework can unify the different paradigms and semantic models needed for modelling heterogeneous systems, and provides facilities for constructing verification tools. Isabelle/UTP is a shallow embedding [46, 78] of UTP into Isabelle/HOL [64], and so has access to Isabelle's proof capabilities. Isabelle is highly extensible, provides efficient automated proof [12], and has facilities needed to develop plugins for performing analysis on mathematical models of software and hardware.

Isabelle/UTP uses lenses [32, 31] to algebraically characterise mutation of a program state using two functions, similar to Back and von Wright's work [6, 4], and to support semantic reasoning facilities used in

*Corresponding author

Email addresses: simon.foster@york.ac.uk (Simon Foster), james.baxter@york.ac.uk (James Baxter), ana.cavalcanti@york.ac.uk (Ana Cavalcanti), jim.woodcock@york.ac.uk (Jim Woodcock)

program verification and refinement. Lenses were originally developed to support bidirectional programming languages for solving the "view-update problem" in database theory [10]. Here, we apply lenses to the modelling of state mutation and extend it with novel operators and laws.

Lenses allow us to generalise Back and von Wright's approach [6] since we can abstractly model "regions" of the observable state. These regions may correspond to individual variables, sets of variables, and also hierarchies. We develop several novel relations on lenses that allow us to relate these regions, including notions for independence, containment, and equivalence. We also provide operators for composing regions in a set-theoretic manner, which allows us to model alphabets of variables. Thus, lenses allow us to characterise syntax-related aspects of program verification, such as free variables, substitution, frames, and aliasing. We mechanise an algebraic structure for lenses in Isabelle/HOL, and show how it unifies a variety of state space models [73]. We draw comparisons with Back and von Wright's variable axioms [6, 4] and separation algebra [19]. Our account of lenses is a unifying algebra for observation and mutation of program state.

Upon our algebraic foundation of observation spaces, we construct UTP's relational program model. We develop a "deep" expression model [80], which is technically shallow, and yet has explicit syntax constructors supporting inductive proofs. We develop UTP predicates and binary relations [53], and provide a rich set of mechanically verified algebraic theorems, including the famous "laws of programming" [58], which form the basis for axiomatic semantics. Crucially, lenses allow us to express meta-logical provisos that involve syntax-related properties, such as whether two variables are different or whether an expression depends on a particular variable, without needing a deep embedding [42, 80].

We then develop several semantic presentations, including operational semantics, Hoare logic [54, 46, 57], and refinement calculus [61], with linking theorems showing how they are connected. From these, we also develop tactics for symbolic execution of relational programs [47], verification using Hoare logic [57], and an example verification of a find-and-replace algorithm. This illustrates the practicability and extensibility of our tool, which is not hampered by our deep expression model and other meta-logical machinery.

Finally, we demonstrate the mechanisation of UTP theories within the relational model, which allows us to support a hierarchy of advanced computational paradigms, including reactive programming [36], hybrid programming [33, 62], and object oriented programming [72]. Our work significantly advances previous contributions [79] with UTP theories whose observation spaces are typed by Isabelle, and which link to established libraries of algebraic theorems [9, 3] that facilitate efficient derivation of programming laws.

This paper is an extension of two conference papers [43, 35]. We extend and refine the material on lenses from [43], including a precise account of the algebra, a substantial body of novel theorems for each operator, a novel command for constructing lens-based state spaces, and additional motivating examples. We extend [35] with additional theorems that can be derived for UTP theories, a complete example based on timed relations with its mechanisation, and the use of frames in refinement calculus. On the whole, we simply present the core theorems without proofs, and therefore refer the interested reader to a number of companion reports [40, 41] and our repository[1]. All the definitions, theorems, and proofs in this paper are mechanically validated in Isabelle/HOL, and usually accompanied by an icon () linking to the corresponding resources in our repository. Though our work is primarily based on Isabelle/HOL, we prefer to use more traditional mathematical notations [75, 57] in this work[2], since we believe this makes the results more accessible.

In summary, our contributions are as follows: (1) mechanisation of lens theory in Isabelle/HOL, including fundamental algebraic theorems, and extension with novel relations and combinators; (2) facilities for automating construction of observation spaces; (3) an expression model, founded on lenses, providing both efficient proof and syntax-related queries, such as free variables and substitution; (4) a generic relational program model and proven laws of programming; (5) application to development of unified verification calculi, such as operational semantics, Hoare logic, and refinement calculus, including treatment of aliasing and frames; (6) characterisation of mechanised UTP theories to support various computational paradigms.

In §2 we motivate Isabelle/UTP, and review foundational work. In §3 to §6, we describe our contributions. The paper overview given in Figure 1 shows how the foundational parts of Isabelle/UTP are connected, and

[1] Isabelle/UTP Repository: https://github.com/isabelle-utp/utp-main

[2] The precedence of operators in this paper, from highest to lowest, is : †, :=, ⊛, ;, $\lhd \cdot \rhd$, =, ¬, ∧, ∨, ⇒, ⇔, ♯.

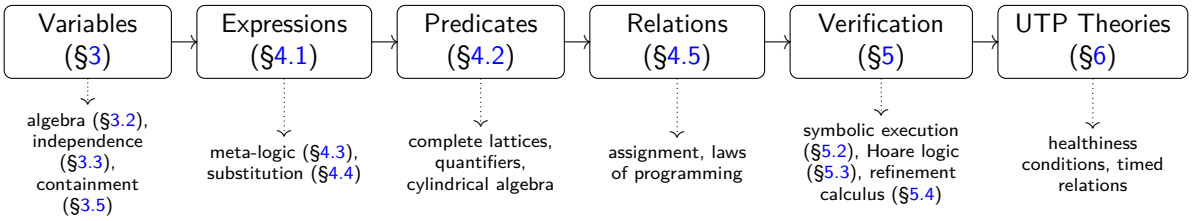

Figure 1: Roadmap of the Isabelle/UTP Foundations

the sections in which they are documented. In §3 we describe how state spaces and variables in a program can be modelled algebraically using lenses. In §4 we describe the core of Isabelle/UTP, first defining its expression model in §4.1, predicates in §4.2, meta-logical facilities in §4.3 and §4.4, and the relational program model in §4.5 with proof support and mechanised laws of programming. In §5 we use this relational program model to build tools for symbolic evaluation, verification using Hoare calculus with several examples in Isabelle/UTP, and also Morgan's refinement calculus [61]. In §6 we describe how different computational paradigms can be captured and mechanised using Isabelle/UTP theories, illustrating this using a UTP theory for concurrent and reactive programs. In §7 we survey related work, and in §8, we conclude.

2. Preliminaries and Motivation

In this section we motivate the contributions in our paper and briefly survey foundational work. We first introduce UTP (Section 2.1) and Isabelle/HOL (Section 2.2). In Section 2.3 we briefly survey work on semantic embedding to support verification. This leads to the conclusion that the state space modelling approach is the main consideration, and so in Section 2.4 we survey and critique different approaches. Finally in Section 2.5 we explain how Isabelle/UTP advances the state of the art.

2.1. Unifying Theories of Programming

Several authors consider integration of formal methods, notably Broy [16] and Paige [68]. These authors emphasise the centrality of unified formal semantics [52, 50]. Specifically, if diverse formal methods are to be coordinated, then their various semantic models must be reconciled to ensure consistent verification [68, 44]. An important development is Hoare and He's Unifying Theories of Programming (UTP) [56, 57, 21] technique, which fuses several intellectual streams, notably Hehner's predicative programming [49, 52, 55], relational calculus [76], and the refinement calculi of Back and von Wright [6], and Morgan [61].

The goal of UTP is to find the fundamental computational paradigms that underlie programming and modelling languages and characterise them with unifying denotational and algebraic semantics. A significant precursor to UTP is a seminal paper entitled *Laws of Programming* [58], in which nine prominent computer scientists, led by Hoare, give a complete set of algebraic laws for GCL [23]. The purpose of the laws of programming, however, is much deeper: it is to find laws that unify different programming languages.

UTP uses binary relations to model programs as predicates [49, 50]. Relations map an initial value of a variable, such as x, to its later value, x'. These might model program variables or observations of the real world. For example, $clock : \mathbb{N}$ records the passage of time. It makes no sense to assign values to $clock$ that reverse time. Healthiness conditions constrain observations using idempotent functions on predicates. For example, application of $\mathbf{HT}(P) \triangleq (P \wedge clock \leq clock')$ gives a predicate that forbids reverse time travel. If P is unhealthy, for example $clock' = clock - 1$, then application of $\mathbf{HT}$ changes its meaning: $\mathbf{HT}(clock = clock' + 1) = \mathbf{false}$.

The choice of observational variables and healthiness conditions defines a UTP theory. It characterises the set of relations whose elements are the fixed points of the healthiness conditions; $\mathbf{HT}(P) = P$ for our example. An algebra gives the relationship between theory elements, which supports the laws of programming for a particular paradigm [58]. UTP theories can be combined by composing their healthiness conditions, which allows multi-paradigm semantics [67].

Reactive processes [57, chapter 8] is a UTP theory for event-driven programs. It includes observational variables $wait : \mathbb{B}$ and $tr : \mathrm{seq}\, Event$, which represent, respectively, whether a program is quiescent (i.e. waiting for interaction), and the sequence of observed events (drawn from $Event$). An example healthiness condition is $\mathbf{R1}(P) \triangleq tr \leq tr'$, where $\leq$ is the prefix order on sequences. **R1** states that the interaction history must be retained – we may not undo past events.

A mechanisation of UTP in a theorem prover, like Isabelle [64, 26, 42], allows us to develop verification tools from UTP theories. Here, there are two goals that must be balanced: (1) the ability to engineer UTP theories and express the resulting algebraic laws; and (2) support for efficient automated verification [46, 78, 2]. In addition, the UTP is not about one programming language, or even one intermediate verification language (IVL) [11, 29], but unification of diverse modelling and program paradigms through algebraic laws. For example, our mechanisation should permit the algebraic law below [6, page 96].

Example 2.1 (Commutativity of Assignments)**.**

$$(x := e \,;\, y := f) = (y := f \,;\, x := e) \quad \text{provided } x \text{ and } y \text{ are independent}, x \text{ is not free in } f, y \text{ is not free in } e.$$

This law states that two assignments, $x := e$ and $y := f$, can commute provided that they are made to distinct variables, x and y, and they are not mentioned in f and e, respectively. Now, such a law is intuitive and holds in a variety of languages, which makes it a useful artefact for unification. Mechanising a law like this requires that variables are modelled as first-class citizens, so that they are objects of the logic, with which we can formulate the side conditions. In Section 2.3 we consider approaches to semantic embeddings, and motivate the approach we have chosen, but first we consider Isabelle/HOL.

2.2. Isabelle/HOL

Isabelle/UTP is a conservative extension of Isabelle/HOL [64], which is a proof assistant for Higher Order Logic (HOL). It consists of the Pure meta-logic, and the HOL object logic. Pure provides a term language, a polymorphic type system, a syntax-translation framework for extensible parsing and pretty printing, and an inference engine. The jEdit-based IDE allows LaTeX-like term rendering using Unicode.

Isabelle theories consist of type declarations, definitions, and theorems. We prove theorems in Isabelle using tactics. The simplifier tactic, simp, rewrites terms using equational theorems. The auto tactic combines simp with deductive reasoning. Isabelle also has the powerful sledgehammer proof tool [12] which invokes external first-order automated theorem provers on a proof goal, verifying their results using tactics like simp and metis, which is a first-order resolution prover.

HOL implements a functional programming language founded on an axiomatic set theory. This object logic gives us a principled approach to mechanised mathematics. We construct definitions and theorems by applying axioms in the proof kernel. HOL provides inductive datatypes, recursive functions, and records. It provides basic types, including sets ($\mathbb{P}\, A$), total functions ($A \to B$), numbers ($\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$), and lists. These types can be parametric: $[nat]list$.[3] Specialisation unifies two types if one is an instantiation of the type variables of the other. For example, $[nat]list$ specialises $[\alpha]list$, where α is a type parameter.

2.3. Verification and Semantic Embedding

In this section we consider different approaches to developing verification tools and outline the previous approaches to UTP mechanisation [65, 66, 26, 79, 42, 80] in this context.

Development of verification tools is usually conducted by means of a semantic embedding, where the language and deductive reasoning laws are embedded in a proof assistant such as Isabelle/HOL or Coq. Building on Gordon's work seminal work [46], Boulton et al. [14] identify the two fundamental categories of semantic embedding: deep embeddings and shallow embeddings. In a deep embedding, the syntax tree of the language is embedded into the host logic (such as HOL) as a datatype, and this acts as the basis for deductive verification calculi. In contrast, in a shallow embedding, the syntax tree is implicit, and the goal is to directly reuse host logic reasoning facilities.

[3] The square brackets are not used in Isabelle; we add them for readability.

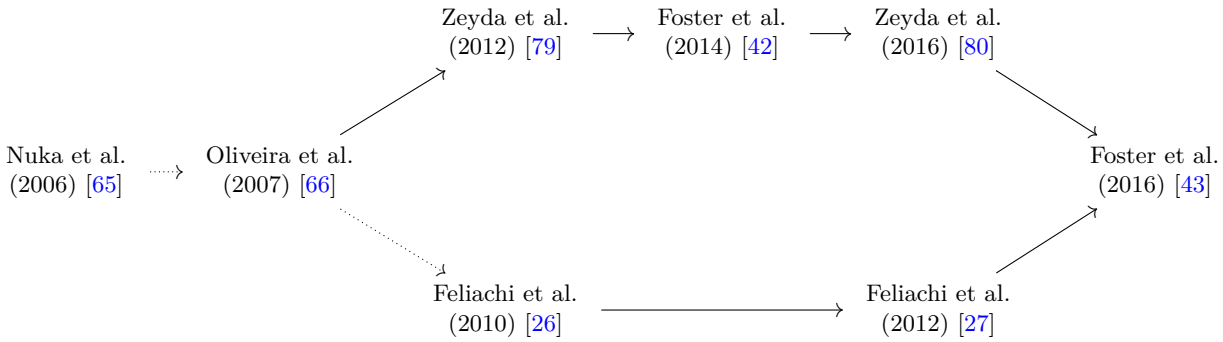

Figure 2: Overview of previous UTP semantic embeddings

Boulton et al. note that deep embeddings allow reasoning over the syntactic structure of programs; for example we can calculate the set of free variables in an expression. Consequently, a deep embedding can certainly support Example 2.1, since syntax is first-class. The deep embedding technique is used, for example, by Nipkow and Klein in their book *Concrete Semantics* [63]. Nevertheless, deep embeddings have the substantial disadvantage that they restrict the use of host logic proof facilities, which hampers efficient verification. While we can mechanise Example 2.1, it is not clear how we can efficiently apply it. Moreover, a particular requirement for a UTP mechanisation is that the syntax tree is extensible, so that additional programming operators can be defined, which is thwarted if we opt for a deep embedding.

In contrast to deep embeddings, shallow embeddings have been very successful in supporting program verification [46, 78, 2, 26, 3, 45]. As a prominent example, the seL4 microkernel verification project uses a shallow embedding called Simpl [2], which illustrates its scalability. Moreover, most of the previous UTP mechanisations are also shallow embeddings [66, 26, 79, 42]. The exceptions are Nuka and Woodcock [65], who follow the deep embedding approach, and Butterfield [17, 18], who develops a bespoke proof tool called $U{\cdot}(TP)^2$. Within the shallow embeddings, broadly there are two streams, begun by Oliveira et al. [66, 79, 42] and Feliachi et al. [26, 27, 28], as illustrated in Figure 2. Both streams have HOL as the host logic, though Oliveira et al. [66] use a dialect called ProofPower-Z[4], whereas Feliachi et al. [26] use Isabelle/HOL. We will consider further differences in Section 2.4, but note that Isabelle/UTP [43] results from the confluence of the ideas from the streams [26, 42].

Isabelle/UTP is developed as a shallow embedding. All shallow embeddings follow roughly the same basic approach [46, 5]. First, we fix a state space $\mathcal{S}$, to describe states of a program. Afterwards, we can model predicates using the type $\mathcal{S} \to \mathbb{B}$, and programs using $\mathcal{S} \times \mathcal{S} \to \mathbb{B}$, which are binary relations. From this foundation, the programming language operators can be defined using the predicative programming approach [49], where programs are represented as below.

Definition 2.2 (Programs as Predicates)**.**

$$
\begin{aligned}
P \,;\, Q &\triangleq (\lambda(s, s') \bullet (\exists\, s_0 \bullet P(s, s_0) \wedge Q(s_0, s'))) \\
x := e &\triangleq (\lambda(s, s') \bullet s'.x = e(s) \wedge s'.y_1 = s.y_1 \wedge \cdots \wedge s'.y_n = s.y_n) \\
\textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ P\ \textbf{\textit{else}}\ Q\ \textbf{\textit{fi}} &\triangleq (\lambda(s, s') \bullet (b(s) \wedge P(s, s')) \vee (\neg b(s) \wedge Q(s, s')))
\end{aligned}
$$

These predicates effectively describe whether a given input-output pair, $(s, s') : \mathcal{S} \times \mathcal{S}$, is an observation of the program. Sequential composition $P \,;\, Q$ is a predicative version of relational composition [76]. It requires that there exists a middle state s_0 such that P and Q have this as a possible final state and initial state, respectively. Assignment $x := e$ states that x in the final state s' has the value e, and every other variable (y_i) retains its value. If-then-else conditional admits the behaviours of P when b is true, and the

[4]ProofPower: http://www.lemma-one.com/ProofPower/index/

behaviours of Q when b is false. We can also denote the partial correctness Hoare triple operator [46]:

$$\{p\}\ Q\ \{r\} \triangleq (\forall (s, s') \bullet p(s) \wedge Q(s, s') \Rightarrow r(s'))$$

A Hoare triple is valid if, for any (s, s') where the precondition p is satisfied by s, and Q has a final state s' when started from s, the postcondition r is satisfied by s'. From this definition, we can prove many of the Hoare logic axioms as theorems [54, 46]. Other axiomatic verification calculi [23, 61, 6, 56] can be characterised in a similar way – this is the standard shallow embedding approach.

However, not all laws are straightforward to express. In shallow embeddings, variables are often not first-class citizens [78, 2, 26]. That being the case, they are not objects of the logic, and so it is not possible to express Example 2.1. Moreover, consider the following variant of the forward assignment law:

$$\{p\}\ x := e\ \{x = e \wedge p\} \quad \text{provided } x \text{ is not free in } p \text{ and } e$$

This is certainly a useful law, as it allows us, for example, to push initial assignments with constants forward, such as $x := 1$. However, it requires that we can determine whether the variable x is free in p. This is seemingly a property that can only be expressed if e and p are syntactic objects. Example laws of this kind exist in many axiomatic calculi [23, 61, 6]. Now, to be clear, the absence of this law does not prevent a particular program from being verified, and so it may not seem important. However, if the goal is unification by characterising such laws abstractly, as is the case in UTP, then this is a significant question. Adequately answering this is key to ensure that Isabelle/UTP is truly a unifying framework. In Definition 2.2, we use the notation $s.x$ to represent the value of variable x in state s. How this operator is represented depends on how we model state spaces, the crucial question for shallow embeddings, which we consider next.

2.4. State Space Modelling

The principal difference between the shallow embeddings in Section 2.3 is their approach to modelling the observation space $\mathcal{S}$. Schirmer and Wenzel provide a helpful discussion on modelling state spaces [73], and so we employ their framework for comparison and critique. They identify four common ways of mechanising the modelling of state: using (1) functions; (2) tuples, (3) records, and (4) abstract types.

The first approach models state as a function, $\mathcal{S} \triangleq \mathit{Var} \to \mathit{Value}$, for suitable value and variable types, and so $s.x \triangleq s(x)$. Gordon [46], Back and von Wright [5], Oliveira et al. [66], and the successor UTP mechanisations [79, 42, 80], follow this approach. It requires a deep model of variables and values. Consequently, it has similarities with deep embeddings, since concepts such as names and typing are first-class citizens. This provides an expressive model with few limitations on possible manipulations of variables in the state space [42]. However, Schirmer and Wenzel highlight two obstacles [73]. First, the machinery required for deep reasoning about values is heavy and *a priori* limits possible value constructions, due to cardinality restrictions in HOL. Second, explicit variable naming means the embedding must tackle syntactic issues, like α-renaming.

Zeyda et al. [80] mitigate the first issue by axiomatically introducing a value universe (*Value*) in Isabelle. This universe has a higher cardinality that any HOL type, and so all normal types can be injected into it. The cost of this approach is the extension of HOL with additional axioms. Moreover, the complexities associated with the second issue remain. Once names and types are first-class citizens, it becomes necessary to replicate a large part of the underlying meta-logic, such as a type checker. This requires great effort and can hit proof efficiency. Even so, the functional state space approach seems necessary to model the dynamic creation of variables, as required, for example, in modelling memory heaps in separation logic [19, 24], so we do not reject it entirely.

The second approach [73] uses tuples to represent state; for example $\mathbb{Z} \times \mathbb{B} \times \mathbb{Z}$ can represent a state space with three variables [78]. The value of each variable can be obtained by decomposing the state using pattern matching, or using projection functions so that $s.x \triangleq \pi_n(s)$. The main issue with this approach is that variable names are not automatically represented by the state space.

The third approach uses records to model state: a technique often used by verification tools in Isabelle [2, 26, 27, 3]. It is similar to the second approach, since records are simply tuples. However, records

come with bespoke selection and update functions for each index; this makes manipulating the state space straightforward. Thus, we have $s.x \triangleq x(s)$, with x being a field selector function. Feliachi et al. use this approach to create their semantic embedding of the UTP in Isabelle/HOL [26]. A record field represents a variable in this model. These can be abstractly represented using pairs of field-query and update functions, f_i and f_i-*upd*. We do not need to encode the set of variable names (Var) in this approach.

The record state space approach greatly simplifies automation of program verification [26, 27, 28, 3, 45]. This is through directly harnessing, rather than replicating, the polymorphic type system and automated proof tactics. The expense, though, is a loss of flexibility compared to the functional approach, particularly in the decomposition of state spaces. Moreover, a field of a record is not a first-class citizen because it is not an object of the logic in Isabelle. This means that record fields lack the semantic structure necessary to capture their behaviour and thus manipulate or compare them – f_i and f_j are simply different functions. Consequently, implementations using records seldom provide general support for syntactic concepts like free variables and substitution.

The previous approaches all use concrete models (types) for $\mathcal{S}$. The fourth approach [73] uses an abstract type to represent a state space, with axiomatised projection functions for each of the variables. In this model we have again that $s.x \triangleq x(s)$, but $x : \mathcal{S} \to \mathcal{V}$ is simply an abstract function without an explicit implementation. This approach is employed by Back and von Wright [6], who use two functions $\mathsf{val}.x : \mathcal{S} \to \mathcal{V}$ and $\mathsf{set}.x : \mathcal{V} \to \mathcal{S} \to \mathcal{S}$, to characterise each variable, together with five axioms that characterise their behaviour (see Definition 7.1). However, as indicated by both Schirmer and Wenzel [73], and Back and Preoteasa [4], this approach requires us to *a priori* fix the number of variables available and their axioms. This hampers modularity in mechanisation, since it is difficult to add new variables to grow a state space.

Schirmer and Wenzel's solution [73] is to adopt the state as functions approach, but improve its flexibility using locales [8]. An Isabelle locale allows the creation of a local theory context, with fixed polymorphic constants and axiomatic laws [9]. Rather than fixing concrete types for *Var* and *Val*, Schirmer and Wenzel characterise these abstractly, and use locale constants to characterise injection and projection functions. This is very similar to the approach adopted in our earlier version of Isabelle/UTP [42], except that the latter uses type classes to assign injections to a polymorphic value universe.

This approach imposes limitations that make it unsuitable for UTP, because it limits polymorphism in variable types[5]. For example, in the UTP theory of reactive processes, a trace variable tr can be given the polymorphic type $[e]list$ for some event type e. When we hide events in a process, the event type can change, since some events are no longer visible. Yet, in a locale, constants have a fixed type and so tr is not truly polymorphic. In comparison, if we define a record with a field $tr : [e]list$ then we can assign it different types. We conclude that there is a need for a different approach to state space modelling.

2.5. Our Approach

Our approach is closest to the abstract type approach, though we aim is to unify all four. For UTP, we need to treat variables as first-class citizens. We see merits both in modelling state as a function, and also as a record. Indeed, we recognise that there is often a need to blend these two representations, as we illustrate using the running example below:

Example 2.3 (Stores Variables and Heaps)**.** Consider a state space with three variables $x : int$, $y : int$, and $hp : addr \to int$. The variables x and y represent program variables in the store, and hp represents a heap mapping addresses to integer values. It can be modelled as a record with three fields. However, we will likely want to perform assignments directly to elements of function hp, for example $hp[loc] := 7$, and so we see the two state representations, functions and records, co-existing. □

Isabelle/UTP generalises the various state-space approaches by abstractly characterising variables algebraically using lenses [32, 31, 59, 69]. A lens consists of two functions: $\mathit{get} : \mathcal{S} \to \mathcal{V}$ that extracts a value from a state, and $\mathit{put} : \mathcal{S} \to \mathcal{V} \to \mathcal{S}$ that puts back an updated view. We characterise lenses as algebraic

[5]This fact was first pointed out to us by Prof. Burkhart Wolff. Constants fixed in the head of an Isabelle locale have fixed types and cannot be polymorphic. In contrast, constants introduced at the global theory level can be fully polymorphic.

structures to which different concrete models can be assigned, which allows us to unify the various state space approaches. Lenses allow characterisation of both individual variables and also state space *regions* that can encompass several variables. We consider, for example, that the heap location $hp[loc]$ is semantically a part of the heap hp, and therefore changes to hp also effect $hp[loc]$. Moreover, in an object oriented program the state is hierarchical: the attribute variables are all part of the object variable.

Since lenses can characterise sets of variables, we can also use them to model frames, as required, for example, by Morgan's refinement calculus [61]. We define operators that allow us to compare and manipulate lenses, including independence ($x \bowtie y$), sublens ($x \preceq y$), and summation ($x + y$), which effectively allows execution of two lenses in parallel, but differently to Pickering's operator [69], which acts on a product space. We also implement N. Foster's lens composition operator ($x \fatsemi y$) [31], which supports hierarchical state spaces.

Isabelle/UTP, therefore, has a high level of proof automation because it is a shallow embedding [78, 26, 27], and we avoid the requirement to explicitly characterise names and values. Like any other object in Isabelle, we can assign a name to a particular lens, but this name is meta-logical. Such globally named lenses can also be polymorphic, which helps us to model observational variables. At the same time, even though Isabelle/UTP is a shallow embedding, the lens axioms provide us with sufficient structure to characterise syntax-like queries, like substitution and free variables. Consequently, lenses allow us to develop a program model that exhibits benefits of both deep and shallow embeddings.

3. Algebraic Observation Spaces

In this section, we present our theory of lenses, which provides an algebraic semantics for state and observation space modelling in Isabelle/UTP. Although some core definitions like the lens laws and composition operator are well known [32, 31, 30], we introduce several novel operators, including summation, and relations like independence and equivalence. We prove several algebraic properties for these operators, which are foundational for our mechanisation of UTP.

All definitions, theorems, and proofs in this section may be found in our Isabelle/HOL mechanisation [40].

3.1. Signature

A lens is used, intuitively, to view and manipulate a region ($\mathcal{V}$) of a state space ($\mathcal{S}$), as illustrated in Figure 3. The view, $\mathcal{V}$, corresponds to the hatched region of $\mathcal{S}$. A region may model the contents of one or more variables, whose type is $\mathcal{V}$. We introduce lenses as two-sorted algebraic structures.

Definition 3.1 (Lenses)**.** A lens is a quadruple $\langle \mathcal{V} \mid \mathcal{S} \mid \mathit{get} : \mathcal{S} \to \mathcal{V} \mid \mathit{put} : \mathcal{S} \to \mathcal{V} \to \mathcal{S} \rangle$, where $\mathcal{V}$ and $\mathcal{S}$ are non-empty sets called the view type and state space[6], respectively, and get and put are total functions[7]. The view $\mathcal{V}$ corresponds to a region of the state space $\mathcal{S}$. We write $\mathcal{V} \Longrightarrow \mathcal{S}$ to denote the type of lenses with state space $\mathcal{S}$ and view type $\mathcal{V}$, and subscript get and put with the name of a particular lens.

This follows the standard definition given by N. Foster [31], though other works [32] employ partial functions. The get function views the current value of the region, and put updates it. Intuitively, we use these structures to model sequences of queries and updates on the state space $\mathcal{S}$ in §4. Each variable in a program can be represented by an individual lens, with operators like assignment utilising get and put to manipulate the corresponding region of memory. For the purpose of example, we describe lenses for record types.

Definition 3.2 (Record Field Lens)**.** We consider the definition of a new record type, $R \triangleq (\!| f_1 : \tau_1, \cdots f_n : \tau_n |\!)$, with n fields, each having a corresponding type. Each field yields a function $f_i : R \to \tau_i$, which queries the current value of a field. Moreover, we can update the value of field f_i in $r : R$ with $k : \tau_i$ using $r(f_i := k)$. We can construct a lens for each field using $\mathit{rec}_{f_i}^{R} \triangleq \langle \tau_i \mid R \mid f_i \mid \lambda\, s\, v \bullet s(f_i := v) \rangle$.

[6] In the lens literature [31, 30, 69] this is referred to as the "source". We refer to it as the state space and observation space, a more general concept, interchangeably, depending on the context.

[7] N. Foster [31] also introduces a function called $\mathit{create} : \mathcal{V} \to \mathcal{S}$ that creates an element of the source. We omit this because it can be defined in terms of put and is not necessary for this paper.

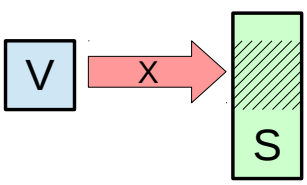

Figure 3: Lens viewing a region

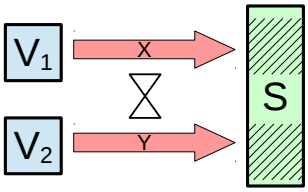

Figure 4: Independence

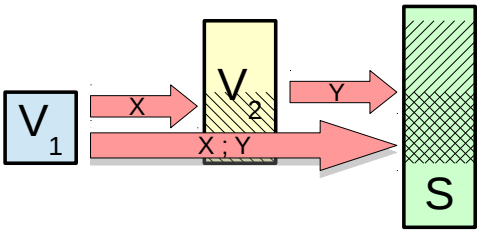

Figure 5: Lens composition

The field lens allows us to employ the "state as records" approach [73, 26], as discussed in Section 2.4. We also consider a second example. Many state spaces are built using the Cartesian product type $S_1 \times S_2$, and consequently it is useful to define lenses for such a space. We therefore define the ***fst*** and ***snd*** lenses.

Definition 3.3 (Product Projection Lenses)**.**

$$\begin{aligned} \mathit{fst}^{S_1,S_2} &\triangleq \langle S_1 \mid S_1 \times S_2 \mid \lambda(x,y) \bullet x \mid \lambda(x,y)\, z \bullet (z,y) \rangle \\ \mathit{snd}^{S_1,S_2} &\triangleq \langle S_2 \mid S_1 \times S_2 \mid \lambda(x,y) \bullet y \mid \lambda(x,y)\, z \bullet (x,z) \rangle \end{aligned}$$

The superscripted state spaces are necessary in order to specify the product type; we omit them when they can be determined from the context. Lenses ***fst*** and ***snd*** allow us to focus on the first and second element of a product type, respectively. Their *get* functions project out these elements, and the *put* functions replace the first and second elements with the given value z. An application of these lenses is the "states as products" approach [73] (see Section 2.4): we can directly model a state space with two variables, for example $x \triangleq \mathit{fst}^{\mathbb{Z},\mathbb{B}}$ and $y \triangleq \mathit{snd}^{\mathbb{Z},\mathbb{B}}$ give a state space with $x : \mathbb{Z}$ and $y : \mathbb{B}$. A final example is the total function lens.

Definition 3.4. $\mathit{fun}_k^{A,B} \triangleq \langle B \mid A \to B \mid \lambda f \bullet f(k) \mid \lambda f\, v \bullet (\lambda x \bullet \mathsf{if}\ x = k\ \mathsf{then}\ v\ \mathsf{else}\ f(x)) \rangle$

The total function lens $\mathit{fun}_k^{A,B}$ views the output of a function $f : A \to B$ associated with a given input value $k : A$. The *get* function simply applies f to k, and the *put* function associates a new output v with k. The function lens allows us to also employ the "state as functions" approach [66, 73]. We can also use it to model an array of integer values with $\mathit{fun}_k^{\mathbb{N},\mathbb{Z}}$, as used in Example 2.3.

3.2. Axiomatic Basis

The use of lenses to model variables depends on *get* and *put* behaving according to a set of axioms.

Definition 3.5 (Total Lenses)**.** A total lens obeys the following axioms:

$$\mathit{get}\,(\mathit{put}\,s\,v) = v \tag{PutGet}$$
$$\mathit{put}\,(\mathit{put}\,s\,v')\,v = \mathit{put}\,s\,v \tag{PutPut}$$
$$\mathit{put}\,s\,(\mathit{get}\,s) = s \tag{GetPut}$$

We write $\mathcal{V} \Longmapsto \mathcal{S}$ for the set of total lenses with view type $\mathcal{V}$ and state space $\mathcal{S}$, and $* \Longmapsto \mathcal{S}$ for the set of total lenses with any view type, whose state space is $\mathcal{S}$.

We mechanise this algebraic structure in Isabelle using locales, following the pattern given by Ballarin [8][8]. Total lenses are usually called "very well-behaved" lenses [32, 31], but we believe "total" is more descriptive, since it is always possible to meaningfully project a view from a state. Axiom PutGet states that if a state has been constructed by application of $\mathit{put}\,s\,v$, then a matching *get* returns the injected value, v. Axiom PutPut states that a later *put* overrides an earlier one, so that the previously injected value v' is replaced by v. Finally, Axiom GetPut states that for any state element s, if we extract the view element and then update the original using it, then we get precisely s back.

We now demonstrate that every field of a record yields a total lens.

[8] We are not using locales to characterise state spaces, like Schirmer and Wenzel [73], but simply to fix the algebra of lenses.

Lemma 3.6. *For any field f_i of a record type R, the record lens $\mathbf{rec}^R_{f_i}$ forms a total lens.*

Proof. For illustration, we prove each lens axiom in turn.

1. PutGet: $\mathit{get}\,(\mathit{put}\,s\,v) = f_i\,(s(f_i := v)) = v$
2. PutPut: $\mathit{put}\,(\mathit{put}\,s\,v')\,v = (\lambda\, s\, v \bullet s(f_i := v))\,(s(f_i := v'))\,v = (\lambda\, v \bullet s(f_i := v')(f_i := v))\,v = \mathit{put}\,s\,v$
3. GetPut: $\mathit{put}\,s\,(\mathit{get}\,s) = (\lambda\, s\, v \bullet s(f_i := v))\,s\,(f_i\,s) = s(f_i := f_i\,s) = s$

Consequently, the record lens is indeed a total lens. □

We can similarly show that ***fst***, ***snd***, and ***fun*** are total lenses[9].

Lemma 3.7. *For any A,B, and $k \in A$, $\mathbf{fst}^{A,B}$, $\mathbf{snd}^{A,B}$, and $\mathbf{fun}^{A,B}_k$ are total lenses.*

While both PutGet and PutPut are satisfied for most useful state-space models we can consider, this is not the case for GetPut. For example, if we consider a lens that projects the valuation of an element $x : A$ from a partial function $f : A \nrightarrow B$, then *get* is only meaningful when $x \in \mathrm{dom}(f)$. Since *get* is total, it must return a value, but this will be arbitrary and therefore placing it back into f alters its domain. We do not consider lenses that do not satisfy GetPut in this paper, but simply observe that total lenses are a useful, though not universal solution for state space modelling.

3.3. Independence

So far we have considered the behaviour of individual lenses, but programs reference several variables and so it is necessary to compare them. One of the most important relationships between lenses is independence of their corresponding views, which is illustrated in Figure 4. We formally characterise independence below.

Definition 3.8 (Independent Lenses)**.** Lenses $X : \mathcal{V}_1 \Longrightarrow \mathcal{S}$ and $Y : \mathcal{V}_2 \Longrightarrow \mathcal{S}$ are independent, written $X \bowtie Y$, provided they satisfy the following laws:

$$\mathit{put}_X\,(\mathit{put}_Y\,s\,v)\,u = \mathit{put}_Y\,(\mathit{put}_X\,s\,u)\,v \tag{LI1}$$

$$\mathit{get}_X\,(\mathit{put}_Y\,s\,v) = \mathit{get}_X\,s \tag{LI2}$$

$$\mathit{get}_Y\,(\mathit{put}_X\,s\,u) = \mathit{get}_Y\,s \tag{LI3}$$

Lenses X and Y, which share the same state space but not necessarily the same view type, are independent provided that applications of their respective *put* functions commute (LI1), and their respective *get* functions are not influenced by the corresponding *put* functions (LI2, LI3). In the encoding of Example 2.3, we have that $x \bowtie y$, $x \bowtie hp$, and $y \bowtie hp$: these are distinct variables. Nevertheless, lens independence captures a deeper concept, since lenses with different types are not guaranteed to be independent, as they might represent a different view on the same region.

Independence can, for example, be used capture the condition for commutativity of assignments. If x and y are variables, then we can characterise the following law

$$(x := e\,;\, y := f) = (y := f\,;\, x := e) \quad \text{provided } x \bowtie y, \text{ and } e \text{ and } f \text{ are constants}$$

which partly formalises Example 2.1. Such assignment laws will be explored further in Section 4. We can show that $\mathbf{fst} \bowtie \mathbf{snd}$ using the calculation below.

Lemma 3.9. $\mathbf{fst} \bowtie \mathbf{snd}$

Proof. We first prove that $\mathit{put}_{\mathbf{fst}}$ commutes with $\mathit{put}_{\mathbf{snd}}$ (LI1) by evaluation, assuming $s = (s_1, s_2)$:

$$\begin{aligned} & \mathit{put}_{\mathbf{fst}}(\mathit{put}_{\mathbf{snd}}\,s\,v)\,u = \mathit{put}_{\mathbf{fst}}(\mathit{put}_{\mathbf{snd}}\,(s_1, s_2)\,v)\,u \\ = {} & \mathit{put}_{\mathbf{fst}}\,(s_1, v)\,u = (u, v) = \mathit{put}_{\mathbf{snd}}\,(u, s_2)\,v \\ = {} & \mathit{put}_{\mathbf{snd}}\,(\mathit{put}_{\mathbf{fst}}\,(s_1, s_2)\,u)\,v = \mathit{put}_{\mathbf{snd}}\,(\mathit{put}_{\mathbf{fst}}\,s\,u)\,v \end{aligned}$$

[9] All omitted proofs can be found in our Isabelle/HOL mechanisation [40].

Similarly, we can also prove LI2 by evaluation:

$$\mathit{get}_{\mathit{fst}}\,(\mathit{put}_{\mathit{snd}}\,s\,v) \;=\; \mathit{get}_{\mathit{fst}}\,(\mathit{put}_{\mathit{snd}}\,(s_1, s_2)\,v) \;=\; \mathit{get}_{\mathit{fst}}\,(s_1, v) \;=\; s_1 \;=\; \mathit{get}_{\mathit{fst}}\,(s_1, s_2) \;=\; \mathit{get}_{\mathit{fst}}\,s$$

Finally, LI3 follows by a symmetric proof. □

A further, illustrative result is the meaning of independence in the total function lens:

Lemma 3.10. $\mathit{fun}_a \bowtie \mathit{fun}_b \Leftrightarrow a \neq b$

Two instances of the total function lens are independent if, and only if, the parametrised inputs a and b are different. This reflects the intuition of a function – every input is associated with a distinct output.

We can actually show that axioms LI2 and LI3 in Definition 3.8 can be dispensed with when X and Y are both total lenses. Consequently, we can adopt a simpler definition of independence:

Theorem 3.11. *If lenses* $X : \mathcal{V}_1 \Longrightarrow \mathcal{S}$ *and* $Y : \mathcal{V}_2 \Longrightarrow \mathcal{S}$ *are both total then*

$$X \bowtie Y \Leftrightarrow \forall (u, v, s) \bullet \mathit{put}_X\,(\mathit{put}_Y\,s\,v)\,u \;=\; \mathit{put}_Y\,(\mathit{put}_X\,s\,u)\,v$$

However, the weaker definition of independence is still useful for the situation when not all three axioms of total lenses are satisfied [34].

3.4. Lens Combinators

Lenses can be independent, but they can also be ordered by containment using the sublens relation $X \preceq Y$, which orders lenses. The intuition is that Y captures a larger region of the state space than X. One interpretation is a subset operator for relating lens sets. Before we can get to the definition of this $\preceq$, we first need to define some basic lens combinators.

Definition 3.12 (Basic Lenses)**.** $\mathbf{0}_S \triangleq \langle \{\emptyset\} \mid S \mid \lambda\, s \bullet \emptyset \mid \lambda\, s\, v \bullet s \rangle \qquad \mathbf{1}_S \triangleq \langle S \mid S \mid \lambda\, s \bullet s \mid \lambda\, s\, v \bullet v \rangle$

Lemma 3.13 (Basic Lenses Closure)**.** *For any* S, $\mathbf{0}_S$ *and* $\mathbf{1}_S$ *are total lenses.*

The **0** lens has a unitary view type, $\{\emptyset\}$. Consequently, for any element of the state space, it always views the same value $\emptyset$. It cannot be used to either observe or change a state, and it is therefore entirely ineffectual in nature. It can be interpreted as the empty set of lenses. Conversely, the **1** lens, with type $S \Longrightarrow S$, views the entirety of the state. It can be interpreted as the set of all lenses in the state space. We sometimes omit type information from these basic lenses when this can be inferred from the context.

It is useful in several circumstances to chain lenses together, provided that the view of one matches the source of the other. For this, we adopt the lens composition operator, originally defined by J. Foster [31].

Definition 3.14. $X \fatsemi Y \;\triangleq\; \langle \mathcal{V}_X \mid \mathcal{S}_Y \mid \mathit{get}_X \circ \mathit{get}_Y \mid \lambda\, s\, v \bullet \mathit{put}_Y\, s\,(\mathit{put}_X\,(\mathit{get}_Y\, s)\, v) \rangle$ if $\mathcal{S}_X = \mathcal{V}_Y$

Lemma 3.15 (Composition Closure)**.** *If* X *and* Y *are total lenses, then* $X \fatsemi Y$ *is a total lens.*

A lens composition, $X \fatsemi Y$, for $X : A \Longrightarrow B$ and $Y : B \Longrightarrow C$ chains together two lenses. It is illustrated in Figure 5. When X characterises an A-shaped region of B, and Y characterises a B-shaped region of C, overall lens $X \fatsemi Y$ characterises an A-shaped region of C. It is useful when we have a state space composed of several individual state components, and we wish to select a variable of an individual component.

Example 3.16. In an object oriented program we may have $m \in \mathbb{N}$ objects whose states are characterised by lenses $o_i : O_i \Longrightarrow \mathcal{S}$, for $i \in \{1..m\}$, where each O_i characterises the respective object state. An object o_k has $n \in \mathbb{N}$ attributes, characterised by lenses $x_j : \tau_j \Longrightarrow O_k$ for $j \in \{1..n\}$. We can select one of these attributes from the global state context by the composition $x_j \fatsemi o_k : \tau_j \Longrightarrow \mathcal{S}$. □

Composition is also useful for collections, such as the heap array in Example 2.3. If $hp : (addr \to \mathbb{Z}) \Longrightarrow \mathcal{S}$, that is, a lens which views a function in $\mathcal{S}$, then we can represent a lookup, $hp[loc]$, by the composition $\mathit{fun}_{loc}^{addr,\mathbb{Z}} \fatsemi hp$[10].

Lens composition obeys a number of useful algebraic properties, as shown below.

Theorem 3.17 (Composition Laws). *If $X : A \Longrightarrow B$, $Y : B \Longrightarrow C$, and $Z : C \Longrightarrow D$ are total lenses then the following identities hold:*

$$X \fatsemi (Y \fatsemi Z) \;=\; (X \fatsemi Y) \fatsemi Z \qquad X \fatsemi \mathbf{1}_B \;=\; \mathbf{1}_A \fatsemi X = X \qquad \mathbf{0}_A \fatsemi X \;=\; \mathbf{0}_B$$

Lens composition is associative, since the order in which lenses are composed is irrelevant, and it has $\mathbf{1}$ as its left and right units. Moreover, $\mathbf{0}$ is a left annihilator, since if the view is reduced to $\{\emptyset\}$ then no further data can be extracted.

While composition can be used to chain lenses in sequence, it is also possible to compose them in parallel, which is the purpose of the next operator.

Definition 3.18 (Lens Sum).

$$X + Y \;\triangleq\; \langle \mathcal{V}_X \times \mathcal{V}_Y \mid \mathcal{S}_X \mid \lambda\, s \bullet (\mathit{get}_X\, s, \mathit{get}_Y\, s) \mid \lambda\, s\, (v_1, v_2) \bullet \mathit{put}_Y\, (\mathit{put}_X\, s\, v_1)\, v_2 \rangle \qquad \text{if } \mathcal{S}_X = \mathcal{S}_Y$$

Lens sum allows us to simultaneously manipulate two regions of $\mathcal{S}$, characterised by $X : \mathcal{V}_1 \Longrightarrow \mathcal{S}$ and $Y : \mathcal{V}_2 \Longrightarrow \mathcal{S}$. Consequently, the view type is the product of the two constituent views: $\mathcal{V}_1 \times \mathcal{V}_2$. The *get* function applies both constituent *get* functions in parallel. The *put* function applies the constituent *put* functions, but in sequence since we are in the function domain. We can prove that total independent lenses are closed under lens sum.

Lemma 3.19 (Sum Closure). *If X and Y are independent total lenses, then $X + Y$ is a total lens.*

We require that $X \bowtie Y$, since manipulation of two overlapping regions could have unexpected results, and then also the order of the *put* functions is irrelevant.

Lens sum can characterise independent concurrent views and updates to the state space. For example, we can encode a simultaneous update to two variables as $(x + y) := (e, f)$. Moreover, lens sum can also be used to characterise sets of independent lenses. If we model three variables using lenses x, y, and z which share the same source and are all independent, then the set $\{x, y, z\}$ can be represented by $x + y + z$.

With the help of lens composition, we can now also prove some algebraic laws for lens sum.

Lemma 3.20. *If X and Y are independent total lenses then the following identities hold:*

$$\mathit{fst} \fatsemi (X + Y) \;=\; X \quad \mathit{snd} \fatsemi (X + Y) \;=\; Y \quad (X + Y) \fatsemi Z \;=\; (X \fatsemi Z) + (Y \fatsemi Z)$$

The first two identities show that *fst* and *snd* composed with $+$ yield the left- and right-hand side lenses, respectively. If we perform a simultaneous update using $X + Y$, but then throw away one them by composing with *fst* and *snd*, then the result is simply X or Y, respectively. The third identity shows that $\fatsemi$ distributes from the right through lens sum. It does not in general distribute from the left as such a construction is not well-formed. We next show some independence properties for the operators introduced so far.

Lemma 3.21 (Independence). *If X, Y, and Z are total lenses then the following laws hold:*

$$\begin{aligned} \mathbf{0} &\bowtie X \\ X \bowtie Y &\Leftrightarrow Y \bowtie X \\ Y \bowtie Z &\Rightarrow (X \fatsemi Y) \bowtie Z \end{aligned} \qquad \begin{aligned} (X \fatsemi Z) &\bowtie (Y \fatsemi Z) \Leftrightarrow X \bowtie Y \\ X \bowtie Z \wedge Y &\bowtie Z \Rightarrow (X + Y) \bowtie Z \end{aligned}$$

[10] Here, *loc* is a constant value, though this can be relaxed to an expression that can depend on other state variables [34].

The **0** lens is independent from any lens, since it views none of the state space. Lens independence is a symmetric relation, as expected. Lens composition preserves independence: if $Y \bowtie Z$ then composing X with Y still yields a lens independent of Z. Referring back to Example 2.3, if $x \bowtie hp$, then clearly also $x \bowtie hp[loc]$. X and Y composed with a common lens Z are independent if, and only if, X and Y are themselves independent. Combining this with Lemma 3.10, we can deduce that $hp[l_1]$ and $hp[l_2]$ are independent if and only $l_1 \neq l_2$. Finally, lens sum also preserves independence: if both X and Y are independent of Z, then also $X + Y$ is independent of Z. Thus, if $x \bowtie hp$ and $y \bowtie hp$, then also $x + y \bowtie hp$.

3.5. Observational Order and Equivalence

We recall that a lens X can view a larger region than another lens Y, with the implication that Y is fully dependent on X. For example, it is clear that in Example 3.16 each object fully possesses each of its attributes, and likewise each attribute lens $x_j \fatsemi o_k$ is fully dependent on lens o_k. We can formalise this using the lens order, $X \preceq Y$, which we can now finally define.

Definition 3.22 (Lens Order)**.** $(X \preceq Y) \triangleq (\mathcal{S}_X = \mathcal{S}_Y \wedge (\exists\, Z : \mathcal{V}_X \Longmapsto \mathcal{V}_Y \bullet X = Z \fatsemi Y))$

A lens $X : V_1 \Longrightarrow S$ is narrower than a lens $Y : V_2 \Longrightarrow S$ provided that they share the same state space, and there exists a total lens $Z : V_1 \Longmapsto V_2$, such that X is the same as $Z \fatsemi Y$. In other words, the behaviour of X is defined by firstly viewing the state using Y, and secondly viewing a subregion of this using Z. An order characterises the size of a lens's aperture: how much of the state space a lens can view. For example, we can prove that $x_j \fatsemi o_k \preceq o_k$, by setting $Z = x_j$ in Definition 3.22. For the same reason, we can also show that $hp[loc] \preceq hp$. The lens order relation is a preorder, as demonstrated below.

Theorem 3.23. *For any $\mathcal{S}$, $(* \Longmapsto \mathcal{S}, \preceq)$ forms a preorder, that is, $\preceq$ is reflexive and transitive. Furthermore, the least element of $* \Longmapsto \mathcal{S}$ is $\mathbf{0}_\mathcal{S}$, and the greatest element is $\mathbf{1}_\mathcal{S}$.*

Clearly, **0** is the narrowest possible lens since it allows us to view nothing, and **1** is the widest lens, since it views the entire state. This is consistent with the intuition that **0** represents the empty set, **1** represents the set of all lenses, and $\preceq$ is a subset-like operator. We can prove the following intuitive theorem for sublenses.

Lemma 3.24. *If X, Y are total lenses and $X \preceq Y$, then the following identities hold:*

$$\mathsf{put}_Y\,(\mathsf{put}_X\,s\,v)\,u = \mathsf{put}_Y\,s\,u \tag{LS1}$$

$$\mathsf{get}_X\,(\mathsf{put}_Y\,s\,v) = \mathsf{get}_X\,(\mathsf{put}_Y\,s'\,v) \tag{LS2}$$

Law LS1 is a generalisation of Axiom PutPut: a later put_Y overrides an earlier put_X when $X \preceq Y$. Law LS2 states that when viewing an update on Y via a narrower lens X we can ignore the valuation of the original state, since the update replaces all the relevant information. We can now use these results to prove a number of ordering lemmas for lens compositions.

Lemma 3.25 (Lens Order)**.** *If X, Y, and Z are total lenses then they satisfy the following laws:*

$$X \fatsemi Y \preceq Y \tag{LO1}$$

$$X \preceq Y \wedge Y \bowtie Z \Rightarrow X \bowtie Z \tag{LO2}$$

$$X \bowtie Z \wedge Y \preceq Z \Rightarrow (X + Y) \preceq (X + Z) \tag{LO3}$$

$$X \bowtie Y \Rightarrow X + Y \preceq Y + X \tag{LO4}$$

$$X \bowtie Y \wedge X \bowtie Z \wedge Y \bowtie Z \Rightarrow X + (Y + Z) \preceq (X + Y) + Z \tag{LO5}$$

As we observed, composition of X and Y yields a narrower lens than Y (LO1): $hp[loc] \preceq hp$. Independence is preserved by the ordering, since a subregion of a larger independent region is also clearly independent (LO2) – $hp[loc] \bowtie x$ when $hp \bowtie x$. Lens sum also preserves the ordering in its right-hand component (LO3). Moreover, lens sum is commutative with respect to $\preceq$ (LO4), and also associative, assuming appropriate

independence properties (LO5). From these laws, and utilising the preorder theorems, we can prove various useful corollaries, such as

$$X \bowtie Y \Rightarrow X \preceq (X + Y)$$

which shows that X is narrower than $X + Y$: the latter is an upper bound. Thus, if we intuitively interpret $\preceq$ as $\subseteq$, then $+$ corresponds to $\cup$, and we can combine independent lens sets: $\{a, b\} \cup \{c, d\} = (a + b) + (c + d)$. We can also show that

$$X \preceq Z \wedge Y \preceq Z \wedge X \bowtie Y \Rightarrow (X + Y) \preceq Z$$

which shows that sum provides the least upper bound: $\cup$ preserves $\subseteq$. Finally, we can also induce an equivalence relation on lenses using the lens order in the usual way:

Definition 3.26 (Lens Equivalence)**.** $(X \approx Y) \triangleq (X \preceq Y \wedge Y \preceq X)$

We define $\approx$ as the cycle of a preorder, and consequently we can prove that it forms an equivalence relation.

Corollary 3.27 (Lens Equivalence Relation)**.** *For any $\mathcal{S}$, $(* \Longmapsto \mathcal{S}, \approx)$ forms a setoid, that is, $\approx$ is an equivalence relation on the set $* \Longmapsto \mathcal{S}$ – it is reflexive, symmetric, and transitive.*

Proof. Reflexivity and transitivity follow by Theorem 3.23, and symmetry follows by definition. □

Lens equivalence is a heterogeneously typed relation that is different from equality ($X = Y$), since it requires only that the two state spaces are the same, whilst the view types of X and Y can be different. Consequently, it can be used to compare lenses of different view types and show that two sets of lenses are isomorphic. This makes this relation much more useful for evaluating observational equivalence between two lenses that have apparently differing views, and yet characterise precisely the same region. For example, in general we cannot show that $x + y + z = z + y + x$, since these constructions have different view types: $\mathcal{V}_x \times \mathcal{V}_y \times \mathcal{V}_z$ and $\mathcal{V}_z \times \mathcal{V}_y \times \mathcal{V}_x$, respectively, and so the formula is not even type correct. We can, however, show that $x + y + z \approx z + y + x$. This is reflected by the following set of algebraic laws.

Theorem 3.28. *If X, Y, and Z are all total lenses then they satisfy the following identities:*

$$\begin{aligned} X + (Y + Z) &\approx (X + Y) + Z && \text{if } X \bowtie Y, X \bowtie Z, Y \bowtie Z \\ X + \mathbf{0} &\approx X \\ X + Y &\approx Y + X && \text{if } X \bowtie Y \end{aligned}$$

Lens summation is associative, has $\mathbf{0}$ as a unit, and commutative, modulo $\approx$, and assuming independence of the components, which is consistent with the lens set interpretation. Lenses thus form a partial commutative monoid [24], modulo $\approx$, also known as a separation algebra [19], where $X + Y$ is effectively defined only when $X \bowtie Y$. Independence corresponds with separation algebra's "separateness" relation, which means that there is no overlap between two areas of memory. We can also use $\approx$ to determine whether two independent lenses, X and Y, partition the entire state space using the identity $X + Y \approx \mathbf{1}$ [34]. Finally, we can prove the following additional properties of equivalence.

Theorem 3.29. *If X_1, X_2, Y_1, Y_2, and Y are total lenses then the following laws hold:*

- *If $X_1 \bowtie Y$ and $X_1 \approx X_2$ then $X_2 \bowtie Y$;*
- *If $X_1 \approx X_2$, $Y_1 \approx Y_2$, and $X_1 \bowtie Y_1$ then $X_1 + Y_1 \approx X_2 + Y_2$;*
- *If $\mathcal{S}_{X_1} = \mathcal{V}_Y$ and $X_1 \approx X_2$ then $X_1 \fatsemi Y \approx X_2 \fatsemi Y$.*

Independence is, as can be expected, preserved by equivalence. Equivalence is a congruence relation with respect to $+$, provided the summed lenses are independent. It is also a left congruence for lens composition. We can neither prove that it is a right congruence, nor find a counterexample.

3.6. Mechanised State Spaces

Lenses allow us to express provisos in laws with side conditions about variables. Manual construction of state spaces using the lens combinators is tedious and so we have implemented an Isabelle/HOL command for automatically creating a new state space with the following form:

$$\textbf{\textit{alphabet}}\ [\alpha_1, \cdots, \alpha_k] S = ([\beta_1, \cdots, \beta_m] T +)^?\ x_1 : \tau_1\ \cdots\ x_n : \tau_n$$

We name the command ***alphabet***, since it effectively allows the definition of a UTP alphabet (see Section 2.1), which in turn induces a state space. The command creates a new state space type S with k type parameters (α_i, for $1 \leq i \leq k$), optionally extending the parent state space T with m type parameters (β_i, for $1 \leq i \leq m$), and creates a lens for each of the variables $x_i : \tau_i$. It can be used to describe a concrete state space for a program. For brevity, we often abbreviate the ***alphabet*** command by the syntax

$$[\alpha_1, \cdots, \alpha_k]\ S \triangleq [\beta_1, \cdots, \beta_m] T + [x_1 : \tau_1,\ \cdots\ , x_n : \tau_n]$$

when used in mathematical definitions. Internally, the command performs the following steps:

1. generates a record space type S with n fields, which optionally extends a parent state space T;
2. generates a lens x_i for each of the fields using the record lens $\textbf{\textit{rec}}^R_{x_i}$;
3. automatically proves that each lens x_i is a total lens;
4. automatically proves an independence theorem $x_i \bowtie x_j$ for each pair $i, j \in \{1 \cdots n\}$ such that $i \neq j$;
5. generates lenses $\textbf{\textit{base}}_S$ and $\textbf{\textit{more}}_S$ that characterise the "base part" and "extension part", respectively;
6. automatically proves a number of independence and equivalence properties.

We now elaborate on each of these steps in detail.

The new record type S yields an auxiliary type $[\alpha_1, \cdots, \alpha_k, \phi] S\text{-}ext$ with additional type parameter ϕ that characterises future extensions. In particular, the non-extended type $[\alpha_1, \cdots, \alpha_k] S$ is characterised by $[\alpha_1, \cdots, \alpha_k, unit] S\text{-}ext$, where *unit* is a distinguished singleton type. This extensible record type is isomorphic to a product of three basic component types:

$$([\beta_1, \cdots, \beta_m] T \times (\tau_1 \times \cdots \times \tau_n)) \times \phi$$

These characterise, respectively, the part of state space described by T, the part described by the n additional fields, and the extension part ϕ. In the case that the state space does not extend an existing type, we can set $[\beta_1, \cdots, \beta_m] T = unit$.

For each field, the command generates a lens $x_i : \tau_i \Longrightarrow [\alpha_1, \cdots, \alpha_k, \phi] S\text{-}ext$ using the record lens, and proves total lens and independence theorems. Each of these lenses is polymorphic over ϕ, so that they can be applied to the base type and any extension thereof, in the style of inheritance in object oriented data structures. As we show in §6, this polymorphism allows us to characterise a hierarchy of UTP theories.

In addition to the field lenses, we create two special total lenses:

- $\textbf{\textit{base}}_S : [\alpha_1, \cdots, \alpha_k] S \Longrightarrow [\alpha_1, \cdots, \alpha_k, \phi] S\text{-}ext$, which characterises the base part; and
- $\textbf{\textit{more}}_S : \phi \Longrightarrow [\alpha_1, \cdots, \alpha_k, \phi] S\text{-}ext$, which characterises the extension part.

The base part consists of only the inherited fields and those added by S. We automatically prove a number of theorems about these special lenses:

- $\textbf{\textit{base}}_S \bowtie \textbf{\textit{more}}_S$: the base and extension parts are independent;
- $\textbf{\textit{base}}_S + \textbf{\textit{more}}_S \approx \mathbf{1}$: they partition the entire state space;
- for $i \in \{1 \cdots n\}$, $x_i \preceq \textbf{\textit{base}}_S$: each variable lens is part of the base;
- $\textbf{\textit{base}}_S \approx \textbf{\textit{base}}_T + \left(\sum_{i \in \{1 \cdots n\}} x_i\right)$: the base is composed of the parent's base and the variable lenses;

- $\textbf{\textit{more}}_T \approx \left(\sum_{i \in \{1 \cdots n\}} x_i\right) + \textbf{\textit{more}}_S$: the parent's extension is composed of the variable lenses and the child's extension part.

These theorems can help support the Isabelle/UTP laws of programming, which we elaborate in the next section. We emphasise, though, that the ***alphabet*** command is not the only way to construct a state space with lenses, and nor do the results that follow depend on the use of this command. We could, for example, axiomatise a collection of lenses, including independence relations over an abstract state space type, following Schirmer and Wenzel [73]. However, the ***alphabet*** command is a convenient tool in many circumstances.

4. Mechanising the UTP Relational Calculus

In this section we describe the core of Isabelle/UTP, including its expression model, meta-logical operators, predicate calculus, and relational calculus, building upon our algebraic model of state spaces. A direct result is an expressive model of relational programs which can be used in proving fundamental algebraic laws of programming [58], and for formal verification (§5). Moreover, the relational model is foundational to the mechanisation of UTP theories, and thus advanced computational paradigms, as we consider in §6. An overview of the Isabelle/UTP concepts and theories can be found in Fig. 12 at the end of the paper.

4.1. Expressions

Expressions are the basis of all other program and model objects in Isabelle/UTP, in that every such object is a specialisation of the expression type. An expression language typically includes literals, variables, and function symbols, all of which are also accounted for here. We model expressions as functions on the observation space: $[A, \mathcal{S}]\mathit{uexpr} \cong (\mathcal{S} \to A)$[11], where A is the return type, which is a standard shallow embedding approach [78, 6, 26]. However, lenses allow us to formulate syntax-like constraints, but without the need for deeply embedded expressions. A major advantage of this model is that we need not preconceive of all expression constructors, but can add them by definition.

We diverge from the standard shallow embedding approach, because we give explicit constructors for expressions. Usually shallow embeddings use syntax translations to transparently map between program expressions and the equivalent lifted expressions, for example,

$$e + (f - g) \rightsquigarrow \lambda\, s \bullet e(s) + (f(s) - g(s))$$

Here, we prefer to have expression constructors as first-class citizens.

Definition 4.1 (Expression Constructors)**.** Assume types A, B, C, and $\mathcal{S}$. We declare the constants:

$$\begin{aligned}
&\mathit{var} : (A \Longrightarrow \mathcal{S}) \to [A, \mathcal{S}]\mathit{uexpr} \\
&\mathit{var}\, x \triangleq \lambda\, s \bullet \mathit{get}_x\, s \\
&\mathit{lit} : A \Longrightarrow [A, \mathcal{S}]\mathit{uexpr} \\
&\mathit{lit}\, k \triangleq \lambda\, s \bullet k \\
&\mathit{cond} : [\mathbb{B}, \mathcal{S}]\mathit{uexpr} \to [A, \mathcal{S}]\mathit{uexpr} \to [A, \mathcal{S}]\mathit{uexpr} \to [A, \mathcal{S}]\mathit{uexpr} \\
&\mathit{cond}\, b\, u_1\, u_2 \triangleq \lambda\, s \bullet \mathsf{if}\ b(s)\ \mathsf{then}\ u_1(s)\ \mathsf{else}\ u_2(s) \\
&\mathit{uop} : (A \to B) \to [A, \mathcal{S}]\mathit{uexpr} \to [B, \mathcal{S}]\mathit{uexpr} \\
&\mathit{uop}\, f\, u \triangleq \lambda\, s \bullet f(u(s)) \\
&\mathit{bop} : (A \to B \to C) \to [A, \mathcal{S}]\mathit{uexpr} \to [B, \mathcal{S}]\mathit{uexpr} \to [C, \mathcal{S}]\mathit{uexpr} \\
&\mathit{bop}\, g\, u\, v \triangleq \lambda\, s \bullet g(u(s))\,(v(s))
\end{aligned}$$

where $x : A \Longrightarrow \mathcal{S}$ is a lens; $k : A$ is a HOL constant; $f : A \to B$ and $g : A \to B \to C$ are functions; and $b : [\mathbb{B}, \mathcal{S}]\mathit{uexpr}$, $u, u_1, u_2 : [A, \mathcal{S}]\mathit{uexpr}$, and $v : [B, \mathcal{S}]\mathit{uexpr}$ are expressions.

[11] We use a ***typedef*** to create an isomorphic but distinct type. This allows us to have greater control over definition of polymorphic constants and syntax translations, without unnecessarily constraining these for the function type.

```
typedef ('t, 'α) uexpr = "UNIV :: ('α ⇒ 't) set" ..

setup_lifting type_definition_uexpr

lift_definition var :: "('a ⟹ 's) ⇒ ('a, 's) uexpr" is "λ x s. get⇘x⇙ s" .

lift_definition lit :: "'a ⇒ ('a, 's) uexpr" ("«_»") is "λ v s. v" .

lift_definition uop :: "('a ⇒ 'b) ⇒ ('a, 's) uexpr ⇒ ('b, 's) uexpr" is "λ f u s. f (u s)" .

lift_definition bop :: "('a ⇒ 'b ⇒ 'c) ⇒ ('a, 's) uexpr ⇒ ('b, 's) uexpr ⇒ ('c, 's) uexpr"
is "λ f u v s. f (u s) (v s)" .

lift_definition cond :: "(bool, 's) uexpr ⇒ ('a, 's) uexpr ⇒ ('a, 's) uexpr ⇒ ('a, 's) uexpr"
  ("(3_ ◁ _ ▷/ _)" [52,0,53] 52)
  is "λ b u₁ u₂ s. if (b s) then u₁ s else u₂ s" .
```

Figure 6: UTP Expression Model in Isabelle/HOL

The mechanisation of the core expression language is shown in Fig. 6, which uses Isabelle's lifting package [60] to create each of the expression constructors. The operator $\mathit{var}\, x$ is a variable expression, and returns the present value in the state characterised by lens x. For convenience, we assume that x, y, z, and decorations thereof, are lenses, and often use them directly as variable expressions without explicitly using *var*. We also use $\boldsymbol{v}$ to denote the $\mathbf{1}$ lens; this is effectively a special variable for the entire state.

The operator $\mathit{lit}\, k$ represents a literal, or alternatively an arbitrary lifted HOL value, and corresponds to a constant function expression. We use the notation $\ll k \gg$ to denote a literal k. As well as lens-based variables, which are used to model program variables, expressions can also contain HOL variables, which are orthogonal and constant with respect to the program variables. HOL variables in literal constructions ($\ll x \gg$) correspond to logical variables [61], also called "ghost variables", which are important for verification [54, 61]. We use the notations x, y, and z to denote logical variables in expressions.

Operator $\mathit{cond}\, b\, u_1\, u_2$ denotes a conditional expression; if b is true then it returns u_1, otherwise u_2. It evaluates the boolean expression b under the incoming state, and chooses the expression based on this. We use the notation $u_1 \lhd b \rhd u_2$ adopted in the UTP as a short hand for it.

Operators *uop* and *bop* lift HOL functions into the expression space by a pointwise lifting. With them we can transparently use HOL functions as UTP expression functions, for instance the summation $e + f$ is denoted by $\mathit{bop}\,(+)\, e\, f$. Moreover, it is often possible to lift theorems from the underlying operators to the expressions themselves, which allows us to reuse the large library of HOL algebraic structures in Isabelle/UTP. For instance, if we know that $(A, +, 0)$ is a monoid, then also we can show that for any $\mathcal{S}$, $([A, \mathcal{S}]\mathit{uexpr}, \mathit{bop}\,(+), \mathit{lit}\, 0)$ forms a monoid. For convenience, we therefore often overload mathematically defined functions as expression constructs without further comment. In particular, we often overload $=$ as both equivalence of two expressions ($e = f$), and an expression of equality within an expression ($x = 5$).

This deep expression model allows us to mimick reasoning usually found in a deep embedding: the constructors above are like datatype constructs, but are really semantic definitions. We can then prove theorems about these constructs that allow us to reason in an inductive way, which is central to our approach to meta-logical reasoning. At the same time, we have developed a lifting parser in Isabelle/UTP, which allows automatic translation between HOL expressions and UTP expressions. We also have a tactic, rel-auto, that quickly and automatically eliminates the expression structure, resulting in a HOL expression.

The rel-auto tactic performs best when $\mathcal{S}$ is constructed using the ***alphabet*** command of §3.6, because we can enumerate all the field lenses. Given a state space $[x_1 : \tau_1, \cdots x_n \tau_n]$, we can eliminate the s state space variable in a proof goal by replacing it with a tuple of logical variables $(x_1 \cdots x_n)$. This, in turn, means that we can eliminate each lens and replace it with a corresponding logical variable, for example:

$$(x + y) - z > 3 \rightsquigarrow (x + y) - z > 3$$

The result is a simpler expression containing only logical variables, though of course with the loss of lens properties. This means that we have both the additional expressivity and fidelity afforded by lenses, and

the proof automation of Isabelle/HOL.

4.2. Predicate Calculus

A predicate is an expression with a Boolean return type, $[\mathcal{S}]\mathit{upred} \triangleq [\mathbb{B}, \mathcal{S}]\mathit{uexpr}$, so that predicates are a subtype of expressions. The majority of predicate calculus operators ($\neg$, $\wedge$, $\vee$, $\Rightarrow$) are obtained by pointwise lifting of the equivalent operators in HOL. We also define the indexed operators $\bigwedge_{i \in A} P(i)$ and $\bigvee_{i \in A} P(i)$ similarly. The quantifiers are defined below. In order to notationally distinguish HOL from UTP operators, in the following definitions we subscript them with a H.

Definition 4.2 (Predicate Calculus Operators)**.**

$$\begin{aligned} \exists\, x \bullet P &\triangleq \lambda\, s \bullet \exists_{\mathsf{H}}\, v : \mathcal{V}_x \bullet P(\mathit{put}_x\, s\, v) \\ \exists\, x \bullet P(x) &\triangleq \lambda\, s \bullet \exists_{\mathsf{H}}\, x \bullet P(x)(s) \\ [P] &\triangleq \forall\, \boldsymbol{v} \bullet P \\ (P \sqsubseteq Q) &\triangleq (\forall_{\mathsf{H}}\, s \bullet (Q \Rightarrow P)s) \end{aligned}$$

Existential quantification over a lens x quantifies possible values for the lens $v : \mathcal{V}_x$, and updates the state with this using put. Universal quantification is obtained by duality. The emboldened existential quantifier, $\exists$, quantifies a logical variable in a parametric predicate $P(x)$ by a direct lifting of the corresponding HOL quantifier. We emphasise that $\exists$ and $\exists$ are semantically very different: in a lens quantification, $\exists\, x \bullet P$, the lens x must be an existing lens or expression. This lens is not bound by the quantification, unlike $\exists\, x \bullet P(x)$ where x becomes a logical variable bound in P. Lens quantification is elsewhere called *liberation* [25, 22] since it removes any restrictions on the valuation of x.

The universal closure, $[P]$, universally quantifies every variable in the alphabet of P using the state variable $\boldsymbol{v}$. The refinement relation $P \sqsubseteq Q$ is then defined as a HOL predicate, requiring that Q implies P in every state s.

Since the definitions are by lifting of the underlying HOL operators, we obtain the following theorem.

Theorem 4.3. *For any $\mathcal{S}$, $([\mathcal{S}]\mathit{upred}, \bigvee, \mathit{false}, \bigwedge, \mathit{true}, \neg)$ forms a complete Boolean algebra, that is:*

- *$([\mathcal{S}]\mathit{upred}, \vee, \mathit{false}, \wedge, \mathit{true}, \neg)$ is a Boolean algebra, and*
- *$([\mathcal{S}]\mathit{upred}, \sqsubseteq)$ is a complete lattice with infimum $\bigvee$, supremum $\bigwedge$, top element false, and bottom true.*

As usual, via the Knaster-Tarski theorem, for any monotonic function $F : [\mathcal{S}]\mathit{upred} \to [\mathcal{S}]\mathit{upred}$ we can describe the least and greatest fixed points, $\mu\, F$ and νF, which in UTP are called the weakest and strongest fixed points, and obey the usual fixed point laws. We can also algebraically characterise the UTP variable quantifiers using Cylindric Algebra [53], which axiomatises the quantifiers of first-order logic.

Theorem 4.4. *For any $\mathcal{S}$, $([\mathcal{S}]\mathit{upred}, \vee, \wedge, \neg, \mathit{false}, \mathit{true}, \exists, =)$ forms a Cylindric Algebra, meaning that the following laws are satisfied for total lenses x, y, and z:*

$$(\exists\, x \bullet \mathit{false}) = \mathit{false} \tag{C1}$$

$$(\exists\, x \bullet P) \sqsubseteq P \tag{C2}$$

$$(\exists\, x \bullet (P \wedge (\exists\, x \bullet Q))) = ((\exists\, x \bullet P) \wedge (\exists\, x \bullet Q)) \tag{C3}$$

$$(\exists\, x \bullet \exists\, y \bullet P) = (\exists\, y \bullet \exists\, x \bullet P) \tag{C4}$$

$$(x = x) = \mathit{true} \tag{C5}$$

$$(y = z) = (\exists\, x \bullet y = x \wedge x = z) \qquad \mathit{if}\, x \bowtie y, x \bowtie z \tag{C6}$$

$$\mathit{false} = \begin{pmatrix} (\exists\, x \bullet x = y \wedge P) \wedge \\ (\exists\, x \bullet x = y \wedge \neg P) \end{pmatrix} \qquad \mathit{if}\, x \bowtie y \tag{C7}$$

From this algebra, the usual laws of quantification can be derived [53]. These laws illustrate the difference in expressive power between HOL and UTP variables. For the former, we cannot pose meta-logical questions like whether two variable names x and y refer to the same region, such as may be the case if they are aliased. For this kind of property, we can use lens independence $x \bowtie y$, as required by laws C6 and C7. We also prove the following laws for quantification.

Theorem 4.5. *If A and B are total lenses, then the following identities hold:*

$$(\exists\, A + B \bullet P) = (\exists\, A \bullet \exists\, B \bullet P) \qquad \textit{if}\, A \bowtie B \qquad \text{(Ex1)}$$

$$(\exists\, B \bullet \exists\, A \bullet P) = (\exists\, A \bullet P) \qquad \textit{if}\, B \preceq A \qquad \text{(Ex2)}$$

$$(\exists\, A \bullet P) = (\exists\, B \bullet P) \qquad \textit{if}\, A \approx B \qquad \text{(Ex3)}$$

Here, lenses A and B can be interpreted as variable sets. Ex1 shows that quantifying over two disjoint sets of variables equates to quantification over both. Disjointness of variable sets is modelled by requiring that the corresponding lenses are independent. Ex2 shows that quantification over a larger lens subsumes a smaller lens. Finally Ex3 shows that if we quantify over two lenses that identify the same subregion then those two quantifications are equal. We now have a complete set of operators and laws for the predicate calculus.

4.3. Meta-Logic

Lenses treat variables as semantic objects that can be checked for independence, ordered, and composed in various ways. As we have noted, we can consider such manipulations as meta-logical with respect to the predicate. We add further specialised meta-logical queries for expressions.

Often we want to check which regions of the state space an expression depends on, for example to support laws of programming and verification calculi, like Example 2.1. In a deep embedding, this is characterised syntactically using free variables. However, lenses allow us to characterise a corresponding semantic notion called "unrestriction", which is originally due to Oliveira et al. [66].

Definition 4.6 (Unrestriction)**.**

$$\text{-}\sharp\text{-} \;:\; (A \Longrightarrow S) \to [B, S]\mathit{uexpr} \to \mathbb{B}$$

$$(x \sharp e) \;\triangleq\; (\forall (s, k) \bullet e(\mathit{put}_x\, s\, k) = e(s))$$

Intuitively, lens x is unrestricted in expression e, written $x \sharp e$, provided that e's valuation does not depend on x. Specifically, the effect of e evaluated under state s is the same if we change the value of x. For example, $x \sharp (y + 2 > y)$ is true, provided that $x \bowtie y$, since the truth value of $y + 2 > y$ is unaffected by changing x. Unrestriction is a weaker notion than free variables: if x is not free in e then $x \sharp e$, but not the inverse. For example, $x \sharp (x = \mathit{true} \vee x = \mathit{false})$ for $x : \mathbb{B}$ is true, since this expression is always true no matter the valuation of x. As we shall see, unrestriction is a sufficient notion to formalise the provisos for the laws of programming. Below are some key laws for establishing whether an expression is unrestricted by a variable.

Lemma 4.7 (Unrestriction Laws)**.** *If x and y are total lenses, then the following laws hold:*

$$\frac{}{x \sharp \mathit{lit}\, k} \qquad \frac{x \bowtie y}{x \sharp \mathit{var}\, y} \qquad \frac{x \sharp u}{x \sharp \mathit{uop}\, f\, u} \qquad \frac{x \sharp u \quad x \sharp v}{x \sharp \mathit{bop}\, f\, u\, v} \qquad \frac{}{\mathbf{0} \sharp u} \qquad \frac{x \preceq y \quad y \sharp u}{x \sharp u} \qquad \frac{x \bowtie y \quad x \sharp u \quad y \sharp u}{(x + y) \sharp u}$$

$$\frac{x \preceq y}{x \sharp (\exists\, y \bullet P)} \qquad \frac{x \bowtie y \quad x \sharp P}{x \sharp (\exists\, y \bullet P)} \qquad \frac{x \preceq y}{x \sharp (\forall\, y \bullet P)} \qquad \frac{x \bowtie y \quad x \sharp P}{x \sharp (\forall\, y \bullet P)}$$

These laws are formulated in the style of an inductive definition, but in reality they are a set of lemmas in Isabelle over our deep expression model. Expression $\mathit{lit}\, k$ does not depend on the state since it always returns k: any lens x is unrestricted. Any lens y that is independent of x is unrestricted $\mathit{var}\, x$. The laws for uop and bop require simply that the component expressions have lens x unrestricted. The $\mathbf{0}$ lens is unrestricted in any expression u, since it characterises none of the state space.

Unrestriction is preserved by the lens order $\preceq$. The summation of lenses x and y is unrestricted in u provided that x and y are independent, and both are unrestricted in u. Again, we note that $+$ can effectively be used to group lenses in order to characterise a set of variables. Following a lens quantification over y, any sublens x of y becomes unrestricted. On the other hand, the restriction of any lens independent of x is unchanged. The final two laws are the dual case for the universal lens quantification.

We can also prove the following correspondence between predicate unrestriction and quantification.

Lemma 4.8. *If x is a total lens then $(x \sharp P) \Leftrightarrow (\exists\, x \bullet P) = P$.*

We can alternatively characterise unrestriction of x by showing that quantifying over x in e has no effect (it is a fixed point), which is a well-known property from Cylindric Algebra [53] and also Liberation Algebra [25, 22]. Specifically, quantification liberates the lens x so that it is free to take any value. Nevertheless, Lemma 4.8 cannot replace Definition 4.6, as it applies only if P is a predicate, and not for an arbitrary expression. We can prove a number of useful corollaries for quantification.

Corollary 4.9. *If $x \sharp P$ then $(\exists\, x \bullet P) = P$ and $(\exists\, x \bullet P \wedge Q) = (P \wedge (\exists\, x \bullet Q))$.*

The cases for universal quantification ($\forall$) also hold by duality.

Aside from checking for use of variables, UTP theories often require that the state space of a predicate can be extended with additional variables. An example of this is the variable block operator that adds a new local variable. In Isabelle/UTP, alphabets are implicitly characterised by state-space types, rather than explicitly as sets of variables. Consequently, we perform alphabet extensions by manipulation of the underlying state space using lenses. We define the following operator to extend a state space.

Definition 4.10 (Alphabet Extrusion)**.**

$$\begin{aligned} \text{-}\oplus\text{-} &: [A, S_1]\mathit{uexpr} \to (S_1 \Longrightarrow S_2) \to [A, S_2]\mathit{uexpr} \\ e \oplus a &\triangleq (\lambda\, s \bullet e(\mathit{get}_a\, s)) \end{aligned}$$

Alphabet extrusion, $e \oplus a$, uses the lens $a : S_1 \Longrightarrow S_2$ to alter the state space of $e : [A, S_1]\mathit{uexpr}$ to become S_2. The lens effectively describes how one state space can be embedded into another. The operator can be used either to extend a state space, or coerce it to an isomorphic one. For example, if we assume we have a state space composed of two sub-regions: $\mathcal{S} \triangleq A \times B$, then, a predicate $P : [A]\mathit{upred}$, which acts on state space A, can be coerced to one acting on $\mathcal{S}$ by an alphabet extrusion: $P \oplus \mathbf{fst} : [\mathcal{S}]\mathit{upred}$. Naturally, we know that the resulting predicate is unrestricted by the B region.

Lemma 4.11 (Alphabet Extrusion Laws)**.**

$$\begin{aligned} \mathit{lit}\, k \oplus a &= \mathit{lit}\, k & \mathit{var}\, x \oplus a &= \mathit{var}\,(x \fatsemi a) \\ \mathit{uop}\, f\, u \oplus a &= \mathit{uop}\, f\,(u \oplus a) & \mathit{bop}\, g\, u\, v \oplus a &= \mathit{bop}\, g\,(u \oplus a)\,(v \oplus a) \end{aligned}$$

Alphabet extrusion has no effect on a literal expression, other than to change its type, because it refers to no lenses. A variable constructor has its lens augmented by composing it with the alphabet lens a. The extrusion simply distributes through unary and binary expressions.

4.4. Substitutions

Substitution is an operator for replacing a variable in an expression with another expression: $e[f/x]$. Like free variables, it is often considered as a meta-logical operator [46]. However, shallow embeddings can support a similar operator at the semantic level [46, 4], for which the substitution laws can be proved as theorems. Here, we generalise this using lenses, and treat substitutions as first-class citizens that can contain multiple variable mappings, and also conditional substitutions. This allows us to unify substitution and multiple variable assignment, which we demonstrate in Theorem 4.21 (LP4) and Corollary 4.24.

Substitutions can be modelled as total functions $\sigma, \rho : \mathcal{S} \to \mathcal{S}$ that transform an initial state to a final state. However, it is more intuitive to consider a substitution as a set of mappings from variables to expressions: $[x \mapsto_{\mathsf{s}} e, y \mapsto_{\mathsf{s}} f]$. The simplest substitution, $id \triangleq (\lambda\, x \bullet x)$, leaves the state unchanged. We define operators for querying and updating our semantic substitution objects below.

Definition 4.12 (Substitution Query and Update)**.**

$$\langle\sigma\rangle_{\mathsf{s}}\, x \triangleq (\lambda\, s \bullet \mathit{get}_x\,(\sigma(s))) \qquad \sigma(x \mapsto_{\mathsf{s}} e) \triangleq (\lambda\, s \bullet \mathit{put}_x\,(\sigma(s))\,(e(s)))$$

Substitution query $\langle\sigma\rangle_{\mathsf{s}}\, x : [A, \mathcal{S}]\mathit{uexpr}$ returns the expression associated with $x : A \Longrightarrow \mathcal{S}$ in $\sigma : \mathcal{S} \to \mathcal{S}$ by composition of the latter with get_x. Substitution update assigns the expression $e : [A, \mathcal{S}]\mathit{uexpr}$ to the lens x in σ. The definition constructs a function of type $\mathcal{S} \to \mathcal{S}$ that inputs the state s, evaluates e with respect to s, calculates the state space updated by σ, and then uses put to update the value of x in $\sigma(s)$ with the evaluated expression. We can then introduce the short-hands

$$\sigma(x_1 \mapsto_{\mathsf{s}} e_1, \cdots, x_n \mapsto_{\mathsf{s}} e_n) \triangleq (x_1 \mapsto_{\mathsf{s}} e_1) \cdots (x_n \mapsto_{\mathsf{s}} e_n)$$
$$[x_1 \mapsto_{\mathsf{s}} e_1, \cdots, x_n \mapsto_{\mathsf{s}} e_n] \triangleq \mathit{id}(x_1 \mapsto_{\mathsf{s}} e_1, \cdots, x_n \mapsto_{\mathsf{s}} e_n)$$

which, respectively, update a substitution σ in n variables by assigning e_i to each variable x_i, and construct a new substitution from a set of maplets. If we have $x_i \bowtie x_j$ for each such variable, then these updates are effectively concurrent, regardless of the expressions. Otherwise, a syntactically later assignment (x_j) can override an earlier one (x_i with $i < j$). This allows us to model multiple variable assignment, and also evaluation contexts for programs and expressions.

We also prove a number of laws about substitutions constructed from maplets.

Lemma 4.13. *If x and y are total lenses, then the following identities hold:*

$$\langle\sigma(x \mapsto_{\mathsf{s}} e)\rangle_{\mathsf{s}}\, x = e \tag{SB1}$$
$$\sigma(x \mapsto_{\mathsf{s}} \langle\sigma\rangle_{\mathsf{s}}\, x) = \sigma \tag{SB2}$$
$$[x \mapsto_{\mathsf{s}} x] = \mathit{id} \tag{SB3}$$
$$\sigma(x \mapsto_{\mathsf{s}} e, y \mapsto_{\mathsf{s}} f) = \sigma(y \mapsto_{\mathsf{s}} f, x \mapsto_{\mathsf{s}} e) \quad \text{if } x \bowtie y \tag{SB4}$$
$$\sigma(x \mapsto_{\mathsf{s}} e, y \mapsto_{\mathsf{s}} f) = \sigma(y \mapsto_{\mathsf{s}} f) \quad \text{if } x \preceq y \tag{SB5}$$

Law SB1 states that looking up x in a constructed substitution returns its associated expression, e. It essentially follows from lens axiom PutGet. Law SB2 is an η-conversion principle, which follows from GetPut. Law SB3 states that updating a variable to itself has no effect. Law SB4 states that two substitution maplets, $x \mapsto_{\mathsf{s}} e$ and $y \mapsto_{\mathsf{s}} f$, commute provided that x and y are independent. Similarly, SB5 states that a later assignment for y overrides an earlier assignment for x when y is a wider lens than x. This is, of course, true in particular when $x = y$, which reduces to lens axiom PutPut.

Substitutions can be composed in sequence using function composition. Conditional substitutions can be expressed using the following construct.

Definition 4.14 (Conditional Substitution)**.** $\sigma \blacktriangleleft b \blacktriangleright \rho \triangleq (\lambda\, s \bullet \mathsf{if}\ b(s)\ \mathsf{then}\ \sigma(s)\ \mathsf{else}\ \rho(s))$.

A conditional substitution is equivalent to σ when b is true, and ρ otherwise. The definition evaluates b under the incoming state s, and then chooses which substitution to apply based on this. Substitutions can be applied to an expression using the following operator.

Definition 4.15 (Substitution Application)**.**

$$-\dagger- : (\mathcal{S} \to \mathcal{S}) \to [A, \mathcal{S}]\mathit{uexpr} \to [A, \mathcal{S}]\mathit{uexpr}$$
$$\sigma \dagger e \triangleq (\lambda\, s \bullet e(\sigma(s)))$$

Application of a substitution σ to an expression e simply evaluates e in the context of state $\sigma(s)$. We can also model the classical syntax for substitution, $P[v/x] \triangleq [x \mapsto_{\mathsf{s}} v] \dagger P$, and prove the substitution laws [4].

Lemma 4.16 (Substitution Application Laws)**.**

$$\sigma \dagger \mathit{var}\, x = \langle \sigma \rangle_{\mathsf{s}}\, x \tag{SA1}$$
$$\sigma(x \mapsto_{\mathsf{s}} e) \dagger u = \sigma \dagger u \qquad \text{if } x \sharp u \tag{SA2}$$
$$\sigma \dagger \mathit{uop}\, f\, v = \mathit{uop}\, f\, (\sigma \dagger v) \tag{SA3}$$
$$\sigma \dagger \mathit{bop}\, f\, u\, v = \mathit{bop}\, f\, (\sigma \dagger u)\, (\sigma \dagger v) \tag{SA4}$$
$$(\exists\, y \bullet P)[e/x] = (\exists\, y \bullet P[e/x]) \qquad \mathit{if}\, x \bowtie y, y \sharp e \tag{SA5}$$
$$\mathit{id} \dagger e = e \tag{SA6}$$
$$\sigma \dagger \rho \dagger e = (\rho \circ \sigma) \dagger e \tag{SA7}$$
$$\rho(x \mapsto_{\mathsf{s}} e) \circ \sigma = (\rho \circ \sigma)(x \mapsto_{\mathsf{s}} \sigma \dagger e) \tag{SA8}$$
$$\sigma(x \mapsto_{\mathsf{s}} e) \blacktriangleleft b \blacktriangleright \rho(x \mapsto_{\mathsf{s}} f) = (\sigma \blacktriangleleft b \blacktriangleright \rho)(x \mapsto_{\mathsf{s}} (e \triangleleft b \triangleright f)) \tag{SA9}$$

Application of σ to a variable x is the valuation of x in σ (SA1). A substitution maplet for an unrestricted variable can be removed (SA2). Substitutions distribute through both unary (SA3) and binary (SA4) operators. A singleton substitution for variable x can pass through an existential quantification over y provided that x and y are independent, and e is unrestricted by y (SA5), which prevents variable capture. Application of id has no effect (SA6), and application of two substitutions can be expressed by their composition (SA7). SA8 shows that when σ is composed with another substitution composed of maplets $(x \mapsto_{\mathsf{s}} e)$, it is simply applied to the expression e of every such maplet. SA9 shows how two substitutions with matching maplets can be conditionally composed, by distributing the conditional.

We can also use substitutition and unrestriction to prove the one-point law [50] from predicate calculus, as employed in Hehner's classic textbook on predicative programming [51].

Theorem 4.17. *If x is a total lens and $x \sharp e$ then $(\exists\, x \bullet P \wedge x = e) = P[e/x]$*

As for expressions, we define an operator to extend the alphabet of a substitution.

Definition 4.18 (Substitution Alphabet Extrusion)**.** $\sigma \oplus_{\mathsf{s}} a \triangleq (\lambda\, s \bullet \mathit{put}_a\, s\, (\sigma(\mathit{get}_a\, s)))$

We use the lens $a : S_1 \Longrightarrow S_2$ to coerce $\sigma : S_1 \to S_1$ to the state space S_2. The resulting substitution first obtains an element of S_1 from the incoming state $s : S_2$ using get_a, applies the substitution to this, and then places the updated state back into s using put_a. This is the essence of a framed computation; the parts of s outside of the view of a are unchanged.

4.5. Relational Programs

A relation is a predicate on a product space $\mathcal{S}_1 \times \mathcal{S}_2$, specifically, where $\mathcal{S}_1$ and $\mathcal{S}_2$ are the state spaces before and after execution, respectively. All laws that have been proved for expressions and predicates therefore hold for relations. We define types for both heterogeneous relations, $[\mathcal{S}_1, \mathcal{S}_2]\mathit{urel} \triangleq [\mathcal{S}_1 \times \mathcal{S}_2]\mathit{upred}$, and homogeneous relations $[\mathcal{S}]\mathit{hrel} \triangleq [\mathcal{S}, \mathcal{S}]\mathit{urel}$. Operators *true* and *false* can be specialised in this relational setting, and stand for the most and least non-deterministic relations. Due to their special role, we use the notation ***true*** and ***false*** to explicitly refer to these relational counterparts.

In common with formal languages like Z [75] and B [1], UTP [57] uses the notational convention for variables that x is the initial value and x' is its final value. The former can be denoted by the lens composition $x ; \mathit{fst}$, and the latter by $x ; \mathit{snd}$, where x is actually the name of a lens of type $\tau \Longrightarrow \mathcal{S}$. In this presentation we define the operators below for lifting variables, expressions, and substitutions into the product space.

Definition 4.19 (Pre- and Postcondition Lifting)**.**

$$x^{\blacktriangleleft} \triangleq x ; \mathit{fst} \qquad x^{\blacktriangleright} \triangleq x ; \mathit{snd} \qquad e^{\blacktriangleleft} \triangleq e \oplus \mathit{fst} \qquad e^{\blacktriangleright} \triangleq e \oplus \mathit{snd} \qquad \sigma^{\blacktriangleleft} \triangleq \sigma \oplus_{\mathsf{s}} \mathit{fst} \qquad \sigma^{\blacktriangleright} \triangleq \sigma \oplus_{\mathsf{s}} \mathit{snd}$$

The $^{\blacktriangleleft}$ and $^{\blacktriangleright}$ lift a lens, expression, or substitution, into the first and second components of a product state space $\mathcal{S}_1 \times \mathcal{S}_2$. We deviate from the standard notation to avoid the ambiguity between "x" meaning an

expression variable or an initial relational variable. Operator $e^{\blacktriangleleft}$ lifts an expression $e : [A, \mathcal{S}_1]\mathit{uexpr}$ to an expression on the product state space $\mathcal{S}_1 \times \mathcal{S}_2$, for any given $\mathcal{S}_2$. If e is a predicate on the state variables, then $e^{\blacktriangleleft}$ is a predicate on the initial state, that is a precondition. Similarly, $e^{\blacktriangleright}$ constructs a postcondition with state space $\mathcal{S}_1 \times \mathcal{S}_2$ from $e : [A, \mathcal{S}_2]\mathit{uexpr}$. The analogous operators $\sigma^{\blacktriangleleft}$ and $\sigma^{\blacktriangleright}$ lift a substitution to the product space.

We can now define the main programming operators of the relational calculus.

Definition 4.20 (Programming Operators)**.**

$$
\begin{aligned}
(P \mathbin{;} Q) &\triangleq (\exists\, v_0 \bullet P[v_0/\boldsymbol{v}^{\blacktriangleright}] \wedge Q[v_0/\boldsymbol{v}^{\blacktriangleleft}]) \\
\mathit{II} &\triangleq (\boldsymbol{v}^{\blacktriangleright} = \boldsymbol{v}^{\blacktriangleleft}) \\
\langle \sigma \rangle &\triangleq (\boldsymbol{v}^{\blacktriangleright} = \sigma(\boldsymbol{v}^{\blacktriangleleft}))
\end{aligned}
\qquad\qquad
\begin{aligned}
P \lhd b \rhd Q &\triangleq (b^{\blacktriangleleft} \wedge P) \vee (\neg b^{\blacktriangleleft} \wedge Q) \\
b \circledast Q &\triangleq \nu X \bullet P \mathbin{;} X \lhd b \rhd \mathit{II}
\end{aligned}
$$

These broadly follow the common definitions given in relational calculus and the UTP book [57], but with subtle differences due to our use of lenses. Relational composition $P \mathbin{;} Q$ existentiality quantifies a logical variable v_0 that stands for the intermediate state between P and Q. It is substituted as the final state of P (using $\boldsymbol{v}^{\blacktriangleright}$), and the initial state of Q; the resulting predicates are then conjoined. This definition yields a homogeneous composition operator, though in Isabelle/UTP composition is heterogeneous and has type $[\mathcal{S}_1, \mathcal{S}_2]\mathit{urel} \to [\mathcal{S}_2, \mathcal{S}_3]\mathit{urel} \to [\mathcal{S}_1, \mathcal{S}_3]\mathit{urel}$[12].

Relational identity (II), or skip, equates the initial state with the final state, leaving all variables unchanged. $\langle \sigma \rangle$ is a generalised assignment operator, originally proposed by Back and von Wright [6, page 2] using a substitution $\sigma : \mathcal{S} \to \mathcal{S}$. Its definition states that the final state is equal to the initial state with σ applied. The substitution σ can be constructed as a set of maplets, so that a singleton assignment $x := v$ can be expressed as $\langle x \mapsto_{\mathbf{s}} v \rangle$, and a multiple variable assignment as $\langle x_1 \mapsto_{\mathbf{s}} v_1, \cdots, x_n \mapsto_{\mathbf{s}} v_n \rangle$. Since x can be any lens in an assignment, we can use it to assign any part of the state hierarchy that can be so characterised, for example an element of the heap (Example 2.3) or the attribute of an object (Example 3.16).

Conditional $P \lhd b \rhd Q$ states that if b is true then behave like P, otherwise behave like Q. In the UTP book [57] the fact that b acts on initial variables is a syntactic convention, whereas here this wellformedness condition is imposed by construction using $b^{\blacktriangleleft}$. For completeness, we use this operator, and the fact that all predicates, including relations, form a complete lattice, to define the while loop operator $b \circledast P$. We use the strongest fixed-point, ν, since we use it for partial correctness verification in §5.

We now have all the operators of a simple imperative programming language, and can prove the laws of programming [58, 57], but with a generalised presentation.

Theorem 4.21 (Generalised Laws of Programming)**.**

$$(P \mathbin{;} Q) \mathbin{;} R = P \mathbin{;} (Q \mathbin{;} R) \quad \text{(LP1)}$$
$$\mathit{II} \mathbin{;} P = P \mathbin{;} \mathit{II} = P \quad \text{(LP2)}$$
$$\mathit{false} \mathbin{;} P = P \mathbin{;} \mathit{false} = \mathit{false} \quad \text{(LP3)}$$
$$\langle \sigma \rangle \mathbin{;} P = \sigma^{\blacktriangleleft} \dagger P \quad \text{(LP4)}$$
$$\langle \sigma \rangle \lhd b \rhd \langle \rho \rangle = \langle \sigma \blacktriangleleft b \blacktriangleright \rho \rangle \quad \text{(LP5)}$$
$$(P \mathbin{;} (b \circledast P)) \lhd b \rhd \mathit{II} = b \circledast P \quad \text{(LP6)}$$
$$P \lhd \mathit{true} \rhd Q = P \quad \text{(LP7)}$$
$$P \lhd \mathit{false} \rhd Q = Q \quad \text{(LP8)}$$
$$P \lhd \neg b \rhd Q = Q \lhd b \rhd P \quad \text{(LP9)}$$
$$(P \lhd b \rhd Q) \mathbin{;} R = (P \mathbin{;} R) \lhd b \rhd (P \mathbin{;} R) \quad \text{(LP10)}$$

The majority of these are standard, and therefore we select only a few for commentary. LP4 is a generalisation of the forward assignment law: an assignment by σ followed by relation P is equivalent to σ applied to the initial variables of P. The more traditional formulation [58, 57], $x := e \mathbin{;} P = P[e/x]$, is an instance of this law. Similarly, LP5 is a generalised conditional assignment law which combines σ and ρ into a

[12] Technically, in Isabelle/UTP we obtain ; by lifting the HOL relational composition operator, and so the equation in Definition 4.20 is actually a special case theorem. Nevertheless, it gives adequate intuition for this paper, and avoids the introduction of a further layer of abstraction.

single conditional substitution. All the other assignment laws can be proved, but we need some additional properties for relational substitutions, which are shown below. LP10 would normally require as a proviso that b is an expression in initial variables only, but in our setting this fact follows by construction.

Lemma 4.22 (Relational Substitutions and Assignment)**.**

$$\sigma^{\blacktriangleleft} \dagger (P \mathbin{;} Q) = (\sigma^{\blacktriangleleft} \dagger P) \mathbin{;} Q \quad \text{(RS1)} \qquad \sigma^{\blacktriangleleft} \dagger \langle \rho \rangle = \langle \rho \circ \sigma \rangle \quad \text{(RS3)}$$

$$\sigma^{\blacktriangleright} \dagger (P \mathbin{;} Q) = P \mathbin{;} (\sigma^{\blacktriangleright} \dagger Q) \quad \text{(RS2)} \qquad \sigma^{\blacktriangleleft} \dagger (P \lhd b \rhd Q) = (\sigma^{\blacktriangleleft} \dagger P) \lhd (\sigma \dagger b) \rhd (\sigma^{\blacktriangleleft} \dagger Q) \quad \text{(RS4)}$$

RS1 shows that a precondition substitution applies only to the first element of a sequential composition, and RS2 is its dual. RS3 shows that precondition substitution applied to an assignment can be expressed as a composite assignment. RS4 shows that $\sigma^{\blacktriangleleft}$ applied to a conditional distributes through all three arguments. The precondition annotation is removed when applied to b since this is a precondition expression already. Combining Theorem 4.21 with Lemma 4.22 we can prove the following corollaries of generalised assignment.

Corollary 4.23. $\langle id \rangle = \mathit{II} \qquad \langle \sigma \rangle \mathbin{;} \langle \rho \rangle = \langle \rho \circ \sigma \rangle \qquad \langle \sigma \rangle \mathbin{;} (P \lhd b \rhd Q) = (\langle \sigma \rangle \mathbin{;} P) \lhd \sigma \dagger b \rhd (\langle \sigma \rangle \mathbin{;} Q)$

Moreover, with Lemma 4.16, Theorem 4.21, and Lemma 4.22 we can prove the classical assignment laws [58].

Corollary 4.24 (Assignment Laws)**.**

$$\begin{aligned} x := e \mathbin{;} y := f &= x, y := e, f[e/x] & x, y, z := e, f, g &= y, x, z := f, e, g \\ x := x &= \mathit{II} & x := e \mathbin{;} (P \lhd b \rhd Q) &= (x := e \mathbin{;} P) \lhd b[e/x] \rhd (x := e \mathbin{;} Q) \\ x, y := e, y &= x := e & x := e \lhd b \rhd x := f &= x := (e \lhd b \rhd f) \end{aligned}$$

With these laws we can collapse any sequential and conditional composition of assignments into a single assignment [58]. We illustrate this with the calculation below.

Example 4.25 (Assignment Calculation)**.** Assume x, y, and z are independent total lenses, then:

$$\begin{aligned} x := 3 \mathbin{;} y := x^2 + 4 \mathbin{;} z := z \cdot y + x &= \langle x \mapsto_{\mathsf{s}} 3 \rangle \mathbin{;} \langle y \mapsto_{\mathsf{s}} x^2 + 4 \rangle \mathbin{;} \langle z \mapsto_{\mathsf{s}} z \cdot y + x \rangle && \text{[Definition]} \\ &= \langle [y \mapsto_{\mathsf{s}} x^2 + 4] \circ [x \mapsto_{\mathsf{s}} 3] \rangle \mathbin{;} \langle z \mapsto_{\mathsf{s}} z \cdot y + x \rangle && \text{[4.23]} \\ &= \langle (id \circ [x \mapsto_{\mathsf{s}} 3])(y \mapsto_{\mathsf{s}} [x \mapsto_{\mathsf{s}} 3] \dagger x^2 + 4 \rangle \mathbin{;} \langle z \mapsto_{\mathsf{s}} z \cdot y + x \rangle && \text{[SA8]} \\ &= \langle x \mapsto_{\mathsf{s}} 3, y \mapsto_{\mathsf{s}} 3^2 + 4 \rangle \mathbin{;} \langle z \mapsto_{\mathsf{s}} z \cdot y + x \rangle && \text{[4.16]} \\ &= \langle [z \mapsto_{\mathsf{s}} z \cdot y + x] \circ [x \mapsto_{\mathsf{s}} 3, y \mapsto_{\mathsf{s}} 9 + 4] \rangle && \text{[4.23]} \\ &= \langle x \mapsto_{\mathsf{s}} 3, y \mapsto_{\mathsf{s}} 13, z \mapsto z \cdot 13 + 3 \rangle && \square \end{aligned}$$

The result is a simultaneous assignment to the three variables, x, y, and z. The value of z depends on its initial value, which is unknown, and so the variable is retained.

We have shown in this section how lenses and our mechanisation of UTP support a relational program model that satisfies the laws of programming. In the next section we use them to derive operational semantics and verification calculi in Isabelle/UTP.

5. Automating Verification Calculi

In this section we demonstrate how the foundations established in §3 and §4 can be applied to automated program analysis. We show how concrete programs can be modelled, symbolically executed, and automatically verified using mechanically validated operational and axiomatic semantics, including Hoare logic and refinement calculus. Though the mechanisation of such calculi has been achieved several times before [46, 78, 63, 3], the novelty of our work is that lenses allow us to (1) express the meta-logical provisos of variables for the various calculi, (2) unify a variety of state space models, and (3) reason about aliasing of variables using independence and containment. We illustrate this with a number of standard examples, and a larger verification of a find-and-replace algorithm that utilises lenses in modelling arrays.

We consider the encoding of programs in Isabelle/UTP in §5.1, symbolic execution in §5.2, Hoare logic verification in §5.3, and finally refinement calculus in §5.4.

```
alphabet sfact = x::nat y::nat

definition pfact :: "nat ⇒ sfact hrel" where
  "pfact X =
    x := X ;; y := 1 ;;
    while x > 1 invr y * fact(x) = fact(X)
    do y := y * x ;; x := x - 1 od"
```

Figure 7: Factorial Program in Isabelle/UTP

5.1. Encoding Programs

Imperative programs can be encoded using the operators given in §4, and a concrete state space with the required variables. We can describe the factorial algorithm as shown in the following example.

Example 5.1 (Factorial Program)**.** We define a state space, $sfact \triangleq [x : \mathbb{N}, y : \mathbb{N}]$, consisting of two lenses x and y. We can then define a program for computing factorials as follows:

$$pfact : \mathbb{N} \to [sfact]\,\mathit{hrel}$$

$$pfact(X) \triangleq \left(\begin{array}{l} x := X\,;\ y := 1\,; \\ \textbf{\textit{while}}\ x > 1\ \textbf{\textit{do}} \\ \quad y := y * x\,;\ x := x - 1 \\ \textbf{\textit{od}} \end{array} \right)$$

Constant *pfact* is a function taking a natural number X as input and producing a program of type $[sfact]\,\mathit{hrel}$ that computes the factorial. The given value is assigned to UTP variable x, and 1 is assigned to y. The program iteratively multiplies y by x, and decrements x. In the final state, y has the factorial. □

This program can be entered into Isabelle/UTP using almost the same syntax as shown above. This is illustrated in Figure 7, where we use the ***alphabet*** command to create the state space, and then define the algorithm over this space using the ***definition*** command. Our parser automatically distinguishes whether a variable name is a lens, such as x, or logical variable, such as X, using type inference.

Although in this case we have manually constructed a state space, we can also pass the program variables as parameters to the program. Specifically, we can instead define

$$pfact(X : \mathbb{N}, x : \mathbb{N} \Longrightarrow \mathcal{S}, y : \mathbb{N} \Longrightarrow \mathcal{S}) \triangleq \cdots$$

which takes x and y as parameters, mimicking the pass-by-reference mechanism, and hence have a program of type $[\mathcal{S}]\,\mathit{hrel}$ that is polymorphic in the state space $\mathcal{S}$. This is possible because lenses model variables as first-class citizens with the specific implementation abstracted by the axioms. This is in contrast to other mechanisations where variables either must have explicit names [46, 78, 66] or are unstructured entities [26, 3].

As usual, the algorithm includes an invariant annotation [3] that will be explained shortly. We use this as a running example for the Isabelle/UTP symbolic execution and verification components.

5.2. Operational Semantics

Operational semantics are commonly assigned to a programming language by means of an inductive set of transition rules [63]. UTP [57, chapter 10], however, takes a different approach. As usual it characterises a small-step operational semantics for imperative programs using a step relation, $(s, P) \to (t, Q)$, meaning that program P, when started in state s, can transition to program Q, with state t. However, instead of characterising this relation using an inductive set, it gives a denotational semantics to the transition relation, and then proves transition rules as theorems. This has the benefit that linking operational and axiomatic semantics is straightforward.

Here, we adopt a similar idea to describe a reduction semantics for relational programs, and use it to perform symbolic execution, following the pattern given by Gordon and Collavizza [47]. We begin by describing the execution of a program P started in a state context Γ.

Definition 5.2 (Program Evaluation). $(\!|\Gamma, P|\!) \triangleq \langle \Gamma \rangle \mathbin{;} P$ where $P : [\mathcal{S}]\mathit{hrel}$ and $\Gamma : \mathcal{S} \to \mathcal{S}$.

The meaning of $(\!|\Gamma, P|\!)$ is that program P is executed in the variable context Γ, which gives assignments to a selection of variables in $\mathcal{S}$, for example $[x \mapsto_{\mathsf{s}} 5, y \mapsto_{\mathsf{s}} \mathit{true}]$. The expression $(\!|\Gamma, \mathit{II}|\!)$ denotes a program that has terminated in state Γ. The definition above of this operator is very simple, because of our encoding of variable contexts as substitutions: we apply Γ as an assignment, and compose this with the program P. Using this definition we can prove a set of reduction theorems.

Theorem 5.3 (Operational Reduction Rules).

$$\frac{}{(\!|\Gamma, P \mathbin{;} Q|\!) \to (\!|\Gamma, P|\!) \mathbin{;} Q} \qquad \frac{}{(\!|\Gamma, \mathit{II}|\!) \mathbin{;} P \to (\!|\Gamma, P|\!)} \qquad \frac{}{(\!|\Gamma, \langle \sigma \rangle|\!) \to (\!|\sigma \circ \Gamma, \mathit{II}|\!)}$$

$$\frac{(\Gamma \dagger b) \to \mathit{true}}{(\!|\Gamma, P \lhd b \rhd Q|\!) \to (\!|\Gamma, P|\!)} \qquad \frac{(\Gamma \dagger b) \to \mathit{false}}{(\!|\Gamma, P \lhd b \rhd Q|\!) \to (\!|\Gamma, Q|\!)}$$

$$\frac{(\Gamma \dagger b) \to \mathit{true}}{(\!|\Gamma, b \circledast P|\!) \to (\!|\Gamma, P \mathbin{;} (b \circledast P)|\!)} \qquad \frac{(\Gamma \dagger b) \to \mathit{false}}{(\!|\Gamma, b \circledast P|\!) \to (\!|\Gamma, \mathit{II}|\!)}$$

The arrow $P \to Q$ actually denotes an equality predicate $(P = Q)$; we use the arrow notation to aid in comprehension, and to emphasise the left-to-right nature of semantic evaluations. The first two rules on the top line handle sequential composition. The first pushes an evaluation into the first argument of a composition $P \mathbin{;} Q$. The second rule states that when the first argument of a sequential composition has terminated (II), execution moves on to the second argument (P). The third rule handles assignments, creating a new context by precomposing the assignment σ with the current context.

The second and third lines deal with conditional and while-loop iteration, respectively. In all these rules, the context Γ is applied to the condition b as a substitution. If the result is true, the conditional chooses the first branch P, and the while loop makes a copy of the loop body. If the result is false, the conditional chooses the second branch Q, and the while loop terminates.

Using these theorems, we can perform symbolic execution of programs in Isabelle/UTP. To achieve this we load the Isabelle simplifier with the equations of both Theorem 5.3 and Lemma 4.16, the latter of which allows us to apply substitutions. We also utilise the pointwise-lifted semantics given in §4.1 to evaluate expressions using the builtin HOL functional definitions and simplification laws.

In addition, we need the equations of Lemma 4.13 to evaluate and normalise substitutions. However, Law SB4, which allows reordering of subsitution maplets, is symmetric and therefore cannot be directly used as it would cause the simplifier to loop. The issue is that lenses do not *apriori* have a total order that can be used to reorder them. Nevertheless, Law SB4 is important to enable a canonical representation of concrete substitutions. Consequently, we extend the simplifier with a "simproc" [64], a specialised meta-logical simplification procedure that sorts substitution maplets lexicographically using the syntactic lens names. Thus, during symbolic evaluation the variable context will always order the variables lexicographically.

Applying the Isabelle simplifier with these laws, we can symbolically execute programs. Below, we give an example execution of the factorial program from Example 5.1. We do this by using the simplifier to evaluate the term $(\!|id, \mathit{pfact}(4)|\!)$, where id encodes an arbitrary initial assignment for all the variables.

Example 5.4 (Factorial Symbolic Execution)**.**

$$
\begin{aligned}
& (\!| id, pfact(4) |\!) \\
= \ & (\!| id, x := 4 \ ; \ y := 1 \ ; \ (x > 1) \circledast (y := y * x \ ; \ x := x - 1) |\!) \\
= \ & (\!| [x \mapsto_{\mathsf{s}} 4], y := 1 \ ; \ (x > 1) \circledast (y := y * x \ ; \ x := x - 1) |\!) \\
= \ & (\!| [x \mapsto_{\mathsf{s}} 4, y \mapsto_{\mathsf{s}} 1], (x > 1) \circledast (y := y * x \ ; \ x := x - 1) |\!) \\
= \ & (\!| [x \mapsto_{\mathsf{s}} 4, y \mapsto_{\mathsf{s}} 1], (y := y * x \ ; \ x := x - 1) \ ; \ (x > 1) \circledast (y := y * x \ ; \ x := x - 1) |\!) \\
= \ & (\!| [x \mapsto_{\mathsf{s}} 4, y \mapsto_{\mathsf{s}} 4], x := x - 1 \ ; \ (x > 1) \circledast (y := y * x \ ; \ x := x - 1) |\!) \\
= \ & (\!| [x \mapsto_{\mathsf{s}} 3, y \mapsto_{\mathsf{s}} 4], (x > 1) \circledast (y := y * x \ ; \ x := x - 1) |\!) \\
= \ & (\!| [x \mapsto_{\mathsf{s}} 2, y \mapsto_{\mathsf{s}} 12], (x > 1) \circledast (y := y * x \ ; \ x := x - 1) |\!) \\
= \ & (\!| [x \mapsto_{\mathsf{s}} 1, y \mapsto_{\mathsf{s}} 24], (x > 1) \circledast (y := y * x \ ; \ x := x - 1) |\!) \\
= \ & (\!| [x \mapsto_{\mathsf{s}} 1, y \mapsto_{\mathsf{s}} 24], \mathit{II} |\!)
\end{aligned}
$$

□

In this case, the program terminates in final state $[x \mapsto_{\mathsf{s}} 1, y \mapsto_{\mathsf{s}} 24]$, but in the case of a non-terminating program the simplifier will loop. Such a symbolic execution engine is a useful tool for simulation of programs.

Next, we show how we can mechanise the rules of Hoare logic and apply them to verification.

5.3. Hoare Logic

A Hoare triple $\{b\}\, P\, \{c\}$ is an assertion that, if program P is started in a state satisfying predicate b, then all final states satisfy predicate c. The UTP book [57, chapter 2] shows how this assertion can be encoded using a refinement statement. Their definition is mechanised in Isabelle/UTP as given below.

Definition 5.5 (Hoare Triple)**.** $\{b\}\, P\, \{c\} \triangleq ((b^{\blacktriangleleft} \Rightarrow c^{\blacktriangleright}) \sqsubseteq P)$

This states that the Hoare triple is valid when P is a refinement of the specification $b^{\blacktriangleleft} \Rightarrow c^{\blacktriangleright}$. The specification states that when b is satisfied initially, then c is satisfied finally. This is a statement of partial correctness since P could satisfy this condition by not terminating, for example if $P = \textbf{\textit{false}}$, since $Q \sqsubseteq \textbf{\textit{false}}$ for any Q. Definition 5.5 is equivalent to the standard definition for partial correctness [46] illustrated in Section 2.3:

Theorem 5.6. $\{b\}\, P\, \{c\} \Leftrightarrow (\forall (s, s') \bullet b(s) \wedge P(s, s') \Rightarrow c(s'))$

An equivalent characterisation is given in terms of our reduction relation below.

Theorem 5.7. *If* $\{b\}\, P\, \{c\}$ *then* $\forall (\Gamma_1, \Gamma_2) \bullet [(\Gamma_1 \dagger b) \wedge (\!| \Gamma_1, P |\!) \rightarrow (\!| \Gamma_2, \mathit{II} |\!) \Rightarrow (\Gamma_2 \dagger c)]$

This is an example of a UTP linking theorem [57] that relates two semantic presentations, in this case how the Hoare triple is related to the operational semantics. The satisfaction of $\{b\}\, P\, \{c\}$ implies that, for any initial state assignment Γ_1 satisfying precondition b, if P terminates with final state assignment Γ_2, then Γ_2 satisfies c. Thus we have formally related the operational and axiomatic semantics for relational programs. From Definition 5.5 we can also prove, as theorems, the following Hoare calculus laws:

Theorem 5.8 (Hoare Calculus Laws)**.**

$$\frac{[p \Rightarrow \sigma \dagger q]}{\{p\}\ \langle \sigma \rangle\ \{q\}} \qquad \frac{\{p\}\ Q_1\ \{s\} \quad \{s\}\ Q_2\ \{r\}}{\{p\}\ Q_1 \ ; \ Q_2\ \{r\}} \qquad \frac{\{b \wedge p\}\ S\ \{q\} \quad \{\neg b \wedge p\}\ T\ \{q\}}{\{p\}\ S \lhd b \rhd T\ \{q\}} \qquad \frac{\{p \wedge b\}\ S\ \{p\}}{\{p\}\ b \circledast P\ \{\neg b \wedge p\}}$$

The majority of these laws are standard [54]. However, the formulation of the assignment law is more general than the standard law, $\{p[e/x]\}\, x := e\, \{p\}$. It avoids the aliasing problem of classical Hoare logic since substitution depends on semantic independence of lenses, rather than syntactic inequality of variable names. We can model aliased names by giving a lens two (meta-logical) names in Isabelle, x and y. We can also consider the situation when x and y are different constructions and yet not independent, such as hp and $hp[0]$, from Example 2.3. We illustrate this with the following example.

```
lemma pfact_correct: "{true} pfact X {y = fact(X)}"
  unfolding pfact_def
  apply (hoare_auto)
  apply (metis diff_Suc_Suc diff_zero fact_diff_Suc less_SucI mult.assoc)
  apply (metis fact_0 fact_Suc_0 less_Suc0 linorder_neqE_nat mult.right_neutral)
  done
```

Figure 8: Verification of Factorial Algorithm

Example 5.9 (Aliasing)**.** We consider the triple $\{y = 3\}\, x := 2\, \{y = 3\}$. If we attempt its proof using the rules above, and the substitution laws (Lemma 4.16), the result is the predicate $(y = 3) \Rightarrow (y[2/x] = 3)$. If x and y are identical $(x = y)$ then this reduces to $(y = 3) \Rightarrow \mathit{false}$, which is not provable. If $x \neq y$, and yet either $x \approx y$, $x \preceq y$, or $y \preceq x$, then the conjecture is also not provable; indeed we can use nitpick [13, 12] to find a counterexample. If, however, $x \bowtie y$, then we obtain $(y = 3) \Rightarrow (y = 3)$, which is true. □

Similarly, from Definition 5.5 we can also represent derived laws, like the forward assignment law, by making use of unrestriction.

Lemma 5.10. $\{p\}\, x := e\, \{x = e \wedge p\}$ *if* $x \sharp e, x \sharp p$

We can use these laws to automatically verify our factorial program, following the pattern of previous works [2, 63, 3], though using our lens-based Hoare logic laws. We require, as usual, that the program is annotated with loop invariants. This is shown in Figure 7, with loop invariant $y * x! = X!$, which is syntactic sugar to help the proof strategy [3]. The proof strategy, implemented in the Isabelle/UTP tactic hoare-auto, executes the following steps:

1. Combine all composed assignments using Corollary 4.24.
2. Apply the Hoare logic laws of Theorem 5.8 as deduction rules.
3. Perform any substitutions in the resulting predicates using Lemma 4.13 and Lemma 4.16.
4. Apply the rel-auto tactic of §4.1 to each resulting predicate.

The result is a set of HOL predicates that characterise the verification conditions for the program. We exemplify this with the factorial program below.

Example 5.11. $\{\mathit{true}\}\, \mathit{pfact}(X)\, \{y = X!\}$

Proof. Application of hoare-auto yields the following verification conditions:

1. $x > 1 \wedge y * x! = X! \Rightarrow y * x * (x - 1)! = X!$
2. $x \leq 1 \wedge y * x! = X! \Rightarrow y = X!$

In this case, both proof obligations can be discharged using sledgehammer [12]. This is illustrated in Figure 8, with calls to metis for each proof obligation. □

As a further example we consider a program for the find and replace functionality in a text editor.

Example 5.12 (Find and Replace)**.**

$$
\begin{array}{l}
\textbf{\textit{function}}\ \mathit{find\text{-}replace}(arr : [string]array, len : nat, mtc : string, rpl : string) : nat \triangleq \\
\quad \textbf{\textit{var}}\ i, occ : nat \bullet \\
\quad i := 0\ ;\ occ := 0\ ; \\
\quad \textbf{\textit{while}}\ i < len\ \textbf{\textit{do}} \\
\quad\quad \textbf{\textit{if}}\ arr[i] = mtc\ \textbf{\textit{then}}\ arr[i] := rpl\ ;\ occ := occ + 1\ \textbf{\textit{else}}\ arr[i] := arr[i]\ \textbf{\textit{fi}}\ ; \\
\quad\quad i := i + 1 \\
\quad \textbf{\textit{od}}\ ; \\
\quad \textbf{\textit{return}}\ occ
\end{array}
$$

```
alphabet ss = arr::"nat ⇒ string" occ::"nat" i::"nat"

definition find_replace ::
  "(nat ⇒ string) ⇒ nat ⇒ string ⇒ string ⇒ ss hrel" where
"find_replace A len mtc rpl =
  arr := A ;; i := 0 ;; occ := 0 ;;
  while i < len
  invr arr = (λ k. if k < i ∧ A k = mtc then rpl else A k)
     ∧ occ = length (filter ((=) mtc) (map A [0..<i])) ∧ i ≤ len
  do
    if (arr i = mtc) then arr[i] := rpl ;; occ := occ + 1
    else arr[i] := arr i
    fi ;; i := i + 1
  od"
```

Figure 9: Find and Replace in Isabelle/UTP

In this program, we have an array of strings that represents a sequence of words in a text buffer, and we wish to replace every occurence of a particular string, *mtc*, with another string *rpl*. The specified function takes four parameters: an array of strings, *arr*, the length of the array *len*, and the two strings *mtc* and *rpl*. It returns the number of occurences for which it has made a replacement.

We first create local variables i and *occ*, using the ***var*** construct, and initialise them, for iterating over the list and recording occurrences. We then have a while loop that iterates over the length of the array. At each step, if the ith element of the array equals *mtc*, then this element is replaced with *rpl*, and the number of occurences is incremented. Otherwise, the element of the array is left unchanged. Finally, i is incremented and the loop continues. Once the loop exits, the value of *occ* is returned. Variables *occ* and i are modelled in Isabelle/UTP using record lenses, and the array *arr* using a total function lens.

The algorithm is mechanised in Isabelle/UTP as shown in Figure 9. We first create the state space *ss* with variables *arr*, *occ*, and i. We represent the variables *len*, *mtc*, and *rpl* as logical variables, since they are not changed by the algorithm. Moreover, we include an explicit parameter A that represents the initial state of the array. This is need to represent the invariant annotations, which we explain below.

There are at least two desirable properties for this program, both of which are postconditions: (1) *arr* should have all occurrences of *mtc* replaced with *rpl*; and (2) the value of *occ* returned should be number of occurrences in *arr* initially. The first can be represented by the following triple:

$$\{arr = A\}\ \mathit{find\text{-}replace}(arr, len, mtc, rpl)\ \{\forall\, i < len \bullet A[i] = (mtc \lhd arr[i] = rpl \rhd A[i])\}$$

This states that if the array is initially A, then following execution of *find-replace*, for every index i in A that has *mtc*, the array contains *rpl*, and all other elements remain the same. This can be proved using the invariants shown in Figure 9. It has two parts corresponding to properties (1) and (2). The first invariant states that at the ith iteration, *arr* is identical to A, except that every instance of *mtc* up to index i is replaced with *rpl*. The second invariant states that the number of occurrences is equal to the number of instances of *mtc* filtered out of the function before index i. The third invariant is that $i \leq len$. With these three invariants, we can verify the two properties; this is illustrated in Figure 10.

Thus we have shown how Isabelle/UTP can be applied to practical verification of relational programs, which can contain complex state space structures afforded by lenses. We can unify different variants of assignment, such as to multiple variables and collections, using the generalised assignment law. Moreover, the verification infrastructure is semantic and thus efficient in nature, without the need for deep syntax, and yet can utilise meta-logical concepts like substitution and unrestriction, which are not present in other shallow embeddings.

In the next section we consider the mechanisation of Morgan's refinement calculus.

5.4. Refinement Calculus

In this section, we show how lenses can support a Morgan-style refinement calculus [61], extending our previous results on frames in UTP [35]. The refinement calculus is a technique for stepwise development of

```
lemma find_replace_prop1:
  "{true} find_replace A len mtc rpl {∀ i < len. arr(i) = (rpl ◁ A(i) = mtc ▷ A(i)) }"
  unfolding find_replace_def by (hoare_auto, metis less_Suc_eq)

lemma find_replace_prop2:
  "{true} find_replace A len mtc rpl {occ = length (filter ((=) mtc) (map A [0..<len]))}"
  unfolding find_replace_def by (hoare_auto, metis less_Suc_eq)
```

Figure 10: Find and Replace verified in Isabelle/UTP

programs, by refining abstract specifications in increasingly concrete programs. The specification statement, $w{:}[pre, post]$, is a contract asserting that (1) only the variables in w may change, and (2) the postcondition ($post$) should be established when the precondition (pre) holds. We use lenses and our relational program model to mechanise a partial correctness specification statement, and prove several laws relating to frames. Ordinarily, Morgan's calculus specified total correctness, but here we focus on the laws relating to frames, rather than termination of programs. A total correctness calculus can be obtained straightforwardly by combining our results here and in Section 6 with the UTP theory of designs [57, 20].

For this, we first define a notion of equivalence modulo a lens that is needed to characterise frames.

Definition 5.13 (Observation Equivalence)**.** We assume $s_1, s_2 : \mathcal{S}$ and $X : \mathcal{V} \Longrightarrow \mathcal{S}$, and define:

$$s_1 \simeq_X s_2 \triangleq (s_1 = \mathit{put}_X\ s_2\ (\mathit{get}_X\ s_1))$$

This operator states that two states s_1 and s_2 are the same everywhere outside of the region described by X. For example, if we have three variables $x, y, z : \mathbb{Z}$, then we can show that

$$[x \mapsto 1, y \mapsto 2, z \mapsto 3] \simeq_x [x \mapsto 2, y \mapsto 2, z \mapsto 3]$$

since the two states differ only in the value of x. Lenses can represent sets of variables, using $+$, and so we can group several independent variables in a frame, as the following equivalences illustrate:

Lemma 5.14. *Assume $s_1, s_2 : \mathcal{S}$ then the following identities hold:*

$$(s_1 \simeq_{\mathbf{1}_\mathcal{S}} s_2) = \mathit{true} \text{ and } (s_1 \simeq_{\mathbf{0}_\mathcal{S}} s_2) = (s_1 = s_2)$$

This theorem demonstrates the intuition that **1** and **0** characterise the entirety and none of the state space, respectively. The state space outside of **1** is empty, and outside of **0** is the whole state. We can also show that $\simeq_X$ is an equivalence relation.

Lemma 5.15. *If X is a total lens then $\simeq_X$ is an equivalence relation on states.*

We now use observation equivalence to define the specification statement.

Definition 5.16 (Specification Statement)**.**

$$w{:}[pre, post] \triangleq ((pre^{\blacktriangleleft} \Rightarrow post^{\blacktriangleright}) \wedge \boldsymbol{v}^{\blacktriangleleft} \simeq_w \boldsymbol{v}^{\blacktriangleright})$$

A specification statement states that the precondition implies the postcondition, and in addition the initial state is the same as the final state, modulo w. The semantics is similar to the partial correctness Hoare triple in Definition 5.5, except that it only includes the specification. Indeed, we can prove the following well-known correspondence [57, 45].

Theorem 5.17. $\{pre\}\ P\ \{post\} \Leftrightarrow (\boldsymbol{v}{:}[pre, post] \sqsubseteq P)$

A Hoare triple can be re-expressed as a specification statement refinement. The standard triple does not include a frame, and consequently all variables are permitted to change, indicated by the frame being $\boldsymbol{v}$ (i.e. **1**). Moreover, from Definition 5.16 we can derive many of the refinement calculus laws [61] as theorems.

Theorem 5.18 (Refinement Calculus). *If x and w are total lenses, then the following laws hold:*

$$w{:}[pre, post] \sqsubseteq w{:}[pre, post'] \quad \textit{if } post' \Rightarrow post \tag{1}$$

$$w{:}[pre, post] \sqsubseteq w{:}[pre', post] \quad \textit{if } pre \Rightarrow pre' \tag{2}$$

$$x, w{:}[pre, post] \sqsubseteq w{:}[pre, post] \quad \textit{if } x \bowtie w \tag{3}$$

$$w{:}[pre, post] \sqsubseteq \mathit{II} \quad \textit{if } pre \Rightarrow post \tag{4}$$

$$w{:}[pre, post] \sqsubseteq x := e \quad \textit{if } x \preceq w, pre \Rightarrow post[e/x] \tag{5}$$

$$(w := e) = w{:}[\mathit{true}, w = e] \quad \textit{if } w \sharp e \tag{6}$$

$$w{:}[pre, post] \sqsubseteq w{:}[pre, mid] \; ; \; w{:}[mid, post] \tag{7}$$

$$w{:}[pre, post] \sqsubseteq w{:}[b \wedge pre, post] \lhd b \rhd w{:}[(\neg b) \wedge pre, post] \tag{8}$$

These laws use the lens operators as side conditions. Laws (1) and (2) allows strengthening and weakening of a post- and precondition, respectively. Law (3) shows that a variable can be removed from a frame in a refinement, because this effectively adds $x^{\blacktriangleright} = x^{\blacktriangleleft}$ as a conjunct. Laws (4) and (5) show the circumstances under which a specification may be refined to a skip (II) or assignment ($x := e$). For the latter, x must be in the frame w, represented by $x \preceq w$. Law (6) shows that an assignment $w := e$ can be written as a specification statement when e does not depend on w. Law (7) shows how a specification can be divided into two sequential specifications by inserting a midpoint condition. Finally, (8) shows how a conditional can be introduced, by conjoining the precondition with predicate b.

These results illustrate how lenses support modelling of frames. The theorems can be applied to derivation of algorithms by stepwise refinement in Isabelle/UTP. We have therefore shown in this section how our lens-based semantic foundation allows us to support a selection of verification calculi and linking theorems between them. In the next section we consider mechanisation of UTP theories.

6. Mechanising UTP Theories

In this section we apply Isabelle/UTP to the mechanisation of UTP theories. The relational program model in §4 is powerful, but not without limitations. It is well known, for instance, that simple relational programs, which capture only the possible initial and final values of program variables, cannot adequately differentiate terminating and non-terminating behaviour [21]. Furthermore, several programming paradigms, such as real-time, concurrency, and object orientation, require a richer semantic model with more observable information [21, 72, 74, 33].

This semantic enrichment can be facilitated by UTP theories. Semantic information is expressed by adding special observational variables, which encode quantities of a program or model, and invariants that restrict their domain, called healthiness conditions. The observational variables can be used to define specialised operators for a particular computational paradigm, such as a delay or deadline operator for a real-time language. For example, a clock variable, $clock : \mathbb{N}$ can be added to record to passage of time, or a trace variable $tr : [Event]list$ can be added to record events [21]. The healthiness conditions allow us to impose well-formedness invariants over these observational variables, and allow us to prove algebraic theorems that characterise the healthy elements. As usual, healthiness conditions are represented as idempotent predicate transformers, that is, functions on relations over the observational variables. The pay-off here is that general properties of elements of the UTP theory can often be reduced to properties of the healthiness conditions [57, 36].

We characterise UTP theories in Isabelle/UTP as follows.

Definition 6.1. An Isabelle/UTP theory is a pair $(\mathcal{S}, \mathcal{H})$, where $\mathcal{S}$ is an observation (i.e. state) space, $\mathcal{H} : [\mathcal{S}]\mathit{hrel} \to [\mathcal{S}]\mathit{hrel}$ is a healthiness condition, and $\mathcal{H}$ is idempotent: $\mathcal{H} \circ \mathcal{H} = \mathcal{H}$.

We mechanise this algebraic structure in Isabelle using locales [8]. $\mathcal{S}$ characterises the observations that can be made of the model. An observation space can be constructed using the ***alphabet*** command (§3.6), in which case it defines a set of lenses, $x_1 \cdots x_n$, which provide the observational variables. The healthiness conditions are encoded as total endofunctions on homogeneous relations parameterised by $\mathcal{S}$. If a theory has multiple healthiness conditions, then these can be composed function-wise. The simplest UTP theory is the relational theory, $Rel_\alpha \triangleq (\alpha, \lambda\, X \bullet X)$. Any type is an instance of α, and any relation is a fixed-point of $\lambda\, X \bullet X$.

We say that a relation $P : [\mathcal{S}]\mathit{hrel}$ is $\mathcal{H}$-healthy, written P *is* $\mathcal{H}$, when P is a fixed-point of $\mathcal{H}$: $\mathcal{H}(P) = P$. Moreover, we characterise the healthy elements of a theory by the set $[\![\mathcal{H}]\!]_{\mathsf{H}} \triangleq \{P \mid P \textit{ is } \mathcal{H}\}$. Idempotence of $\mathcal{H}$ guarantees the existence of at least one fixed point. This ensures that the UTP theory is non-empty since, for any relation P, $\mathcal{H}(P) \in [\![\mathcal{H}]\!]_{\mathsf{H}}$ since $\mathcal{H}(\mathcal{H}(P)) = \mathcal{H}(P)$. To illustrate, we formalise the UTP theory for timed relational programs introduced in §2.1.

Example 6.2 (Timed Relations)**.**
The parametric observation space $[S]\mathit{rt} \triangleq [\mathit{clock} : \mathbb{N}, \mathit{st} : S]$ has lenses $\mathit{clock} : \mathbb{N}$, which denotes the passage of time, and $\mathit{st} : S$, which denotes the user state. Healthiness function

$$\mathbf{HT}(P) \triangleq (\mathit{clock}^{\blacktriangleleft} \leq \mathit{clock}^{\blacktriangleright} \wedge P)$$

ensures that time advances. ***HT*** is clearly idempotent, since conjunction is idempotent. We define the following operator for introducing a delay for timed relations:

$$\mathbf{wait} : \mathbb{N} \to [[S]\mathit{rt}]\mathit{hrel}$$
$$\mathbf{wait}(n) \triangleq (\mathit{clock}^{\blacktriangleright} = \mathit{clock}^{\blacktriangleleft} + n \wedge \mathit{st}^{\blacktriangleright} = \mathit{st}^{\blacktriangleleft})$$

Here, *clock* is advanced by n, and the state is unchanged. We can prove that ***wait***(n) is ***HT***-healthy, since it advances time. Conversely, the predicate $\mathit{clock}^{\blacktriangleright} = \mathit{clock}^{\blacktriangleleft} - 1$ is not healthy, since it tries to reverse time. The mechanisation of the theory in Isabelle/UTP is shown in Figure 11. □

As in previous work [26, 79], our theories form "families": they characterise relations on several observation spaces, since the observation type is potentially polymorphic, and thus extensible with additional variables. For example, our theory in Example 6.2 is parametric in S. We can characterise a subtheory relation between UTP theories.

Definition 6.3. $([\beta]\mathcal{T}_2, \mathcal{H}_2)$ is a subtheory of $(\mathcal{S}_1, \mathcal{H}_1)$ when $[\beta]\mathcal{T}_2$ specialises $\mathcal{S}_1$, and $[\![\mathcal{H}_2]\!]_{\mathsf{H}} \subseteq [\![\mathcal{H}_1]\!]_{\mathsf{H}}$.

Subtheories allow us to arrange theories in a hierarchy with descendants that specialise the observation space with additional variables and constraints.

UTP theories also include a signature: a set of operators that construct healthy elements of the theory. We say that an operator

$$F : [\mathcal{S}]\mathit{hrel}^n \to [\mathcal{S}]\mathit{hrel}$$

in n parameters, is in the signature if $[\![\mathcal{H}]\!]_{\mathsf{H}}$ is closed under F. Formally, we require that:

$$\forall\, P_1, \cdots, P_n \bullet P_1 \textit{ is } \mathcal{H} \wedge \cdots \wedge P_n \textit{ is } \mathcal{H} \implies F(P_1, \cdots, P_n) \textit{ is } \mathcal{H}$$

The function F can either denote a new operator defined on $\mathcal{S}$, or an existing operator defined over a parent observation space. For example, nondeterministic choice $P \sqcap Q$ and sequential composition $P\ ;\ Q$ often inhabit the signature of several theories, since they are typed by arbitrary relations and so can be instantiated by any observation space. This also means that the corresponding algebraic laws for a parent operator F can be directly applied to elements of a new subtheory. For example, sequential composition is always associative, since every theory is a subtheory of Rel_α.

We exemplify this by giving the signature of our theory of timed relations.

Lemma 6.4. $[\![\mathbf{HT}]\!]_{\mathsf{H}}$ *is closed under the following relational operators:* $\mathbb{I}$, $;$, $\lhd\, b \,\rhd$, *and* $x := v$ *when* $x \bowtie \mathit{clock}$, *and so these are also within the theory signature, as demonstrated by the laws below:*

```
alphabet 's rt = clock :: nat  st :: 's

type_synonym 's time_rel = "'s rt hrel"

utp_def Wait :: "nat ⇒ 's time_rel" where "Wait(n) = ($clock´ = $clock + «n» ∧ $st´ = $st)"

utp_def HT :: "'s time_rel ⇒ 's time_rel" where "HT(P) = (P ∧ $clock ≤ $clock´)"

theorem HT_idem: "HT(HT(P)) = HT(P)" by rel_auto

theorem HT_mono: "P ⊑ Q ⟹ HT(P) ⊑ HT(Q)" by rel_auto

lemma HT_Wait: "HT(Wait(n)) = Wait(n)" by (rel_auto)

theorem Wait_Wait: "Wait(m) ;; Wait(n) = Wait (m + n)" by (rel_auto)
```

Figure 11: Timed Relations in Isabelle/UTP

$$\frac{}{\mathbb{I}\ \textit{is}\ \textbf{\textit{HT}}} \qquad \frac{P\ \textit{is}\ \textbf{\textit{HT}} \quad Q\ \textit{is}\ \textbf{\textit{HT}}}{(P\ ;\ Q)\ \textit{is}\ \textbf{\textit{HT}}} \qquad \frac{P\ \textit{is}\ \textbf{\textit{HT}} \quad Q\ \textit{is}\ \textbf{\textit{HT}}}{(P \lhd b \rhd Q)\ \textit{is}\ \textbf{\textit{HT}}} \qquad \frac{x \bowtie \mathit{clock}}{(x := v)\ \textit{is}\ \textbf{\textit{HT}}} \qquad \frac{}{\textbf{\textit{wait}}(n)\ \textit{is}\ \textbf{\textit{HT}}}$$

Once the signature operators have been established, the final step is to prove the characteristic algebraic theorems for them. These laws effectively provide an algebraic semantics for the UTP theory, and can be used to aid in the construction of program verification tools. There are three ways to obtain such theorems in Isabelle/UTP. Firstly, we can inherit them from a parent theory by utilising Definition 6.3. Secondly, we can import them from an algebraic structure by proving the algebra's axioms. Thirdly, we can prove them manually using Isabelle/UTP's proof tactics.

We exemplify the third approach for the new delay operator, using our rel-auto proof tactic (§4.1).

Theorem 6.5 (Delay Laws)**.**

$$\begin{aligned}
\textbf{\textit{wait}}(0) &= \mathbb{I} \\
\textbf{\textit{wait}}(m)\ ;\ \textbf{\textit{wait}}(n) &= \textbf{\textit{wait}}(m+n) \\
\textbf{\textit{wait}}(m)\ ;\ (P \lhd b \rhd Q) &= (\textbf{\textit{wait}}(m)\ ;\ P) \lhd b \rhd (\textbf{\textit{wait}}(m)\ ;\ Q) && \textit{if } \mathit{clock} \sharp b \\
\textbf{\textit{wait}}(m)\ ;\ x := v &= x := v\ ;\ \textbf{\textit{wait}}(m) && \textit{if } \mathit{clock} \sharp v, x \bowtie \mathit{clock}
\end{aligned}$$

Waiting for a zero length duration is simply a skip operation ($\mathbb{I}$), sequential composition of two delays, by m and n time units, is equivalent to a single $m+n$ delay. A delay distributes through a conditional provided that b does not refer to the *clock* variable. A delay commutes with an assignment $(x := v)$ provided that the delay expression does not depend on x and v does not depend on *clock*.

Algebraic theorems for UTP theories can also be obtained by linking to a variety of mechanised algebraic structures. The HOL-Algebra library [9], for example, characterises the axioms of partial orders, lattices, complete lattices, Galois connections, and the myriad of theorems that can be derived from them. A substantial advantage of the UTP approach is that algebraic theorems can often be reduced to proving properties of the underlying healthiness conditions, which allows us to obtain laws with minimal effort. We exemplify this by restricting ourselves to a particular subclass of continuous UTP theories.

Definition 6.6. $\mathcal{H}$ is continuous provided that $\mathcal{H}(\bigsqcap_{i \in A} P(i)) = \bigsqcap_{i \in A} \mathcal{H}(P(i))$ whenever $A \neq \emptyset$.

In a continuous theory, the healthiness condition distributes through arbitrary non-empty relational infima. A corollary of this definition is that $\mathcal{H}$ is also monotonic: $P \sqsubseteq Q \Rightarrow \mathcal{H}(P) \sqsubseteq \mathcal{H}(Q)$. So, by the Knaster-Tarski theorem, also part of HOL-Algebra, we can show that $[\![\mathcal{H}]\!]_H$ forms a complete lattice under refinement $\sqsubseteq$. This allows us to import theorems for recursive and iterative programs into a UTP theory.

From the induced complete lattice, we obtain the following operators: infimum $\bigsqcap_{\mathcal{T}}$, supremum $\bigsqcup_{\mathcal{T}}$, top element $\top_{\mathcal{T}}$, bottom element $\bot_{\mathcal{T}}$, and least fixed-point $\mu_{\mathcal{T}}$, which are all in the theory's signature, and for

which the usual complete lattice and fixed-point calculus theorems [7] hold. Moreover, in a continuous UTP theory, these operators can be calculated using the equational theorems given below.

Lemma 6.7 (Continuous Theory Properties)**.**

$$\begin{aligned}
\top_{\mathcal{T}} &= \mathcal{H}(\textbf{\textit{false}}) \\
\bot_{\mathcal{T}} &= \mathcal{H}(\textbf{\textit{true}}) \\
\textstyle\bigsqcap_{\mathcal{T}} A &= (\top_{\mathcal{T}} \lhd A = \emptyset \rhd \textstyle\bigsqcap A) && \text{if } A \subseteq [\![\mathcal{H}]\!]_{\mathsf{H}} \\
\mu_{\mathcal{T}}\ F &= \mu\, X \bullet F(\mathcal{H}(X)) && \text{if } F : [\![\mathcal{H}]\!]_{\mathsf{H}} \to [\![\mathcal{H}]\!]_{\mathsf{H}} \text{ and } F \text{ is monotonic}
\end{aligned}$$

These lemmas demonstrate the relationship between the theory operators, and the relational ones defined in §4. The theory top and bottom elements are the relational top and bottom with $\mathcal{H}$ applied. The theory infimum $\bigsqcap_{\mathcal{T}} A$ is $\top_{\mathcal{T}}$ when A is empty, and otherwise the relational infimum. Moreover, the least fixed-point operator can be expressed as the relational least fixed-point by precomposition of $\mathcal{H}$. The utility of these theorems is that we can construct nondeterministic choices and recursive programs using the relational operators $\sqcap$ and μ, which again allows reuse of their algebraic laws.

We now demonstrate how we can obtain such theorems for timed relations.

Theorem 6.8. ***HT*** *is continuous, since conjunction distributes through disjunction ($\bigsqcap$). We therefore obtain a complete lattice, and can calculate the top and bottom element:*

$$\top_{rt} = \textbf{\textit{HT}}(\textbf{\textit{false}}) = \textbf{\textit{false}} \qquad \bot_{rt} = \textbf{\textit{HT}}(\textbf{\textit{true}}) = (clock \leq clock')$$

We also obtain a least fixed-point operator for constructing and reasoning about iteration, μ_{rt}, and can use it to denote a timed loop operator: $b \circledast_{rt} P \triangleq (\mu_{rt}\, X \bullet P \mathbin{;} X \lhd b \rhd \mathbb{I})$.

From these theorems, we can proceed to define a timed Hoare calculus and proof tactics following the template given in §5. A concrete timed program can be modelled by creating a suitable state space, $vars \triangleq [x_1 : \tau_1 \cdots x_n : \tau_n]$, defining the program $P : [[vars]\textit{rt}]\textit{hrel}$ using the signature operators, and proving that P *is* ***HT***. Finally, the proven Hoare logic deduction rules can be used for program verification.

In this section, we have demonstrated how a UTP theory can be mechanically validated, a set of signature operators verified, and algebraic theorems for these operators proved. Though the example of timed relations is simple, it illustrates the basic concepts that we have used for more complex theories in Isabelle/UTP. Notably, we have applied our approach to the mechanisation of a hierarchy of UTP theories for reactive programs that uses five observational variables and seven healthiness conditions [36]. This UTP theory has been applied to the development of a verification tool for the *Circus* language [39], which extends our imperative programming language with concurrency and communication [67].

7. Related work

There have been several previous works on application of algebraic semantics to modelling mutation of state. Back and von Wright [6] use two functions $\mathsf{val}.x : \mathcal{S} \to \mathcal{V}$ and $\mathsf{set}.x : \mathcal{V} \to \mathcal{S} \to \mathcal{S}$, to characterise each variable, together with five axioms that characterise their behaviour.

Definition 7.1 (Back and von Wright's Variable Axioms [6, 4])**.**

$$\mathsf{val}.x.(\mathsf{set}.x.a.\sigma) = a \tag{9}$$

$$x \neq y \Rightarrow \mathsf{val}.y.(\mathsf{set}.x.a.\sigma) = \mathsf{val}.y.\sigma \tag{10}$$

$$\mathsf{set}.x.a \mathbin{;} \mathsf{set}.x.b = \mathsf{set}.x.b \tag{11}$$

$$x \neq y \Rightarrow \mathsf{set}.x.a \mathbin{;} \mathsf{set}.y.b = \mathsf{set}.y.b \mathbin{;} \mathsf{set}.x.a \tag{12}$$

$$\mathsf{set}.x.(\mathsf{val}.x.\sigma).\sigma = \sigma \tag{13}$$

The dot operator ($f.x$) denotes function application in Back and von Wright's work. From these axioms, and a predicate transformer semantics, they are able to prove the laws of programming [6], including Example 2.1. The lens functions *put* and *get* correspond to the variable functions set and val, and have almost the same axioms [30]. Specifically, the axioms of total lenses in Definition 3.5 correspond to Back and von Wright's axioms (9), (11), and (13) in Definition 7.1. The other two axioms, (12) and (10), are captured by lens independence (Definition 3.8). In fact, Theorem 3.11 demonstrates that only four of Back and von Wright's variable axioms are necessary [6]; axiom (10) can be proved from the others and so is redundant.

We have based our work on lenses rather than Back and von Wright's approach, which is also the reason for the different order in the parameters of *put* [32, 31]. Nevertheless, we effectively use lenses to relax Back and von Wright's axioms so that we can characterise, not only independence of variables, but also when one variable is "part of" another. Specifically, in Definition 7.1, x and y are individual variables, but with lenses they can equally refer to a set or hierarchical structure. Variable sets are also supported by Back and Preoteasa [4] by lifting these axioms over a list, for instance to support multiple assignment. Our approach has more generality since it allows hierarchy and does not require formalising variable names. Also similar to our work, Back and Preoteasa [4] use their set variable operator to characterise substitution.

J. Foster et al. [31] created lenses as an abstraction for bidirectional programming and solving the view-update problem in database theory [10], which seeks to propagate changes to a computed view table back to the original source table. Fischer et al. [30] give a detailed study of the algebraic laws for *get* and *put*, which is the starting point for our work. Foster et al. provide combinators for composing abstract lenses and concrete lenses based on trees [32, 31]. They have been practically applied in the *Boomerang* language[13] for transformations on textual data structures, such as XML databases. There is also a substantial Haskell library[14] that can be used to develop generic data type transformations. Pickering et al. [69] extend this library to support modular data accessors by introducing various additional combinators, including a form of parallel composition. Our lens sum operator (Definition 3.18) share similarity with the parallel composition operator defined by Pickering et al. [69]. It is different, however, because the latter acts on an explicitly separated state space of the form $A \times B$, whereas lens sum acts on an arbitrary state space.

Variables are also given an algebraic semantics by Dongol et al. [25], through the development of Cylindrical Kleene Algebra, that extends their previous work [3, 45]. The core of their approach is a cylindrification operator, $C_x\, R$, that liberates the variable x in relation R. Their work contains a comprehensive investigation of the algebraic properties of this operator in an algebraic program model. They apply this algebra to characterise assignment, substitution, and frames. They use a functional model of state [73], and thus need to fix types for both variables and values, as explained in Section 2.4. We hope in the future to apply their algebraic structures and generalise them in the context of lenses, building on our previous results [43, 35]. For comparison, relational cylindrification can effectively be implemented using our lens summation and quantifier operators as $C_x\, R \triangleq \exists\, (x^{\blacktriangleleft} + x^{\blacktriangleright}) \bullet R$.

8. Conclusions

In this paper we have given a comprehensive exposition of the foundations of our verification framework, Isabelle/UTP. As we have demonstrated, Isabelle/UTP provides a unified semantic foundation for a variety of computational paradigms and programming languages, with the ability to formulate machine-checked semantic models. These models can then be used in constructing verification tools that harness the powerful automated proof facilities of Isabelle/HOL.

We have described how variables can be modelled as algebraic objects using lenses. This allows us to unify and compose a variety of variable models, and define generic operators for their comparison and manipulation. The lens-based model, in particular, allows us to avoid dependence on syntactic concepts like naming, and instead build upon semantic properties like independence and containment.

We used this algebraic foundation to develop a flexible shallow embedding of UTP, including expressions, predicates, and relations. We provided proof tactics for relational conjectures, and a library of algebraic

[13] Boomerang home page: http://www.seas.upenn.edu/~harmony/
[14] Lenses, Folds and Traversals. https://hackage.haskell.org/package/lens

theorems. In contrast to previous work, our expression model additionally supports syntax-like manipulations, including substitution and unrestriction, without the need for explicit modelling of variable names. A particularly pleasing result is the semantic unification of assignment and substitution, which leads to several elegant algebraic laws. Our mechanisation therefore offers both efficient automated proof and expressivity.

We have used this relational model to develop programming operators, and prove the "laws of programming" [58] as theorems, along with the axioms of several algebraic structures, like complete lattices and cylindrical algebra. We have then used this body of laws in §5 to provide symbolic execution, Hoare logic verification, and refinement calculus components for relational programs. We show how lenses allow us to perform symbolic evaluation, reason about aliasing of different variables, and model frames and Morgan's specification statement [61]. Though we use UTP notation throughout, different syntactic flavours can easily be accommodated using Isabelle's powerful syntax processing facilities [77].

Finally we have described how the relational model can be applied to mechanised UTP theories. We have shown how UTP theories are represented, in terms of observation spaces and healthiness conditions. Our theory model supports reuse of algebraic laws by a notion of inheritance, whereby a UTP theory can be extended with additional observational quantities and refined by further healthiness conditions. We have shown how further laws can be obtained with links to algebraic structures.

In conclusion, we have made the first steps toward realising the UTP vision [57, Chapter 0] of integrated formal methods, underpinned by unifying semantics, with mechanised support. In many ways, this is only the beginning, as there remains a large number of computational paradigms that are not yet represented in Isabelle/UTP. A major item for future work is the mechanisation of the various pen-and-paper UTP theories that have been developed by the UTP community over the past twenty years [57, 20, 72, 74, 15]. We are currently working on various UTP theories to support verification of low-level code, hybrid programs [33, 62], control law block diagrams [38], and state machines [35]. Moreover, we also plan to analyse the efficiency of the various proof tactics described here and see if they scale to larger models.

With respect to verification of low-level code, one of the major features needed is a model of dynamically allocated memory addressed by pointers. We have hinted back in §3.2 that we can weaken the axioms of total lenses to describe "partial lenses". These are only defined for a subset of the possible states, which can be used to distinguish allocated and dangling pointers. We are currently using this idea to mechanise separation logic [71] in Isabelle/UTP, by combining lenses and separation algebra [19, 24, 34]. In tandem with this, we plan to study the algebraic structure of lenses in more depth and from the perspective of category theory, due to heterogeneous nature of the equivalence operator. In particular, we will explore the links between lenses and recent work on cylindrical Kleene algebra [25].

With respect to verification of hybrid dynamical systems, we have previously described an extension of the relational calculus with continuous variables [38, 37]. The generic trace model of reactive designs [37] and our extensible mechanisation of UTP theories means that we can specialise the reactive design hierarchy in a different direction to describe hybrid reactive systems, where the trace is a piecewise continuous function. This can be applied, for instance, to assign a UTP semantics to a language like Hybrid CSP [48], which integrates continuous evolution with discrete CSP-style events and concurrency. Proof support for such a language depends on the ability to reason about invariants of differential equations, and so we are also integrating Platzer's differential induction technique [70, 62], as employed by the KeYmaera tool[15], into Isabelle/UTP. Moreover, we have previously given a UTP semantics to the dynamical systems modelling language Modelica [38], and so we will also develop proof facilities for this.

Acknowledgements

We would like thank Prof. Burkhart Wolff for his invaluable feedback on our work, and for first pointing us in the direction of lenses as a possible research direction. We also thank the anonymous reviewers of our paper, whose thorough and constructive comments on our work have greatly enhanced its quality.

[15]KeYmaera: http://symbolaris.com/info/KeYmaera.html

This work is funded by the EPSRC projects CyPhyAssure[16] (Grant EP/S001190/1), RoboCalc[17] (Grant EP/M025756/1), RoboTest (EP/R025479/1), and the Royal Academy of Engineering.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
[2] E. Alkassar, M. Hillebrand, D. Leinenbach, N. Schirmer, and A. Starostin. The Verisoft approach to systems verification. In *VSTTE 2008*, volume 5295 of *LNCS*, pages 209–224. Springer, 2008.
[3] A. Armstrong, V. Gomes, and G. Struth. Building program construction and verification tools from algebraic principles. *Formal Aspects of Computing*, 28(2):265–293, 2015.
[4] R.-J. Back and V. Preoteasa. An algebraic treatment of procedure refinement to support mechanical verification. *Formal Aspects of Computing*, 17(1), 2005.
[5] R.-J. Back and J. von Wright. Refinement concepts formalised in higher order logic. *Formal Aspects of Computing*, 2(3):247–272, 1990.
[6] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.
[7] R. Backhouse, C. Aarts, B. Boiten, H. Doombos, et al. Fixed-point calculus. *Information Processing Letters*, 53:131–136, 1995.
[8] C. Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In *Proc. 5th Intl. Conf. on Mathematical Knowledge Management (MKM)*, volume 4108 of *LNCS*, pages 31–43. Springer, 2006.
[9] C. Ballarin et al. The Isabelle/HOL Algebra Library. Technical report, Isabelle/HOL Library, 2018. https://isabelle.in.tum.de/library/HOL/HOL-Algebra/document.pdf.
[10] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.
[11] M. Barnett, B.-Y. Chang, R. DeLine, B. Jacobs, and R. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, LNCS. Springer, 2005.
[12] J. C. Blanchette, L. Bulwahn, and T. Nipkow. Automatic proof and disproof in Isabelle/HOL. In *FroCoS*, volume 6989 of *LNCS*, pages 12–27. Springer, 2011.
[13] J. C. Blanchette and T. Nipkow. A counterexample generator for higher-order logic based on a relational model finder. In *Interactive Theorem Proving*, volume 6172 of *LNCS*, pages 131–146. Springer, 2010.
[14] Richard J. Boulton, Andrew D. Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In Victoria Stavridou, Thomas F. Melham, and Raymond T. Boute, editors, *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, Nijmegen, 22–24 June 1992*, volume A-10 of *IFIP Transactions*, pages 129–156. North-Holland, 1992.
[15] R. Bresciani and A. Butterfield. A UTP semantics of pGCL as a homogeneous relation. In *IFM*, volume 7321 of *LNCS*, pages 191–205. Springer, 2012.
[16] M. Broy and O. Slotosch. Enriching the software development process by formal methods. In *Applied Formal Methods – FM-Trends 98*, volume 1641 of *LNCS*, pages 44–61. Springer, 1998.
[17] A. Buttefield. Saoithin: A theorem prover for UTP. In *UTP*, volume 6445 of *LNCS*, pages 137–156. Springer, 2010.
[18] A. Butterfield. The logic of $U \cdot (TP)^2$. In *UTP*, volume 7681 of *LNCS*, pages 124–143. Springer, 2012.
[19] C. Calcagno, P. O'Hearn, and Y. Hongseok. Local action and abstract separation logic. In *LICS*. IEEE, 2007.
[20] A. Cavalcanti and J. Woodcock. A tutorial introduction to designs in unifying theories of programming. In *Proc. 4th Intl. Conf. on Integrated Formal Methods (IFM)*, volume 2999 of *LNCS*, pages 40–66. Springer, 2004.
[21] A. Cavalcanti and J. Woodcock. A tutorial introduction to CSP in unifying theories of programming. In *PSSE*, volume 3167 of *LNCS*, pages 220–268. Springer, 2006.
[22] R. J. Colvin, I. J. Hayes, and L. A. Meinicke. Designing a semantic model for a wide-spectrum language with concurrency. *Formal Aspects of Computing*, 29, 2017.
[23] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.
[24] B. Dongol, V. Gomes, and G. Struth. A program construction and verification tool for separation logic. In *MPC 2015*, volume 9129 of *LNCS*, pages 137–158. Springer, 2015.
[25] B. Dongol, I. Hayes, L. Meinicke, and G. Struth. Cylindric Kleene lattices for program construction. In *MPC*, volume 11825 of *LNCS*, pages 192–225. Springer, October 2019.
[26] A. Feliachi, M.-C. Gaudel, and B. Wolff. Unifying theories in Isabelle/HOL. In *UTP 2010*, volume 6445 of *LNCS*, pages 188–206. Springer, 2010.
[27] A. Feliachi, M.-C. Gaudel, and B. Wolff. Isabelle/Circus: a process specification and verification environment. In *VSTTE 2012*, volume 7152 of *LNCS*, pages 243–260. Springer, 2012.
[28] A. Feliachi, M.-C. Gaudel, and B. Wolff. Symbolic test-generation in HOL-TESTGEN/CirTA: A case study. *Intl J. Software and Informatics*, 9(2):122–203, 2015.

[16] CyPhyAssure Project: https://www.cs.york.ac.uk/circus/CyPhyAssure/
[17] RoboCalc Project: https://www.cs.york.ac.uk/circus/RoboCalc/

[29] J.-C. Filliâtre and A. Paskevich. Why3 – where programs meet provers. In *Programming Languages and Systems (ESOP)*, volume 7792 of *LNCS*. Springer, 2013.
[30] S. Fischer, Z. Hu, and H. Pacheco. A clear picture of lens laws. In *MPC 2015*, pages 215–223. Springer, 2015.
[31] J. Foster. *Bidirectional programming languages*. PhD thesis, University of Pennsylvania, 2009.
[32] J. Foster, M. Greenwald, J. Moore, B. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007.
[33] S. Foster. Hybrid relations in Isabelle/UTP. In *UTP*, volume 11885 of *LNCS*, pages 130–153. Springer, 2019.
[34] S. Foster and J. Baxter. Automated algebraic reasoning for collections and local variables with lenses. In M. Winter, editor, *Proc. 18th Intl. Conf. on Relational and Algebraic Methods in Computer Science (RAMiCS)*, volume 12062 of *LNCS*. Springer, April 2020.
[35] S. Foster, J. Baxter, A. Cavalcanti, A. Miyazawa, and J. Woodcock. Automating verification of state machines with reactive designs and Isabelle/UTP. In *Proc. 15th. Intl. Conf. on Formal Aspects of Component Software*, volume 11222 of *LNCS*. Springer, October 2018.
[36] S. Foster, A. Cavalcanti, S. Canham, J. Woodcock, and F. Zeyda. Unifying theories of reactive design contracts. *Theoretical Computer Science*, 802, September 2020.
[37] S. Foster, A. Cavalcanti, J. Woodcock, and F. Zeyda. Unifying theories of time with generalised reactive processes. *Information Processing Letters*, 135:47–52, 2018.
[38] S. Foster, B. Thiele, A. Cavalcanti, and J. Woodcock. Towards a UTP semantics for Modelica. In *UTP*, volume 10134 of *LNCS*, pages 44–64. Springer, 2016.
[39] S. Foster, K. Ye, A. Cavalcanti, and J. Woodcock. Calculational verification of reactive programs with reactive relations and Kleene algebra. In *Proc. 17th Intl. Conf. on Relational and Algebraic Methods in Computer Science (RAMICS)*, volume 11194 of *LNCS*. Springer, October 2018.
[40] S. Foster and F. Zeyda. Optics in Isabelle/HOL. *Archive of Formal Proofs*, August 2018. https://www.isa-afp.org/entries/Optics.html.
[41] S. Foster, F. Zeyda, Y. Nemouchi, P. Ribeiro, and B. Wolff. Isabelle/UTP: Mechanised Theory Engineering for Unifying Theories of Programming. *Archive of Formal Proofs*, 2019. https://www.isa-afp.org/entries/UTP.html.
[42] S. Foster, F. Zeyda, and J. Woodcock. Isabelle/UTP: A mechanised theory engineering framework. In *UTP*, volume 8963 of *LNCS*, pages 21–41. Springer, 2014.
[43] S. Foster, F. Zeyda, and J. Woodcock. Unifying heterogeneous state-spaces with lenses. In *ICTAC*, volume 9965 of *LNCS*. Springer, 2016.
[44] A. J. Galloway and B. Stoddart. Integrated formal methods. In *Proc. INFORSID*. INFORSID, 1997.
[45] V. B. F. Gomes and G. Struth. Modal Kleene algebra applied to program correctness. In *21st. Intl. Symp. on Formal Methods (FM)*, volume 9995 of *LNCS*. Springer, 2016.
[46] M. J. C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current trends in hardware verification and automated theorem proving*, pages 387–439. Springer, May 1989.
[47] Mike Gordon and Hélène Collavizza. Forward with Hoare. In A. W. Roscoe, Clifford B. Jones, and Kenneth R. Wood, editors, *Reflections on the Work of C. A. R. Hoare*, pages 101–121. Springer, 2010.
[48] J. He. From CSP to hybrid systems. In A. W. Roscoe, editor, *A classical mind: essays in honour of C. A. R. Hoare*, pages 171–189. Prentice Hall, 1994.
[49] E. C. R. Hehner. Predicative programming. *Communications of the ACM*, 27(2):134–151, 1984.
[50] E. C. R. Hehner. A practical theory of programming. *Science of Computer Programming*, 14:133–158, 1990.
[51] E. C. R. Hehner. *A Practical Theory of Programming*. Springer, 1993.
[52] E. C. R. Hehner and A. J. Malton. Termination conventions and comparative semantics. *Acta Informatica*, 25, 1988.
[53] L. Henkin, J. Monk, and A. Tarski. *Cylindric Algebras, Part I*. North-Holland, 1971.
[54] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
[55] C. A. R. Hoare. Programs are predicates. *Philisophical Transactions of the Royal Society*, 312(1522), October 1984.
[56] C. A. R. Hoare. Unified theories of programming. Technical report, Oxford Computing Laboratory, July 1994.
[57] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
[58] T. Hoare, I. Hayes, J. He, C. Morgan, A. Roscoe, J. Sanders, I. Sørensen, J. Spivey, and B. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–687, August 1987.
[59] M. Hofmann, B. Pierce, and D. Wagner. Symmetric lenses. In *Proc. 38th Intl. Symp. on Principles of Programming Languages (POPL)*, pages 371–384. IEEE, 2011.
[60] B. Huffman and O. Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *CPP*, volume 8307 of *LNCS*, pages 131–146. Springer, 2013.
[61] C. Morgan. *Programming from Specifications*. Prentice-Hall, January 1996.
[62] J. H. Y. Munive, G. Struth, and S. Foster. Differential Hoare logics and refinement calculi for hybrid systems with Isabelle/HOL. In *18th Intl. Conf on Relational and Algebraic Methods in Computer Science (RAMiCS)*, volume 12062 of *LNCS*. Springer, April 2020.
[63] T. Nipkow and G. Klein. *Concrete Semantics with Isabelle/HOL*. Springer, December 2014.
[64] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
[65] G. Nuka and J. Woodcock. Mechanising a unifying theory. In *UTP*, volume 4010 of *LNCS*, pages 217–235. Springer, 2006.
[66] M. Oliveira, A. Cavalcanti, and J. Woodcock. Unifying theories in ProofPower-Z. In *UTP 2006*, volume 4010 of *LNCS*, pages 123–140. Springer, 2007.
[67] M. Oliveira, A. Cavalcanti, and J. Woodcock. A UTP semantics for Circus. *Formal Aspects of Computing*, 21:3–32, 2009.

[68] R. F. Paige. A meta-method for formal method integration. In *Proc. 4th. Intl. Symp. on Formal Methods Europe (FME)*, volume 1313 of *LNCS*, pages 473–494. Springer, 1997.
[69] M. Pickering, J. Gibbons, and N. Wu. Profunctor optics: Modular data accessors. *The Art, Science, and Engineering of Programming*, 1(2), 2017.
[70] A. Platzer. Differential dynamic logic for hybrid systems. *Journal of Automated Reasoning*, 41(2):143–189, 2008.
[71] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*. IEEE, 2002.
[72] T. Santos, A. Cavalcanti, and A. Sampaio. Object-Orientation in the UTP. In *UTP 2006*, volume 4010 of *LNCS*, pages 20–38. Springer, 2006.
[73] N. Schirmer and M. Wenzel. State spaces – the locale way. In *SSV 2009*, volume 254 of *ENTCS*, pages 161–179, 2009.
[74] A. Sherif, A. Cavalcanti, J. He, and A. Sampaio. A process algebraic framework for specification and validation of real-time systems. *Formal Aspects of Computing*, 22(2):153–191, 2010.
[75] M. Spivey. *The Z-Notation - A Reference Manual*. Prentice Hall, Englewood Cliffs, N. J., 1989.
[76] A. Tarski. On the calculus of relations. *J. Symbolic Logic*, 6(3):73–89, 1941.
[77] F. Tuong and B. Wolff. Deeply integrating C11 code support into Isabelle/PIDE. In *Formal Integrated Development Environment (F-IDE)*, volume 310 of *EPTCS*, pages 13–28, 2019.
[78] J. von Wright. Program refinement by theorem prover. In *Proc. 6th Refinement Workshop*, pages 121–150. Springer, January 1994.
[79] F. Zeyda and A. Cavalcanti. Mechanical reasoning about families of UTP theories. *Science of Computer Programming*, 77(4):444–479, April 2012.
[80] F. Zeyda, S. Foster, and L. Freitas. An axiomatic value model for Isabelle/UTP. In *UTP*, volume 10134 of *LNCS*, pages 155–175. Springer, 2016.

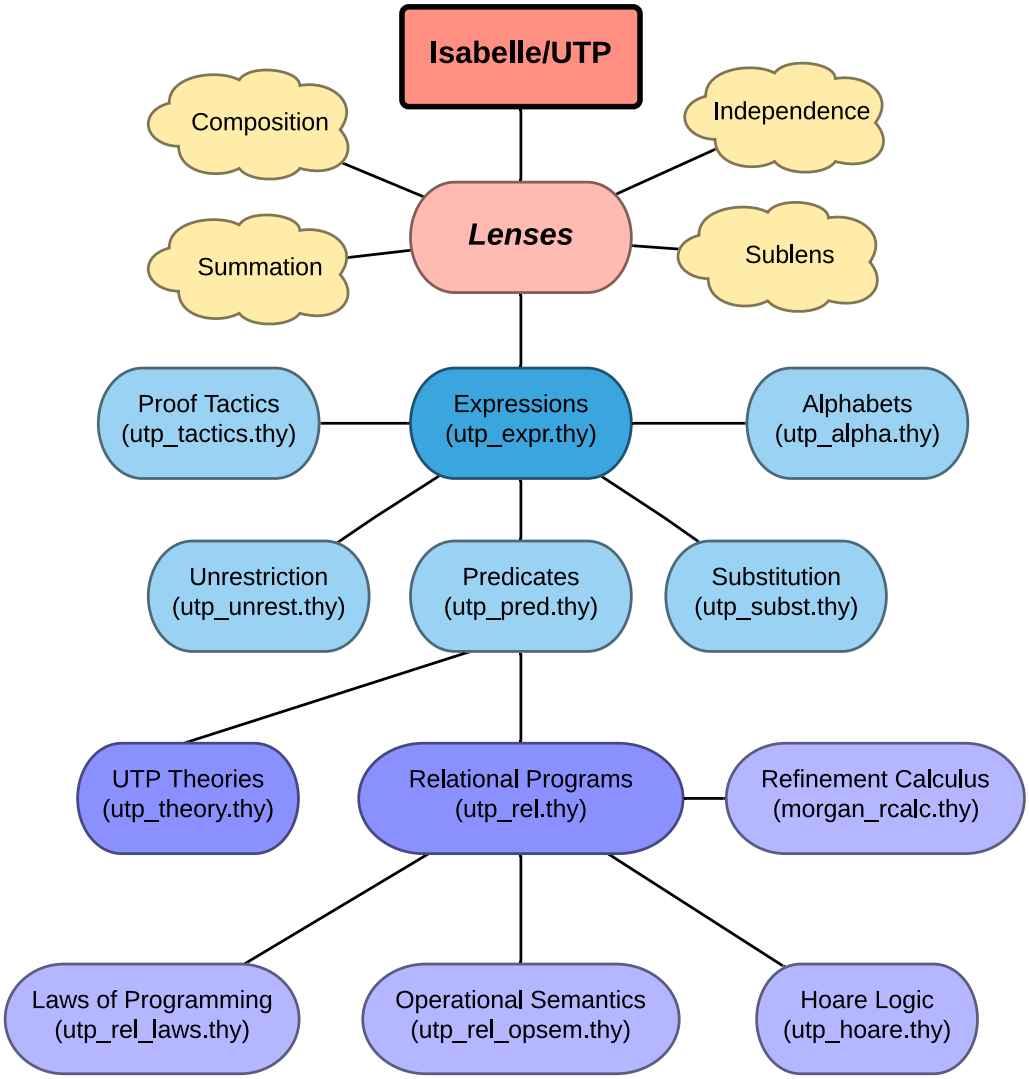

Figure 12: Main Isabelle/UTP Concepts and Theories