Unifying heterogeneous state-spaces with lenses

Simon Foster[1], Frank Zeyda[2], and Jim Woodcock[1]
{simon.foster,jim.woodcock}@york.ac.uk f.zeyda@tees.ac.uk

[1] University of York, Department of Computer Science, York, YO10 5GH, UK.
[2] Teesside University, School of Computing, Middlesbrough, TS1 3BA, UK.

Abstract. Most verification approaches embed a model of program state into their semantic treatment. Though a variety of heterogeneous state-space models exists, they all possess common theoretical properties one would like to abstractly capture, such as the laws of programming. In this paper, we propose lenses as a universal state-space modelling solution. Lenses provide an abstract interface for manipulating data types through spatially-separated views. We define a lens algebra that allow their composition and comparison. We show how this algebra can be used to model variables and alphabets in Hoare and He's Unifying Theories of Programming. The combination of lenses and relational algebra gives rise to a model for UTP in which its fundamental laws can be verified. Moreover, we illustrate how they can be used to model more state complex notions such as memory stores and parallel states. We provide a mechanisation in Isabelle/HOL that validates our theory, and facilitates its use in program verification.

1 Introduction

Predicative programming [16] is a unification technique that uses predicates to describe abstract program behaviour and executable code alike. Programs are denoted as logical predicates that characterise the observable behaviours as mappings between the state before and after execution. Thus one can apply predicate calculus to reason about programs, as well as prove the algebraic laws of programming themselves [19]. These laws can then be applied to construct semantic presentations for the purpose of verification, such as operational semantics, Hoare calculi, separation logic, and refinement calculi, to name a few [2,9]. This further enables the application of automated theorem provers to build program verification tools, an approach which has seen multiple successes [1,22].

Modelling the state space of a program and manipulation of its variables is a key problem to be solved when building verification tools [27]. Whilst relation algebra, Kleene algebra, quantales, and related algebraic structures provide excellent models for point-free laws of programming [14,4], when one considers point-wise laws for operators that manipulate state, like assignment, additional behavioural semantics is needed. State spaces can be heterogeneous, that is consisting of different models of state and variables. For example, separation logic [26] considers both the store, a static mapping from names to values, and

the heap, a dynamic mapping from addresses to values. Nevertheless one would like a uniform interface for the different variable models to facilitate generic laws of programming. When considering parallel programs [20], one also needs to consider division of the state space into non-interfering regions for concurrent threads, and their eventual reconciliation post execution. Moreover, we have the overarching need for meta-logical operators on state, like variable substitution and freshness, that are often considered informally but are vital to express and mechanise many laws of programming [19,16,20].

We thus propose *lenses* [12] as a unifying solution to state-space modelling. Lenses are a proposed solution to the view-update problem in data base theory [13], and are similarly applied to manipulation of data structures in functional programming [11]. They employ well-behaved *get* and *put* functions to identify a particular view of a source data structure, and allow one to perform transformations on it independently of the wider context.

Our contribution is an extension of the theory of lenses to allow their use in modelling variables as abstract views on the state space with a uniform semantic interface. We define a novel lens algebra for manipulation of variables and state spaces, including separation algebra style operators [7] such as state (de)composition, that enable abstract reasoning. Our algebra has been mechanised in Isabelle/HOL including a repository of verified lens laws. We apply the lens algebra to model heterogeneous state space models within the context of Hoare and He's Unifying Theories of Programming [20,8] (UTP), a predicative programming framework with a modular approach to denotational model construction. We use lenses to semantically model UTP variables and the predicate calculus' meta-logical functions, with no need for explicit abstract syntax, and thence provide a purely algebraic basis for the meta-logical laws, predicate calculus laws, and the laws of programming. We have thence further used Isabelle/HOL [23] to mechanise a large repository of such laws, which both validates the soundness of our model and also allows their application to building program verification tools[3].

The structure of our paper is as follows. In §2, we provide background material and related work. In §3, we provide a mechanised theory of lenses, in the form of an algebraic hierarchy, concrete instantiations, and algebraic operators, including a useful equivalence relation. This theory is standalone, and we believe has further applications beyond modelling state. Crucially, all the constructions we describe require only a first-order polymorphic type system which makes it suitable for Isabelle/HOL. In §4, we apply the theory of lenses to show how different state abstractions can be given a unified treatment. In this context, we construct the UTP's relational calculus, associated meta-logical operators, and prove various laws of programming. Along the way we show how our model satisfies various important algebraic structures to validate its adequacy. We also use lenses to give an account to parallel state in §4.4. Finally, in §5, we conclude.

[3] For supporting Isabelle theories, including mechanised proofs for all laws in this paper, see `http://cs.york.ac.uk/~simonf/ictac2016`

$$
\begin{array}{rcl@{\qquad}rcl}
x := v & \triangleq & x' = v \wedge y' = y & P \,;\, Q & \triangleq & \exists\, x_0 \bullet P[x_0/x'] \wedge Q[x_0/x] \\
(P \lhd b \rhd Q) & \triangleq & (b \wedge P) \vee (\neg b \wedge Q) & P^{*} & \triangleq & \nu X \bullet P \,;\, X
\end{array}
$$

Table 1. Imperative programming in the alphabetised relational calculus

2 Background and related work

2.1 Unifying Theories of Programming

The UTP [20] is a framework for defining denotational semantic models based on an alphabetised predicate calculus. A program is denoted as a set of possible observations. In the relational calculus, imperative programs are in view and thus observations consist of before variables x and after variables x'. This allows operators like assignment, sequential composition, if-then-else, and iteration to be denoted as predicates over these variables, as illustrated in Table 1. From these denotations, algebraic laws of programming can be proved, such as those in Table 2, and more specialised semantic models developed for reasoning about programs, such as Hoare calculi and operational semantics. UTP also supports more sophisticated modelling constructs; for example concurrency is treated in [20, Chapter 7] via the *parallel-by-merge* construct $P \,||_M\, Q$, a general scheme for parallel composition that creates two copies of the state-space, executes P and Q in parallel on them, and then merges the results through the merge predicate M. This is then applied to UTP theory of communiation in Chapter 8, and henceforth to give a UTP semantics to the process calculus CSP [8,18].

Mechanisation of the UTP for the purpose of verification necessitates a model for the predicate and relational calculi [15,28] that must satisfy such laws. LP1 and LP2 are point-free laws, and can readily be derived from algebras like relation algebra or Kleene algebra [14]. The remaining laws, however, are point-wise in the sense that they rely on the predicate variables. Whilst law LP3 can be modelled with KAT [2] (Kleene Algebra with Tests) by considering b to be a test, the rest explicitly reference variables. LP4 and LP5 require that we support quantifiers and substitution. LP6 additionally requires we can specify free variables. Thus, to truly provide a generic algebraic foundation for the UTP, a more expressive model supporting these operators is needed.

$$
\begin{array}{rcll}
(P \,;\, Q) \,;\, R & = & P \,;\, (Q \,;\, R) & & \text{(LP1)} \\
P \,;\, \mathbf{false} & = & \mathbf{false} \,;\, P \;=\; \mathbf{false} & & \text{(LP2)} \\
\mathbf{while}\, b\, \mathbf{do}\, P & = & (P \,;\, \mathbf{while}\, b\, \mathbf{do}\, P) \lhd b \rhd \mathbb{I} & \text{if } \forall\, x \bullet x' \notin \mathsf{fv}(b) & \text{(LP3)} \\
P \,;\, Q & = & \exists\, x_0 \bullet P[x_0/x] \,;\, Q[x_0/x'] & & \text{(LP4)} \\
x := e \,;\, P & = & P[e/x] & & \text{(LP5)} \\
(x := e \,;\, y := f) & = & (y := f \,;\, x := e) & \text{if } x \neq y, x \notin \mathsf{fv}(f), y \notin \mathsf{fv}(e) & \text{(LP6)}
\end{array}
$$

Table 2. Typical laws of programming

2.2 Isabelle/HOL

Isabelle/HOL [23] is a proof assistant for Higher Order Logic. It includes a functional specification language, a proof language for discharging specified goals in terms of proven theorems, and tactics that help automate proof. Its type system supports first-order parametric polymorphism, meaning types can carry variables – e.g. $\alpha\,\mathsf{list}$ for type variable α. Built-in types include total functions $\alpha \Rightarrow \beta$, tuples $\alpha \times \beta$, booleans bool, and natural numbers nat. Isabelle also includes partial function maps $\alpha \rightharpoonup \beta$, which are represented as $\alpha \Rightarrow \beta\,\mathsf{option}$, where $\beta\,\mathsf{option}$ can either take the value $\mathsf{Some}\,(v : \beta)$ or None. Function $\mathrm{dom}(f)$ gives the domain of f, and function $\mathsf{the} : \alpha\,\mathsf{option} \Rightarrow \alpha$ extracts the valuation from a Some constructor, or returns an underdetermined value if None is present.

Record types can be created using $\textbf{record}\,\mathcal{R} = f_1 : \tau_1 \cdots f_n : \tau_n$, where $f_i : \tau_i$ is a field. Each field f_i yields a query function $f_i : \mathcal{R} \Rightarrow \tau_i$, and update function $f_i\text{-}upd : (\tau_i \Rightarrow \tau_i) \Rightarrow (\mathcal{R} \Rightarrow \mathcal{R})$ with which to transform $\mathcal{R}$. Moreover Isabelle provides simplification theorems for record instances $(\!|\, f_1 = v_1 \cdots f_n = v_n \,|\!)$:

$$f_i(\!|\, \cdots f_i \ = v \cdots \,|\!) = v \qquad\qquad f_i\text{-}upd\, g\,(\!|\, \cdots f_i = v \cdots \,|\!) = (\!|\, \cdots f_i = g(v) \cdots \,|\!)$$

The HOL logic includes an equality relation $_ = _ : \alpha \Rightarrow \alpha \Rightarrow \mathsf{bool}$ that equates values of the same type α. In terms of tactics, Isabelle provides an equational simplifier simp, generalised deduction tactics blast and auto, and integration of external automated provers using the $\mathsf{sledgehammer}$ tool [6].

Our paper does not rely on detailed knowledge of Isabelle, as we present our definitions and theorems mathematically, though with an Isabelle feel. Technically, we make heavy use of the $\mathsf{lifting}$ and $\mathsf{transfer}$ packages [21] that allow us to lift definitions and associated theorems from super-types to sub-types. We also make use of Isabelle's **locale** mechanism to model algebraic hiearchies as in [14].

2.3 Mechanised state-spaces

Several mechanisations of the UTP in Isabelle exist [10,15,28] that take a variety of approaches to modelling state; for a detailed survey see [28]. A general comparison of approaches to modelling state was made in [27] which identifies four models of state, namely state as functions, tuples, records, and abstract types, of which the first and third seem the most prevalent.

The first approach models state as a function $\mathsf{Var} \Rightarrow \mathsf{Val}$, for suitable types to model values and variables. This approach is taken by [24,15,9,28], and requires a deep model of variables and values, in which concepts such as typing are first-class. This provides a highly expressive model with few limitations on possible manipulations [15]. However, [27] highlights two obstacles: (1) the machinery required for deep reasoning about program values is heavy and *a priori* limits possible constructions, and (2) explicit naming of variables requires one to consider issues like α-renaming. Whilst we believe our previous work [28] effectively mitigates (1), at the expense of introducing axioms, the complexities associated with (2) still remain. Nevertheless, the approach seems necessary

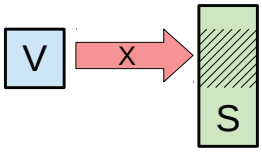

Fig. 1. Visualisation of a simple lens

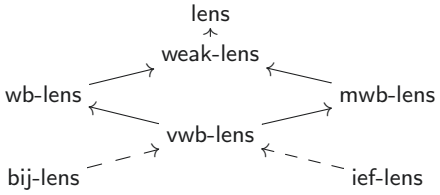

Fig. 2. Lens algebraic hierarchy

to model dynamic creation of variables, as required, for example, in separation logic [26,9].

Use of records to model state is well-applied to verification in Isabelle [1,10,2]. A variable in this model is abstractly represented by pairing the field-query and update functions, f_i and f_i-*upd*, yielding a nameless representation. As shown in [10,2], this approach greatly simplifies automation of program verification in comparison to the functional approach through directly harnessing the polymorphic type system and automated proof tactics. However, the expense is a loss of flexibility compared to the functional approach, particularly in regards to decomposition of state-spaces and handling of extension as required for local variables [27]. Moreover, those employing records seldom provide general support for meta-logical concepts like substitution, and do not abstractly characterise their behaviour.

Our lens approach generalises all these models by abstractly characterising the behaviour of state and variables. Lenses were created as an abstraction for bidirectional programming and solving the view-update problem [12]. They abstract different views on a data space, and allow their manipulation independently of the context. A lens consists of two functions: *get* that extracts a view from a larger source, and *put* that puts back an updated view. [11] gives a detailed study of the algebraic lens laws for these functions. Combinators are also provided for composing lenses [13,12]. They have been practically applied in the *Boomerang* language[4] for transformations on textual data structures.

Our lens approach is indeed related to the state-space solution in [27] of using Isabelle locales to characterise a state type abstractly and polymorphically. A difference though is the use of explicit names, where our lenses are nameless. Moreover, the core lens laws [11] bear a striking resemblance to Back's variable laws [5], which he uses to form the basis for the meta-logical operators of substitution, freshness, and specification of procedures.

3 Lenses

In this section, we introduce our lens algebra, which is later used in §4 to give a uniform interface for variables. The lens laws in §3.1 and composition operator of §3.3 are adapted from [12,11], though the remaining operators, such as independence and sublens, are novel. All definitions and theorems have been mechanically validated[3].

[4] Boomerang home page: `http://www.seas.upenn.edu/~harmony/`

3.1 Lens laws

A lens $X : V \Longrightarrow S$, for source type S and view type V, identifies V with a subregion of S, as illustrated in Figure 1. The arrow denotes X and the hatched area denotes the subregion V it characterises. Transformations on V can be performed without affecting the parts of S outside the hatched area. The lens signature consists of a pair of total functions[5] $\mathit{get}_X : S \Rightarrow V$ that extracts a view from a source, and $\mathit{put}_X : S \Rightarrow V \Rightarrow S$ that updates a view within a given source. When speaking about a particular lens we omit the subscript name. The behaviour of a lens is constrained by one or more of the following laws [11].

$$\mathit{get}\,(\mathit{put}\,s\,v) = v \qquad \text{(PutGet)}$$

$$\mathit{put}\,(\mathit{put}\,s\,v')\,v = \mathit{put}\,s\,v \qquad \text{(PutPut)}$$

$$\mathit{put}\,s\,(\mathit{get}\,s) = s \qquad \text{(GetPut)}$$

PutGet states that if we update the view in s to v, then extracting the view yields v. PutPut states that if we make two updates, then the first update is overwritten. GetPut states that extracting the view and then putting it back yields the original source. These laws are often grouped into two classes [12]: *well-behaved lenses* that satisfy PutGet and GetPut, and *very well-behaved lenses* that additionally satisfy PutPut. We also identify *weak lenses* that satisfy only PutGet, and *mainly well-behaved lenses* that satisfy PutGet and PutPut but not GetPut. These weaker classes prove useful in certain contexts, notably in the map lens implementation (see §3.2). Moreover [11,12] also identify the class of *bijective lenses* that satisfy PutGet and also the following law.

$$\mathit{put}\,s\,(\mathit{get}\,s') = s' \qquad \text{(StrongGetPut)}$$

StrongGetPut states that updating the view completely overwrites the state, and thus the source and view are, in some sense, equivalent. Finally we have the class of *ineffectual lenses* whose views do not effect the source. Our complete algebraic hierarchy of lenses is illustrated in Figure 2, where the arrows are implicative.

3.2 Concrete lenses

We introduce lenses that exemplify the above laws and are applicable to modelling different kinds of state-spaces. The function lens (fl) can represent total variable state functions $\mathsf{Var} \Rightarrow \mathsf{Val}$ [15], whilst the map lens (ml) can represent heaps [9]. The record lens (rl) can represent static variables [10,2].

Definition 1 (Function, Map, and Record lenses).

$$\begin{array}{lll@{\qquad}lll}
\mathit{get}_{\mathsf{fl}(k)} & \triangleq & \lambda f.\,f(k) & \mathit{put}_{\mathsf{fl}(k)} & \triangleq & \lambda f\ v.\,f(k := v) \\
\mathit{get}_{\mathsf{ml}(k)} & \triangleq & \lambda f.\,\mathsf{the}(f(k)) & \mathit{put}_{\mathsf{ml}(k)} & \triangleq & \lambda f\ v.\,f(k \mapsto v) \\
\mathit{get}_{\mathsf{rl}(f_i)} & \triangleq & f_i & \mathit{get}_{\mathsf{rl}(f_i)} & \triangleq & \lambda\, r\ v.\,f_i\text{-}\mathit{upd}\,(\lambda\, x.\, v)\, r
\end{array}$$

[5] Partial functions are sometimes used in the literature, e.g. [13]. We prefer total functions, as these circumvent undefinedness issues and are at the core of Isabelle/HOL.

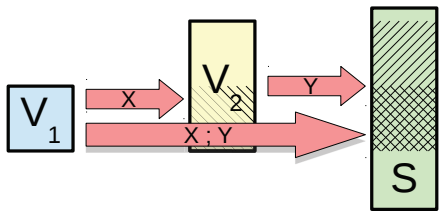

Fig. 3. Lens composition visualised

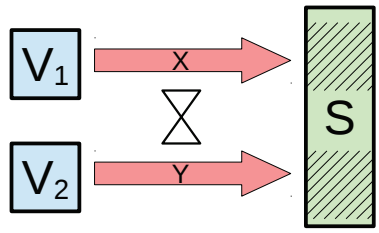

Fig. 4. Lens independence visualised

The (total) function lens $\mathsf{fl}(k)$ focusses on a specific output associated with input k. The *get* function applies the function to k, and the *put* function updates the valuation of k to v. It is a very well-behaved lens:

Theorem 1 (The function lens is very well-behaved).

Proof. Included in our mechanised Isabelle theories[3].

The map lens $\mathsf{ml}(k)$ likewise focusses on the valuation associated with a given key. The map lens is not well-behaved since we cannot satisfy GetPut, as $f(k \mapsto \mathsf{the}(f(k))) \neq f$ when $k \notin \mathrm{dom}(f)$ since the functions have different domains.

Theorem 2 (The map lens is mainly well-behaved).

Finally, we consider the record lens $\mathsf{rec}(f_i)$. As mentioned in §2.3, each record field yields a pair of functions f_i and f_i-upd, and associated simplifications for record instances. Together these can be used to prove the following theorem:

Theorem 3 (Record lens). *Each $f_i : \mathcal{R} \Rightarrow \tau_i$ yields a very well-behaved lens.*

This must be proved on a case-by-case basis for each field in each newly defined record; however the required proof obligations can be discharged automatically.

3.3 Lens algebraic operators

Lens composition $X \fatsemi Y : V_1 \Longrightarrow S$, for $X : V_1 \Longrightarrow V_2$ and $Y : V_2 \Longrightarrow S$ allows one to focus on regions within larger regions. The intuition in Figure 3 shows how composition of X and Y yields a lens that focuses on the V_1 subregion of S. For example, if a record has a field which is itself a record, then lens composition allows one to focus on the inner fields by composing the lenses for the outer with those of the inner record. The definition is given below.

Definition 2 (Lens composition).

$$\mathit{put}_{X \fatsemi Y} \triangleq \lambda\, s\, v.\, \mathit{put}_Y\, s\, (\mathit{put}_X\, (\mathit{get}_Y\, s)\, v) \qquad \mathit{get}_{X \fatsemi Y} \triangleq \mathit{get}_X \circ \mathit{get}_Y$$

The *put* operator first extracts view V_2 from source S, puts $v : V_1$ into this, and finally puts the combined view. The *get* operator simply composes the *get* functions. Lens composition is closed under all lens classes ($\{\mathsf{weak}, \mathsf{wb}, \mathsf{mwb}, \mathsf{vwb}\}$-lens). We next define the unit lens, $\mathbf{0} : \mathsf{unit} \Longrightarrow S$, and identity lens, $\mathbf{1} : S \Longrightarrow S$.

Definition 3 (Unit and identity lenses).

$$\mathit{put}_{\mathbf{0}} \triangleq \lambda\, s\, v.s \qquad \mathit{get}_{\mathbf{0}} \triangleq \lambda\, s.() \qquad \mathit{put}_{\mathbf{1}} \triangleq \lambda\, s\, v.v \qquad \mathit{get}_{\mathbf{1}} \triangleq \lambda\, s.s$$

The unit lens view is the singleton type unit. Its put has no effect on the source, and get returns the single element (). It is thus an ineffectual lens. The identity lens identifies the view with the source, and it is thus a bijective lens. Lens composition and identity form a monoid. We now consider operators for comparing lenses which may have different view types, beginning with lens independence.

Definition 4 (Lens independence). *Lenses $X : V_1 \Longrightarrow S$ and $Y : V_2 \Longrightarrow S$ are independent, written $X \bowtie Y$, provided they satisfy the following laws:*

$$\begin{aligned} \mathit{put}_X(\mathit{put}_Y\ s\ v)\ u &= \mathit{put}_Y(\mathit{put}_X\ s\ u)\ v && \text{(PutsComm)} \\ \mathit{get}_X(\mathit{put}_Y\ s\ v) &= \mathit{get}_X\ s && \text{(PutIrr1)} \\ \mathit{get}_Y(\mathit{put}_X\ s\ u) &= \mathit{get}_Y\ s && \text{(PutIrr2)} \end{aligned}$$

Intuitively, two lenses are independent if they identify disjoint regions of the source as illustrated in Figure 4. We characterise this by requiring that the put functions of X and Y commute (PutsComm), and that the put functions of each lens has no effect on the result of the get function of the other (PutIrr1,PutIrr2). For example, independence of function lenses follows from inequality of the respective inputs, i.e. $\mathsf{fl}(k_1) \bowtie \mathsf{fl}(k_2) \Leftrightarrow k_1 \neq k_2$. Lens independence is a symmetric relation, and it is also irreflexive ($\neg(X \bowtie X)$), unless X is ineffectual.

The second type of comparison between two lenses is containment.

Definition 5 (Sublens relation). *Lens $X : V_1 \Longrightarrow S$ is a sublens of $Y : V_2 \Longrightarrow S$, written $X \preceq Y$, if the equation below is satisfied.*

$$X \preceq Y \triangleq \exists\, Z : V_1 \Longrightarrow V_2.\ Z \in \mathit{wb\text{-}lens} \wedge X = Z \fatsemi Y$$

The intuition of sublens is simply that the source region of X is contained within that of Y. The definition is explained by the following commuting diagram:

$$\begin{tikzcd} & S & \\ V_1 \arrow[ur, "X"] \arrow[rr, dashed, "Z"] & & V_2 \arrow[ul, "Y"'] \end{tikzcd}$$

Intuitively, Z is a "shim" lens that identifies V_1 with a subregion of V_2. Then, focusing on region V_1 in V_2, followed by focusing on V_2 in S is the same as focusing on V_1 in S. The sublens relation is transitive and reflexive, and thus a preorder. Moreover $\mathbf{0}$ is the least element ($\mathbf{0} \preceq X$), and $\mathbf{1}$ is the greatest element ($X \preceq \mathbf{1}$), provided X is well-behaved. Thus sublens orders lenses by how "effective" they are in terms of the proportion of the source captured. We have also proved the following theorem relating independence to sublens:

Theorem 4 (Sublens preserves independence).

If $X \preceq Y$ and $Y \bowtie Z$ then also $X \bowtie Z$

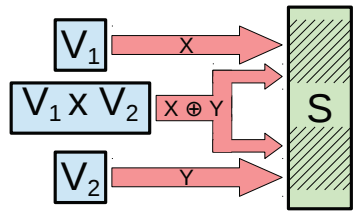

Fig. 5. Lens sum visualised

We use sublens to induce an equivalence relation $X \approx Y \triangleq X \preceq Y \wedge Y \preceq X$. It is a weaker notion than homogeneous HOL equality $=$ between lenses as it allows the comparison of lenses with differently-typed views. We next prove two correspondences between bijective and ineffectual lenses.

Theorem 5 (Bijective and ineffectual lenses equality equivalence).

$$X \in \textit{ief-lens} \Leftrightarrow X \approx \mathbf{0} \qquad X \in \textit{bij-lens} \Leftrightarrow X \approx \mathbf{1}$$

The first law states that ineffectual lenses are equivalent to $\mathbf{0}$, and the second that bijective lenses are equivalent to $\mathbf{1}$. Showing that a lens is bijective thus entails demonstrating that it characterises the whole state space, though potentially with a different view type. We lastly describe lens summation.

Definition 6 (Lens sum).

$$\mathit{put}_{X \oplus Y} \triangleq \lambda\, s\, (u, v).\, \mathit{put}_X\, (\mathit{put}_Y\, s\, v)\, u \qquad \mathit{get}_{X \oplus Y} \triangleq \lambda\, s.(\mathit{get}_X\, s, \mathit{get}_Y\, s)$$

The intuition is given in Figure 5. Given independent lenses $X : V_1 \Longrightarrow S$ and $Y : V_2 \Longrightarrow S$, their sum yields a lens $V_1 \times V_2 \Longrightarrow S$ that characterises both subregions. The combined *put* function executes the *put* functions sequentially, whilst the *get* extracts both values simultaneously. A notable application is to define when a source can be divided into two disjoint views $X \bowtie Y$, a situation we can describe with the formula $X \oplus Y \approx \mathbf{1}$, or equivalently $X \oplus Y \in \mathsf{bij\text{-}lens}$, which can be applied to framing or division of a state space for parallel programs (see §4.4). Lens sum is closed under all lens classes. We also introduce two related lenses for viewing the left and right of a product source-type, respectively.

Definition 7 (First and second lenses).

$$\mathit{put}_{\mathsf{fst}} \triangleq (\lambda(s,t)u.(u,t)) \qquad \mathit{get}_{\mathsf{fst}} \triangleq \mathit{fst}$$
$$\mathit{put}_{\mathsf{snd}} \triangleq (\lambda(s,t)u.(s,u)) \qquad \mathit{get}_{\mathsf{snd}} \triangleq \mathit{snd}$$

We then prove the following lens sum laws:

Theorem 6 (Sum laws). *Assuming $X \bowtie Y$, $X \bowtie Z$, and $Y \bowtie Z$:*

$$\begin{aligned}
X \oplus Y &\approx Y \oplus X & X \oplus (Y \oplus Z) &\approx (X \oplus Y) \oplus Z \\
X \oplus \mathbf{0} &\approx X & (X \oplus Y) \fatsemi Z &= (X \fatsemi Z) \oplus (Y \fatsemi Z) \\
X &\preceq X \oplus Y & \mathsf{fst} \oplus \mathsf{snd} &= \mathbf{1} \\
X \oplus Y &\bowtie Z \quad \text{if } X \bowtie Z \text{ and } Y \bowtie Z
\end{aligned}$$

Lens sum is commutative, associative, has $\mathbf{0}$ as its identity, and distributes through lens composition. Naturally, each summand is a sublens of the whole, and it preserves independence as the next law demonstrates. The remaining law demonstrates that a product is fully viewed by its first and second component.

4 Unifying state-space abstractions

In this section, we apply our lens theory to modelling state-spaces in the context of the UTP's predicate calculus. We construct the core calculus (§4.1), meta-logical operators (§4.2), apply these to the relational laws of programming (§4.3), and finally give an algebraic basis to parallel-by-merge (§4.4). We also show that our model satisfies various important algebras, and thus justify its adequacy.

4.1 Alphabetised predicate calculus

Our model of alphabetised predicates is $\alpha \Rightarrow \mathsf{bool}$, where α is a suitable type for modelling the alphabet, that corresponds to the state-space. We do not constrain the structure of α, but require that variables be modelled as lenses into it. For example, the record lens rl can represent a typed static alphabet [10,2], whilst the map lens ml can support dynamically allocated variables [9]. Moreover, lens composition can be used to combine different lens-based representations of state. We begin with the definition of types for expressions, predicates, and variables.

Definition 8 (UTP types).

$$(\tau, \alpha)\,\mathsf{uexpr} \triangleq (\alpha \Rightarrow \tau) \qquad \alpha\,\mathsf{upred} \triangleq (\mathsf{bool}, \alpha)\,\mathsf{uexpr}$$

$$(\alpha, \beta)\,\mathsf{urel} \triangleq (\alpha \times \beta)\,\mathsf{upred} \qquad (\tau, \alpha)\,\mathsf{uvar} \triangleq (\tau \Longrightarrow \alpha)$$

All types are parametric over alphabet type α. An expression $(\tau, \alpha)\,\mathsf{uexpr}$ is a query function mapping a state α to a given value in τ. A predicate $\alpha\,\mathsf{upred}$ is a boolean-valued expression. A (heterogeneous) relation is a predicate whose alphabet is $\alpha \times \beta$. A variable $x : (\tau, \alpha)\,\mathsf{uvar}$ is a lens that views a particular subregion of type τ in α, which affords a very general state model. We already have meta-logical functions for variables, in the form of lens equivalence $\approx$ and lens independence $\bowtie$. Moreover, we can construct variable sets using operators $\mathbf{0}$ which corresponds to $\emptyset$, $\oplus$ which corresponds to $\cup$, $\mathbf{1}$ which corresponds to the whole alphabet, and $\preceq$ that can model set membership $x \in A$. Theorem 6 justifies these interpretations. We define several core expression constructs for literals, variables, and operators, from which most other operators can be built.

Definition 9 (UTP expression constructs).

$$\mathsf{lit} : \tau \Rightarrow \tau\,\mathsf{uexpr} \qquad \mathsf{var} : (\tau, \alpha)\,\mathsf{uvar} \Rightarrow (\tau, \alpha)\,\mathsf{uexpr}$$
$$\mathsf{lit}\,k \triangleq \lambda\, s.\, k \qquad \mathsf{var}\,x \triangleq \lambda\, s.\, \mathit{get}_x\, s$$

$$\mathsf{uop} : (\tau \Rightarrow \phi) \Rightarrow (\tau, \alpha)\,\mathsf{uexpr} \Rightarrow (\phi, \alpha)\,\mathsf{uexpr}$$
$$\mathsf{uop}\,f\,v \triangleq \lambda\, s.\, f\,(v(s))$$

$$\mathsf{bop} : (\tau \Rightarrow \phi \Rightarrow \psi) \Rightarrow (\tau, \alpha)\,\mathsf{uexpr} \Rightarrow (\phi, \alpha)\,\mathsf{uexpr} \Rightarrow (\psi, \alpha)\,\mathsf{uexpr}$$
$$\mathsf{bop}\,f\,u\,v \triangleq \lambda\, s.\, f\,(u(s))\,(v(s))$$

A literal lit lifts a HOL value to an expression via a constant λ-abstraction, so it yields the same value for any state. A variable expression var takes a lens and applies the *get* function on the state space s. Constructs uop and bop lift functions to unary and binary operators, respectively. These lifting operators enable a proof tactic for predicate calculus we call pred-tac [15] that uses the transfer package [21] to compile UTP expressions and predicates to HOL predicates, and afterwards apply auto or sledgehammer to discharge the resulting conjecture. Unless otherwise stated, all theorems below are proved in this manner.

The predicate calculus' boolean connectives and equality are obtained by lifting the corresponding HOL functions, leading to the following theorem:

Theorem 7 (Boolean Algebra). *UTP predicates form a Boolean Algebra*

We define the refinement order on predicates $P \sqsubseteq Q$, as usual, as universally closed reverse implication $[Q \Rightarrow P]$, and use it to prove the following theorem.

Theorem 8 (Complete Lattice). *UTP predicates form a Complete Lattice*

This then equips us with suprema ($\bigsqcup$), infima ($\bigsqcap$), and fixed points (μ, ν) which allow us to express recursion. The bottom of the lattice is **true**, the most non-deterministic program or specification, and the top is **false**, the miraculous program. The next step is to define the existential and universal quantifiers, whose behaviours are modelled using the lens operation *put*:

Definition 10 (Existential and universal quantifiers).

$$\exists\, x \bullet P \triangleq (\lambda\, s.\, \exists\, v.P(\mathit{put}_x\; s\; v)) \qquad \forall\, x \bullet P \triangleq (\lambda\, s.\, \forall\, v.P(\mathit{put}_x\; s\; v))$$

The quantifiers on the right-hand side are the built-in HOL quantifiers. The existential quantifier declares that there is a valuation for variable x in state s such that P holds, specified using the *put* operator. Universal quantification is defined similarly and satisfies $(\forall\, x \bullet P) = (\neg\, \exists\, x \bullet \neg P)$. We derive the universal closure $[P] \triangleq \forall\, \mathbf{1} \bullet P$, that quantifies all variables in the alphabet ($\mathbf{1}$). We show that alphabetised predicates form a Cylindric Algebra [17] which axiomatises the quantifiers of a first-order logic.

Theorem 9 (Cylindric Algebra). *UTP predicates form a Cylindric Algebra; the following laws are satisfied for well-behaved lenses x, y, and z:*

$$(\exists\, x \bullet \mathbf{false}) \Leftrightarrow \mathbf{false} \tag{C1}$$

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

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

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

$$(x = x) \Leftrightarrow \mathbf{true} \tag{C5}$$

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

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

Proof. Most proofs are automatic, the one complexity being C4 which we have to split into cases for (1) $x \bowtie y$, when x and y are different, and (2) $x \approx y$, when they're the same. We thus implicitly assume that variables cannot overlap, though lenses can. C6 and C7 similarly require independence assumptions.

From this algebra, the usual laws of quantification can be derived [17], even for nameless variables. Since lenses can also represent variable sets, we can also model quantification over multiple variables such as $\exists\, x, y, z \bullet P$, which is represented as $\exists\, x \oplus y \oplus z \bullet P$, and then prove the following laws.

Theorem 10 (Existential quantifier laws).

$$(\exists\, A \oplus B \bullet P) = (\exists\, A \bullet \exists\, B \bullet P) \tag{Ex1}$$

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

$$(\exists\, x \bullet P) = (\exists\, y \bullet Q) \qquad \mathit{if}\, x \approx y \tag{Ex3}$$

Ex1 shows that quantifying over two disjoint sets or variables equates to quantification over both. 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.

In addition to quantifiers for UTP variables we also provide quantifiers for HOL variables in UTP expressions, $\exists\, x \bullet P$ and $\forall\, x \bullet P$, that bind x in a closed λ-term. These are needed to quantify logical meta-variables, which are often useful in proof. This completes the specification of the predicate calculus.

4.2 Meta-logical operators

We next move onto the meta-logical operators, first considering fresh variables, which we model by a weaker semantic property known as *unrestriction* [24,15].

Definition 11 (Unrestriction).

$$x \sharp P \;\Leftrightarrow\; (\forall\, s, v \bullet P(\mathsf{put}_x\, s\, v) = P(s))$$

Intuitively, lens x is unrestricted in P, written $x \sharp P$, provided that P's valuation does not depend on x. Specifically, the effect of P evaluated under state s is the same if we change the value of x. It is thus a sufficient notion to formalise the meta-logical provisos for the laws of programming. Unrestriction can alternatively be characterised as predicates whose satisfy the fixed point $P = (\exists\, x \bullet P)$ for very well-behaved lens x. We now show some of the key unrestriction laws.

Theorem 11 (Unrestriction laws).

$$\text{U1}\ \frac{-}{\mathbf{0} \sharp P} \qquad \text{U2}\ \frac{x \preceq y \quad y \sharp P}{x \sharp P} \qquad \text{U3}\ \frac{x \sharp P \quad y \sharp P \quad x \bowtie y}{(x \oplus y) \sharp P}$$

$$\text{U4}\ \frac{-}{x \sharp \mathbf{true}} \qquad \text{U5}\ \frac{x \sharp P \quad x \sharp Q}{x \sharp P \wedge Q} \qquad \text{U6}\ \frac{x \sharp P \quad x \sharp Q}{x \sharp (P = Q)} \qquad \text{U7}\ \frac{x \sharp P}{x \sharp \neg P}$$

$$\text{U8}\ \frac{x \bowtie y}{x \sharp y} \qquad \text{U9}\ \frac{x \in \mathsf{mwb\text{-}lens}}{x \sharp (\exists\, x \bullet P)} \qquad \text{U10}\ \frac{x \bowtie y \quad x \sharp P}{x \sharp (\exists\, y \bullet P)} \qquad \text{U11}\ \frac{-}{x \sharp [P]}$$

Laws U1–U3 correspond to unrestriction of multiple variables using the lens operations; for example U2 states that sublens preserves unrestriction. Laws U4–U7 show that unrestriction distributes through the logical connectives. Laws U8–U11 show the behaviour of unrestriction with respect to variables. U8 states that x is unrestricted in variable expression y if x and y are independent. U9 and U10 relate to unrestriction over quantifiers; the proviso $x \in \mathsf{mwb\text{-}lens}$ means, for example, that a law is applicable to variables modelled by maps. Finally U11 states that all variables are unrestricted in a universal closure.

We next introduce substitution $P[v/x]$, which is also encoded semantically using homogeneous substitution functions $\sigma : \alpha \Rightarrow \alpha$ over state space α. We define functions for application, update, and querying of substitutions:

Definition 12 (Substitution functions).

$$\begin{aligned} \sigma \dagger P &\triangleq \lambda\, s. P(\sigma(s)) \\ \sigma(x \mapsto_s e) &\triangleq (\lambda\, s.\, \mathit{put}_x\, (e(s))\, (\sigma(s)) \\ \langle \sigma \rangle_s\, x &\triangleq (\lambda\, s.\, \mathit{get}_x\, (\sigma(s))) \end{aligned}$$

Substitution application $\sigma \dagger P$ takes the state, applies σ to it, and evaluates P under this updated state. The simplest substitution, $\mathsf{id} \triangleq \lambda\, x.\, x$, effectively maps all variables to their present value. Substitution lookup $\langle \sigma \rangle_s\, x$ extracts the expression associated with variable x from σ. Substitution update $\sigma(x \mapsto_s e)$ assigns the expression e to variable x in σ. It evaluates e under the incoming state s and then puts the result into the state updated with the original substitution σ applied. We also introduce the short-hand $[x_1 \mapsto_s e_1, \cdots, x_n \mapsto_s e_n] = \mathsf{id}(x_1 \mapsto_s e_1, \cdots, x_n \mapsto_s e_n)$. A substitution $P[e_1, \cdots, e_n/x_1, \cdots, x_n]$ of n expressions to corresponding variables is then expressed as $[x_1 \mapsto_s e_1, \cdots, x_n \mapsto_s e_n] \dagger P$.

Theorem 12 (Substitution query laws).

$$\langle \sigma(x \mapsto_s e) \rangle_s\, x = e \tag{SQ1}$$

$$\langle \sigma(y \mapsto_s e) \rangle_s\, x = \langle \sigma \rangle_s\, x \qquad \mathit{if}\, x \bowtie y \tag{SQ2}$$

$$\sigma(x \mapsto_s e, y \mapsto_s f) = \sigma(y \mapsto_s f) \qquad \mathit{if}\, x \preceq y \tag{SQ3}$$

$$\sigma(x \mapsto_s e, y \mapsto_s f) = \sigma(y \mapsto_s f, x \mapsto_s e) \qquad \mathit{if}\, x \bowtie y \tag{SQ4}$$

SQ1 and SQ2 show how substitution lookup is evaluated. SQ3 shows that an assignment to a larger lens overrides a previous assignment to a small lens and SQ4 shows that independent lens assignments can commute. We next prove the laws of substitution application.

Theorem 13 (Substitution application laws).

$$\sigma \dagger x = \langle \sigma \rangle_s\, x \tag{SA1}$$

$$\sigma(x \mapsto_s e) \dagger P = \sigma \dagger e \qquad \mathit{if}\, x \sharp P \tag{SA2}$$

$$\sigma \dagger \mathsf{uop}\, f\, v = \mathsf{uop}\, f\, (\sigma \dagger v) \tag{SA3}$$

$$\sigma \dagger \mathsf{bop}\, f\, u\, v = \mathsf{bop}\, f\, (\sigma \dagger u)\, (\sigma \dagger v) \tag{SA4}$$

$$(\exists\, y \bullet P)[e/x] = (\exists\, y \bullet P[e/x]) \qquad \mathit{if}\, x \bowtie y, y \sharp e \tag{SA5}$$

These laws effectively subsume the usual syntactic substitution laws, for an arbitrary number of variables, many of which simply show how substitution distributes through expression and predicate operators. SA2 shows that a substitution of an unrestricted variable has no effect. SA5 captures when a substitution can pass through a quantifier. The variables x and y must be independent, and furthermore the expression e must not mention y such that no variable capture can occur. Finally, we will use unrestriction and substitution to prove the one-point law of predicate calculus [16, §3.1].

Theorem 14 (One-point).

$$(\exists\, x \bullet P \wedge x = e) = P[e/x] \qquad \textit{if}\, x \in \mathsf{mwb\text{-}lens}, x \sharp e$$

Proof. By predicate calculus with pred-tac.

The one-point law states that a quantification can be eliminated if precisely one value for the quantified variable is specified. We state the requirement "x does not appear in e" with unrestriction. Thus we have now constructed a set of meta-logical operators and laws which can be applied to the laws of programming, all the while remaining within our algebraic lens framework and mechanised model. Indeed, all our operators are deeply encoded first-class entities in Isabelle/HOL.

4.3 Relational laws of programming

We now show how the general setting of lenses within the relational calculus can be applied to prove the common laws of programming. We begin by augmenting the alphabetised predicate calculus with relational variables and operators. Recall that a relation is simply a predicate over a product state: $(\alpha \times \beta)\,\mathsf{upred}$. Input and output variables can therefore be specified as lenses that focus on the before and after state, respectively.

Definition 13 (Relational variables).

$$[\![x]\!] = x \fatsemi \mathbf{fst} \qquad [\![x']\!] = x \fatsemi \mathbf{snd}$$

A variable x is lifted to a input variable x by composing it with **fst**, or to an output variable x' by composing it with **snd**. We can then proceed to define the operators of the relational calculus.

Definition 14 (Relational operators[6]).

$$P \,;\, Q \;\triangleq\; \exists\, v \bullet P[v/\mathbf{1}'] \wedge Q[v/\mathbf{1}] \qquad \mathbb{I} \;\triangleq\; (\mathbf{1}' = \mathbf{1})$$

$$P \lhd b \rhd Q \;\triangleq\; (b \wedge P) \vee (\neg b \wedge Q) \qquad x := v \;\triangleq\; \mathbb{I}[v/x]$$

[6] Technically, we lift the Isabelle/HOL relational operators to implement the operators to harness proof automation, though these definitions are all proven theorems.

The definition of sequential composition is similar to the standard UTP presentation [20], but we use $\mathbf{1}$ and $\mathbf{1}'$ to represent the input and output alphabets of Q and P, respectively. Skip ($\mathbb{I}$) similarly uses $\mathbf{1}$ to state that the before state is the same as the after state. We then combine $\mathbb{I}$ with substitution to define the assignment operator. Note that because x is a lens, and v could be a product expression, that this operator can be used to represent multiple assignments. We also describe the if-then-else conditional operator $P \lhd b \rhd Q$. Sequential composition and skip, combined with the already defined predicate operators, provide us with the facilities for describing point-free while programs [2], which we illustrate by proving that alphabetised relations form a quantale.

Theorem 15 (Unital quantale). *UTP relations form a unital quantale; that is they form a complete lattice and in addition satisfy the following laws:*

$$(P \,;\, Q) \,;\, R = P \,;\, (Q \,;\, R) \qquad\qquad P \,;\, \mathbb{I} = P \;=\; \mathbb{I} ;\, P$$

$$P \,;\, \left(\bigsqcap_{Q \in \mathcal{Q}} Q \right) = \bigsqcap_{Q \in \mathcal{Q}} (P \,;\, Q) \qquad \left(\bigsqcap_{P \in \mathcal{P}} P \right) ;\, Q = \bigsqcap_{P \in \mathcal{P}} (P \,;\, Q)$$

This is proved in the context of Armstrong's Regular Algebra library [3], and so it follows that UTP relations also form a Kleene algebra. This in turn allows definition of iteration using $\mathbf{while}\, b\, \mathbf{do}\, P \triangleq (b \wedge P)^{*} \wedge (\neg b')$, where b' denotes relational converse, and thence to prove the usual laws. We next describe the laws of assignment.

Theorem 16 (Assignment laws).

$$x := e \,;\, P \;=\; P[e/x] \tag{ASN1}$$

$$x := e \,;\, x := f \;=\; x := f \qquad if\, x \sharp f \tag{ASN2}$$

$$x := e \,;\, y := f \;=\; y := f \,;\, x := e \qquad if\, x \bowtie y, x \sharp f, y \sharp e \tag{ASN3}$$

$$x := e \,;\, (P \lhd b \rhd Q) \;=\; (x := e \,;\, P) \lhd b[e/x] \rhd (x := e \,;\, Q) \qquad if\, \mathbf{1}' \sharp b \tag{ASN4}$$

We focus on ASN3 that demonstrates when assignments to x and y commute, and models law LP6 on page 3. Thus we have shown how lenses provide the general setting to model the laws of programming.

4.4 Parallel-by-merge

We further illustrate the flexibility of our model by showing how to express one of the more complicated operators of the UTP: parallel-by-merge. Parallel-by-merge is a general schema for parallel composition as described in [20, Chapter 7] that enables the expression of sophisticated forms of parallelism that merge the output of two programs into a single consistent after state. It is illustrated in Figure 6 for two programs P and Q working on variables x and y. The input

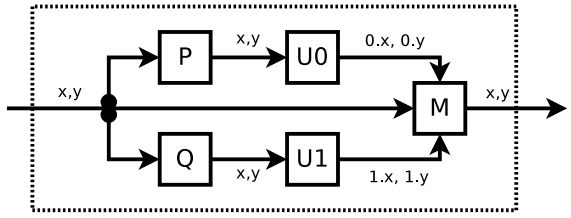

Fig. 6. Pictorial representation of parallel-by-merge $P \,||_M\, Q$

values are fed into P and Q, and their output values are fed into predicates $\mathsf{U0}$ and $\mathsf{U1}$. These rename the variables so that the outputs from both programs can be distinguished by the merge predicate M. M takes as input the variable values before P and Q were executed, and the respective outputs. It then implements a specific mechanism for reconciling these outputs based on a given situation. For example, if P and Q both yield event traces in CSP [18,8], then only those traces that are consistent will be allowed by the merge predicate.

Lenses can be used to define the merge predicate and separating simulations $\mathsf{U0}$ and $\mathsf{U1}$. The merge predicate takes as input three copies of the state: the outputs from P and Q, and the before state. Thus if the state has type A then $M : ((A \times A) \times A, A)\,\mathsf{urel}$, and similarly $\mathsf{U0}, \mathsf{U1} : (A, (A \times A) \times A)\,\mathsf{urel}$. We therefore need syntax to refer to indexed variables $n.x$, and prior variables ${}_<x$, that give the input values, which are described using the following lens compositions:

Definition 15 (Separated and prior variables).

$$[\![0.x]\!] \;=\; x \,⨾\, \mathbf{fst} \,⨾\, \mathbf{fst} \qquad [\![1.x]\!] \;=\; x \,⨾\, \mathbf{snd} \,⨾\, \mathbf{fst} \qquad [\![{}_<x]\!] \;=\; x \,⨾\, \mathbf{snd}$$

Lenses $0.x$ and $1.x$ focus on the first and second elements of the tuple's first element, and ${}_<x$ focusses on the second element. We now define $\mathsf{U0}$ and $\mathsf{U1}$:

Definition 16 (Separating simulations).

$$\mathsf{U0} \;\triangleq\; 0.\mathbf{1}' = \mathbf{1} \wedge {}_<\mathbf{1}' = \mathbf{1} \qquad\qquad \mathsf{U1} \;\triangleq\; 1.\mathbf{1}' = \mathbf{1} \wedge {}_<\mathbf{1}' = \mathbf{1}$$

Both $\mathsf{U0}$ and $\mathsf{U1}$ copy the before value of the whole state into both their respective indexed variables, and also the prior state. We can now describe parallel by merge, given a suitable basic parallel composition operator $||$ which could for example be $\wedge$ or design parallel composition (see [20, Chapter 3]):

Definition 17 (Parallel-by-merge).

$$P \,||_M\, Q \;\triangleq\; ((P \;;\; \mathsf{U0}) \,||\, (Q \;;\; \mathsf{U1})) \;;\; M$$

We also define predicate $\mathsf{swap}_m \triangleq 0.x, 1.x := 1.x, 0.x$ that swaps the left and right copies, and then prove the following generalised commutativity theorem:

Theorem 17 (Commutativity of parallel-by-merge).

If $M \;;\; \mathsf{swap}_m = M$ *then* $P \,||_M\, Q = Q \,||_M\, P$.

This theorem states that if a merge predicate is symmetric, the resulting parallel composition is commutative. In the future we will also show the other properties of parallel composition [20], such as associativity and units. Nevertheless, we have shown that lenses enable a fully algebraic treatment of parallelism.

5 Conclusions

We have presented an enriched theory of lenses, with algebraic operators and comparators, and shown how it can be applied to generically modelling the state space of a program model. We also showed how lenses characterise variables, and can be used to express meta-logical properties and support the laws of programming. The theory of lenses is general, and we believe it has many other applications. We have also defined various other useful lens operations, such as lens quotient that is dual to composition, that space has not allowed us to cover, but is useful for expressing the contraction of a state space. We will thus perform further study of the algebraic properties of these operators.

Lenses have proven to be a useful abstraction for reasoning about state, in terms of properties like independence and combination. We have used out model to prove several hundred laws of predicate and relational calculus from the UTP book [20] and other sources [16,8,25]. We've also mechanised Hoare calculus and a weakest precondition calculus with which basic verifications can be performed. Although space has not permitted details, lenses also enable definition of operators like alphabet extension and restriction, through the description of alphabet coercion lenses $\beta \Longrightarrow \alpha$, that are used to represent local variables and methods. In this context we will explore the links with Back's variable calculus [5].

In the future we hope to apply lenses to further theories of programming, in particular separation logic, since our notion of independence bears a striking resemblance to the frame rule. Indeed we hope in the future to use our UTP theorem prover, *Isabelle/UTP*[7], to construct a comprehensive database of proven laws of programming [20], which can then act as a basis for building UTP-based verification tools inspired by [2].

Acknowledgements. This work is partly supported by EU H2020 project *INTO-CPS*, grant agreement 644047. `http://into-cps.au.dk/`.

References

1. 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.
2. 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.
3. A. Armstrong and G. Struth. Automated reasoning in higher-order regular algebra. In *RAMiCS 2012*, volume 7560 of *LNCS*. Springer, September 2012.
4. A. Armstrong, G. Struth, and T. Weber. Program analysis and verification based on Kleene Algebra in Isabelle/HOL. In *ITP*, volume 7998 of *LNCS*. Springer, 2013.
5. R.-J. Back and V. Preoteasa. Reasoning about recursive procedures with parameters. In *Proc. Workshop on Mechanized Reasoning About Languages with Variable Binding*, MERLIN '03, pages 1–7. ACM, 2003.

[7] See our repository at `github.com/isabelle-utp/utp-main/tree/shallow.2016`

6. 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.
7. C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, pages 366–378. IEEE, July 2007.
8. A. Cavalcanti and J. Woodcock. A tutorial introduction to CSP in unifying theories of programming. In *Refinement Techniques in Software Engineering*, volume 3167 of *LNCS*, pages 220–268. Springer, 2006.
9. 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.
10. 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.
11. S. Fischer, Z. Hu, and H. Pacheco. A clear picture of lens laws. In *MPC 2015*, pages 215–223. Springer, 2015.
12. J. Foster. *Bidirectional programming languages*. PhD thesis, University of Pennsylvania, 2009.
13. 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.
14. S. Foster, G. Struth, and T. Weber. Automated engineering of relational and algebraic methods in Isabelle/HOL. In *RAMICS 2011*, volume 6663 of *LNCS*, pages 52–67. Springer, 2011.
15. 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.
16. E. C. R. Hehner. *A Practical Theory of Programming*. Springer, 1993.
17. L. Henkin, J. Monk, and A. Tarski. *Cylindric Algebras, Part I*. North-Holland, 1971.
18. T. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
19. T. Hoare, I. Hayes, J. He, C. Morgan, A. Roscoe, J. Sanders, I. Sørensen, J. Spivey, and B. Sufrin. The laws of programming. *Communications of the ACM*, 30(8):672–687, August 1987.
20. T. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
21. B. Huffman and O. Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *3rd Intl. Conf. on Certified Programs and Proofs*, volume 8307 of *LNCS*, pages 131–146. Springer, 2013.
22. G. Klein et al. seL4: Formal verification of an OS kernel. In *Proc. 22nd Symp. on Operating Systems Principles (SOSP)*, pages 207–220. ACM, 2009.
23. T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
24. M. Oliveira, A. Cavalcanti, and J. Woodcock. Unifying theories in ProofPower-Z. In *UTP 2006*, volume 4010 of *LNCS*, pages 123–140. Springer, 2007.
25. M. Oliveira, A. Cavalcanti, and J. Woodcock. A UTP semantics for Circus. *Formal Aspects of Computing*, 21:3–32, 2009.
26. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2012*, pages 55–74. IEEE Computer Society, 2002.
27. N. Schirmer and M. Wenzel. State spaces – the locale way. In *SSV 2009*, volume 254 of *ENTCS*, pages 161–179, 2009.
28. F. Zeyda, S. Foster, and L. Freitas. An axiomatic value model for Isabelle/UTP. In *Proc. 6th Intl. Symp. on Unifying Theories of Programming (UTP)*, LNCS. Springer, 2016. To appear.