
Proceedings Paper:
Foster, Simon David orcid.org/0000-0002-9889-9514 and Baxter, James (2020) Automated Algebraic Reasoning for Collections and Local Variables with Lenses. In: 18th International Conference on Relational and Algebraic Methods in Computer Science (RAMiCS 2020). Lecture Notes in Computer Science .

https://doi.org/10.1007/978-3-030-43520-2_7

Automated Algebraic Reasoning for Collections and Local Variables with Lenses

Simon Foster[ORCiD] and James Baxter

University of York
firstname.lastname@york.ac.uk

Abstract. Lenses are a useful algebraic structure for giving a unifying semantics to program variables in a variety of store models. They support efficient automated proof in the Isabelle/UTP verification framework. In this paper, we expand our lens library with (1) dynamic lenses, that support mutable indexed collections, such as arrays, and (2) symmetric lenses, which allow partitioning of a state space into disjoint local and global regions to support variable scopes. From this basis, we provide an enriched program model in Isabelle/UTP for collection variables and variable blocks. For the latter, we adopt an approach first used by Back and von Wright, and derive weakest precondition and Hoare calculi. We demonstrate several examples, including verification of insertion sort.

1 Introduction

The use of algebraic structures for derivation of verification tools using theorem provers has been shown to be a successful and flexible approach [1–4]. It allows us to precisely and abstractly characterise the formal semantics of a spectrum of languages utilising different computational paradigms, including hybrid systems, concurrency, pointers, and probability. Once a suitable algebraic structure is fixed, a large array of axiomatic verification calculi can be generated, including Hoare logic [1], differential dynamic logic [4], separation logic [2], and rely-guarantee calculus [3]. This approach has significant advantages over concrete intermediate verification languages (IVLs) [5,6], since it allows us to unify languages and verification calculi at the algebraic level, and so promotes reuse.

Nevertheless, the underlying algebras for program verification largely focus on the *point-free* programming operators – those that do not explicitly characterise program variables – such as sequential composition ($;$) and non-deterministic choice ($\sqcap$). Kleene Algebra with Tests (KAT) [3, 7], for example, can characterise every operator of imperative while-programs, but is not sufficient to fully capture assignment, substitution, frames, and local variable blocks. Operators that manipulate the store via variables have to be defined in the model rather than the algebra [1, 3]. This technically hampers the reuse of theorems across various languages. At the same time, an algebra of state should allow efficient use of automated proof facilities, so as to support scalable verification tools.

Lenses [8–10] allow us to characterise variables as abstract algebraic objects, which can be composed and manipulated. They provide a generic foundation for verification tools that can maximise proof automation in tools like

Isabelle [11, 12]. Although originating from a different intellectual stream [8], lenses are essentially Back and von Wright's variable manipulation functions [13]. However, lenses are also equipped with several operators that allow us to compose state space query operations in sequence and in parallel, for example. Lenses are the foundation for state modelling in our verification framework, Isabelle/UTP [10, 14, 15], which allows the use of UTP semantic models in developing program verification tools.

In previous work [10], we showed how lenses capture a variety of store models. In this paper, we extend our basic lens model in two ways. Firstly, we develop support for indexed collections, which requires the development of *dynamic lenses*. Secondly, we add support for local variables, for which we harness the work of Hoffmann et al. on *symmetric lenses* [16] that allow us to partition the state space into global and local variable scopes. This allows us to determine whether a particular assignment can be moved outside of a block. From this foundation, we adapt Back and von Wright's block operators [13], and prove Hoare logic theorems. Symmetric lenses allow us to unify a variety of variable block approaches, including extensible records [17] and list-based stacks [18].

In order to illustrate these features, consider the insertion sort algorithm:

Example 1.1 (Insertion Sort). 

$$
\begin{array}{l}
\textbf{function}\ \mathit{insertion\text{-}sort}(\mathit{arr} : [\mathit{int}]\mathit{array}) \\
\quad \textbf{var}\ i, j : \mathit{nat} \bullet \\
\quad \textbf{for}\ i := 1\ \textbf{to}\ (\mathit{length}(\mathit{arr}) - 1)\ \textbf{do} \\
\qquad j := i\ ; \\
\qquad \textbf{while}\ (0 < j \wedge \mathit{arr}[j] < \mathit{arr}[j-1])\ \textbf{do} \\
\qquad\quad (\mathit{arr}[j-1], \mathit{arr}[j]) := (\mathit{arr}[j], \mathit{arr}[j-1])\ ;\ j := j - 1 \\
\qquad \textbf{od}\ \textbf{od}
\end{array}
$$

It introduces two local variables, i and j, that are used to index into the array. The outer loop iterates through the list using i, and the inner loop inserts element i in the correct position into $\mathit{arr}[0...i-1]$, using j to count down. To give this a semantics, we need to (1) allow assignment to the indices of a collection, and (2) extend the state space to add i and j as local variables. Our goal is to support this abstract algorithmic presentation directly in our tool, through a shallow embedding, and provide syntax-directed reasoning support.

Our approach reduces reasoning about programs to proving properties of the state space. It is therefore applicable to any language semantics with an explicit state space model, including reactive [11] and hybrid languages [4, 19]. The approach is therefore abstract, but also maximises Isabelle's proof automation.

The structure of this paper is as follows. After consideration of related work in §2, we describe how lenses give an algebraic semantics to variables in §3. In §4 we give an overview of Isabelle/UTP. In §5, we consider how a state space can be manipulated using lens operators. In §6 we describe dynamic lenses, which are needed for collections, like arrays. In §7, we describe our algebraic characterisation of symmetric lenses, and exhibit several models. In §8, we use symmetric lenses to implement local variable blocks. In §9 we use all the aforementioned results to verify insertion sort in Isabelle/UTP. Finally, in §10, we conclude.

All definitions and theorems are mechanised in Isabelle/UTP, and are often accompanied by an icon () linking to the corresponding repository artefact.

2 Related Work

Isabelle/UTP [10,14,15,20] is a semantic framework for verification tools based in Hoare and He's Unifying Theories of Programming (UTP) [21]. It is broadly comparable to IVLs like Boogie [5] and Why3 [6], but harnesses algebraic and denotational semantic techniques, for application to languages of multiple paradigms. The UTP relational program model is built as a shallow embedding [20] in Isabelle/HOL, and so we compare with similar techniques in this prover.

Simpl is an IVL developed by Schirmer in Isabelle/HOL [17,22]. It is used in the AutoCorres verification platform [23] that was applied in the seL4 project[1]. It uses state monads augmented with exceptions to model low-level code. Our aim is to support the features and efficiency of Simpl, but using relational calculus and algebra to characterise language features abstractly so that they can be transferred between semantic models. Their work does not provide an algebraic semantics for variables, which we provide by lenses, but their comprehensive study of state space modelling techniques is a strong foundation for us [22].

Dongol et al. [24] characterise variables algebraically using Cylindric Kleene Lattices, which extend Kleene algebra with Cylindrification to support quantification. This, in turn, allows expression of both frames and local variable blocks. Their work is largely complementary to ours, since we focus on the algebraic semantics of the variables themselves. They use ordinals as indices into an implicit state space, whereas we characterise the state space explicitly. Our use of lenses also allows us to harness type checking and proof automation in Isabelle.

3 State Space Modelling with Lenses

Here, we review lenses [10], which give algebraic semantics to variables. Novelties include the *list-lens* and a more precise presentation compared to previous work [10]. We use the notation $X : V \Longrightarrow S$ when X is a lens that characterises a V-shaped subregion of a state space S. For instance, in a state space $A \times B$, we can define two lenses: $\mathbf{fst}_B^A : A \Longrightarrow A \times B$ and $\mathbf{snd}_B^A : B \Longrightarrow A \times B$, that select the respective components. As usual [8], we define lenses using two functions:

Definition 3.1. *A lens is a quadruple* $X \triangleq (V, S, \mathsf{get}, \mathsf{put})$*, where* V *and* S *are non-empty sets called the view type and state space, respectively, and* $\mathsf{get} : S \to V$ *and* $\mathsf{put} : S \to V \to S$ *are total functions. We often subscript* get *and* put *with the name of a lens. We define* $\mathsf{create}_X\ v \triangleq \mathsf{put}_X\ (\varepsilon s \bullet s \in S)\ v$*, which constructs an arbitrary, but fixed, state using Hilbert's choice* (ε) *and puts* v *into it.* 

For example, $\mathbf{fst}_B^A \triangleq (A, A \times B, \lambda(x, y) \bullet x, \lambda x'\,(x, y) \bullet (x', y))$, selects and updates the first element of a pair, leaving the second element unchanged. Lenses

[1] The seL4 microkernel verification project: http://sel4.systems.

provide an intuitive and obvious way to model variables in a state space (cf. [13, 18]), which can be queried and updated using the two functions. Intuitively, we can think of them as pointers to distinct regions of a memory store modelled by S. As previously highlighted [10, 22], there are a variety of possible memory models, and lenses provide a uniform algebraic interface for them.

Lenses provide the starting point for the UTP relational calculus [21], which has a model for imperative programs, including operators like sequential composition ($P \fatsemi Q$), conditional ($P \lhd b \rhd Q$), and assignment ($x := e$). Assignment is polymorphic over any lens x, provided that e matches its view type (see §4).

The behaviour of lenses is constrained by three intuitive axioms:

$$\mathit{get}\,(\mathit{put}\,s\,v) = v \quad \text{(L1)} \qquad \mathit{put}\,(\mathit{put}\,s\,v')\,v = \mathit{put}\,s\,v \quad \text{(L2)} \qquad \mathit{put}\,s\,(\mathit{get}\,s) = s \quad \text{(L3)}$$

L1 states that a value put can be retrieved. L2 states that an earlier put is overwritten by a later put. L3 states that retrieving a value and then putting it back yields the original state. We distinguish *total* lenses, that obey all three axioms, from *partial* lenses that obey only L1 and L2. The ***fst*** and ***snd*** lenses are both total. A further example of a total lens is the total function lens:

$$\mathit{fun\text{-}lens}_B(x : A) \triangleq (B, A \to B, \lambda f \bullet f\,x, \lambda f\,v \bullet f(x := v))$$

It points to the value associated with a particular domain element x. It is useful, for instance, when the state space has type $\mathit{Name} \to \mathit{Value}$, which associates a named variable with a value in a given universe. The *get* function simply applies f, and the *put* function updates the value associated with x. It is clear that this lens obeys all three laws. We also define a relation called independence, $X \bowtie Y$, that characterises when two lenses view disjoint regions of the state space:

$$X \bowtie Y \triangleq \forall (s, u, v) \bullet \begin{pmatrix} \mathit{put}_X(\mathit{put}_Y\ s\ v)\ u &=& \mathit{put}_Y(\mathit{put}_X\ s\ u)\ v \\ \land\ \mathit{get}_X(\mathit{put}_Y\ s\ v) &=& \mathit{get}_X\ s \\ \land\ \mathit{get}_Y(\mathit{put}_X\ s\ u) &=& \mathit{get}_Y\ s \end{pmatrix} \quad \text{if } S_X = S_Y$$

It is defined only when the state spaces are the same: $S_X = S_Y$. X and Y are independent provided applications of *put* commute, and each *get* function is unaffected by the corresponding *put* function. If X and Y are both total lenses, then the second and third conjuncts can be omitted. $X \bowtie Y$ means that X and Y do not interact, for example $\mathit{fun\text{-}lens}_B(i) \bowtie \mathit{fun\text{-}lens}_B(j)$, provided that $i \neq j$.

Partial lenses, which do not obey L3, are motivated by partial structures, such as arrays and heaps. The cells of an array can be modelled using list lenses:

$$\mathit{list\text{-}lens}_A(i : \mathbb{N}) \triangleq (A, [A]\mathit{list}, \lambda\, xs \bullet xs\,!\,i, \lambda\, xs\ v \bullet xs[i := v])$$

A lens $\mathit{list\text{-}lens}_A(i) : A \Longrightarrow [A]\mathit{list}$ points to the ith element of an inductive list of values drawn from A. The HOL operator $xs\,!\,i$ returns the ith element of xs, or an arbitrary element of A if $i \geq \#xs$, where $\#xs$ is the length. The operator $xs[i := v]$ updates the ith element to take value v. If xs is not long enough to hold v, it is first expanded by filling in the extra elements with arbitrary values. Here, *list-lens* and *fun-lens* are both examples of lenses indexed by a set. As for *fun-lens*, we have it that $\mathit{list\text{-}lens}_B(i) \bowtie \mathit{list\text{-}lens}_B(j)$ provided that $i \neq j$.

Clearly, *list-lens* satisfies both L1 and L2: we can always place a value in the ith component, potentially several times, and retrieve it. However, it does not satisfy L3. When a list is too short ($i \geq \#xs$), *list-lens*(i) returns an arbitrary value, which, if placed at i, alters the list structure and violates L3. Consequently, whilst *fun-lens* is a total lens, *list-lens* is only partial, and the same follows for data structures like partial functions. Nevertheless, as we shall see, partial lenses are sufficient to support most of the laws we need for verification calculi.

A useful class of state space is induced by records. In Isabelle/UTP, we can define a state type $\mathit{rec\text{-}typ} \triangleq [x_1 : A_1 \cdots x_m : A_m]$ for m fields, each with a given type. Technically, it is isomorphic to a product type $A_1 \times \cdots \times A_m$, but with named lenses for manipulating each field. The ***alphabet*** command automates the creation of these lenses, and generates theorems that $x_i \bowtie x_j$ for any $i \neq j$.

State types can also be extended: $\mathit{rec\text{-}typ}_2 \triangleq \mathit{rec\text{-}typ} + [y_1 : B_1 \cdots y_n : B_n]$, which allows hierarchy. This approach is used in the IVL Impl [17] to represent local variables, and here we adopt a similar approach. The lenses are polymorphic: $x_i : A_i \Longrightarrow [\alpha]\mathit{rec\text{-}typ\text{-}ext}$, where the parameter α allows application of x_i to both *rec-typ* and extensions thereof, such as $\mathit{rec\text{-}typ}_2$ with $\alpha \cong A_{m+1} \times \cdots \times A_n$. This is important, as it means that the same lens name can be used in different state spaces: x_i can both have the type $A_i \Longrightarrow \mathit{rec\text{-}typ}$ and $A_i \Longrightarrow \mathit{rec\text{-}typ}_2$.

4 Relational Programs in Isabelle/UTP

In this section, we briefly introduce the foundations of Isabelle/UTP, which is a shallow embedding of UTP [21] in Isabelle/HOL. UTP is based on a variant of Tarski's relational calculus [25] where each relation is "alphabetised", meaning it is parameterised by the set of variables to which it can refer. In Isabelle/UTP, we instead opt to have relations parameterised by their state space type S, and variables are then lenses viewing this type. We can therefore use the Isabelle type system to ensure well-formedness: only relations and predicates with compatible alphabets can be composed using the Boolean and relational connectives.

Expressions are total functions: $[V, S]\mathit{expr} \triangleq (S \to V)$, for some state space S and type V. Operators can be pointwise lifted and applied to them, for example, if $e, f : [\mathbb{N}, S]\mathit{expr}$, then $e + f$ denotes $\lambda\, s : S \bullet e\, s + f\, s$. If x and y are lenses, then we can use them in expressions: $x + y$ denotes $\lambda\, s : S \bullet \mathit{get}_x\, s + \mathit{get}_y\, s$. We can determine whether e depends on part of the state using the unrestriction [10]:

Definition 4.1. $(x \sharp e) \triangleq (\forall (s, v) \bullet e\,(\mathit{put}_x\, s\, v) = e\, s)$

Lens x is unrestricted in e, written $x \sharp e$, when updating its value using *put* has no effect on the valuation of e. This can occur, for example, when x is a variable that e does not refer to. Unrestriction distributes through lifted functions [10].

Substitutions between two states spaces are modelled with functions, $\sigma : S_1 \to S_2$. A substitution can be updated using $\sigma(x \mapsto e)$. A heterogeneous substitution can be constructed using $(\!|x_1 \mapsto e_1, \cdots, x_n \mapsto e_n|\!)$, when $x_i : A_i \Longrightarrow S_2$ and $e_i : [A_i, S_1]\mathit{expr}$, which is a set of simultaneous updates. A homogeneous substitution, where $S_1 = S_2$, can be constructed similarly but using square brackets:

$[x \mapsto e, \cdots]$. The difference between these two is that the former gives arbitrary values to unassigned variables, whereas the latter copies the original values. Substitutions can also be composed function-wise, $\sigma \circ \rho$, which corresponds to applications of the updates in ρ followed by those in σ.

We can apply a substitution to an expression using $\sigma \dagger e \triangleq e \circ \sigma$, which likewise composes the substitution and expression functions. Although this may seem redundant, it is useful to distinguish a separate operator to enable bespoke rewrite laws in Isabelle. Then, we can obtain the traditional substitution operator: $p[e/x] \triangleq [x \mapsto e] \dagger p$. Substitutions then obey a number of useful laws:

Theorem 4.2. *If x and y are partial lenses, then the following laws hold:*

$$\sigma(x \mapsto e, y \mapsto f) = \sigma(y \mapsto f, x \mapsto e) \quad \text{if } x \bowtie y \tag{1}$$

$$\sigma(x \mapsto e, x \mapsto f) = \sigma(x \mapsto f) \tag{2}$$

$$\sigma(x \mapsto v) \circ \rho = (\sigma \circ \rho)(x \mapsto (\sigma \dagger v)) \tag{3}$$

$$\sigma(x \mapsto e) \dagger x = e \tag{4}$$

$$\sigma(x \mapsto v) \dagger e = \sigma \dagger e \quad \text{if } x \sharp e \tag{5}$$

Substitution updates commute when made to independent lenses (1), and can cancel earlier ones (2). Substitutions can be composed, and (3) shows how to pull out a variable update to the left-most substitution. These laws can be used to show that $[x \mapsto u] \circ [y \mapsto v]$ is equivalent to $[x \mapsto u[v/y], y \mapsto v]$, when $x \bowtie y$. Substitution application distributes through functions in the obvious way, and can be applied to variable expressions (4). If x is unrestricted in an expression, then any assignment to this variable can be dropped (5).

We define predicates, $[S]\mathit{pred} \triangleq [\mathit{bool}, S]\mathit{expr}$, relations, $[S_1, S_2]\mathit{rel} \triangleq [S_1 \times S_2]\mathit{pred}$, and the usual operators over them. Predicates and relations are ordered by refinement ($\sqsubseteq$). We import theorems for structures like complete lattices, quantales, and Kleene algebras [2,3,10]. With substitutions, it is easy to define a generalised assignment operator, in the style of Back and von Wright [13]: $\langle\sigma\rangle$, which lifts a substitution to a relation in the obvious way. This satisfies a useful law, $\langle\sigma\rangle \fatsemi \langle\rho\rangle = \langle\rho \circ \sigma\rangle$, which allows us to combine sequential assignments. Assignments can then be constructed with $x := e \triangleq \langle[x \mapsto e]\rangle$, and combining with non-deterministic choice ($\sqcap$) we define non-deterministic assignment: $x := * \triangleq \bigsqcap_{v \in V_x} x := v$. These definitions satisfy the laws of programming [26].

5 State Space Manipulation

Here, we show how to manipulate state spaces, and coerce variables and expressions between them. We use two additional relations, that are defined using *get* and *put* [10]: (1) $X \preceq Y$: the view of lens X is contained within the view of Y; (2) $X \approx Y$: the views of X and Y are isomorphic. These are both heterogeneous operators that can relate lenses with different view types. Relation $\preceq$ forms a preorder and $\approx$ is an equivalence relation. Ordering is needed because lenses can characterise both variables and sets thereof. We compose lenses, thus combining their respective views, using the pairing operator:

Definition 5.1 (Lens Pairing).

$$X \oplus Y \triangleq \begin{pmatrix} V_X \times V_Y, S_X, (\lambda\, s \bullet (\mathit{get}_X\, s, \mathit{get}_Y\, s)), \\ (\lambda\, s\, (u, v) \bullet \mathit{put}_X\, (\mathit{put}_Y\, s\, v)\, u) \end{pmatrix} \quad \textit{when } S_X = S_Y, X \bowtie Y$$

Lens pairing combines two independent lenses with the same state space, creating a lens whose view type is $V_X \times V_Y$. The get function pairs the results of the get functions for X and Y, while its put function puts each element using the respective put. Using this, a set of variables, $\{x, y, z\}$ can be characterised by a lens, for example, by the summation $x \oplus y \oplus z$. Moreover, we have it that $X \preceq X \oplus Y$, since $X \oplus Y$ views more of the state space than X.

We define two basic total lenses: $\mathbf{1}_S \triangleq (S, S, \lambda\, s \bullet s, \lambda\, s\, v \bullet v)$ whose view and state space are identical, and $\mathbf{0}_S \triangleq (\{\emptyset\}, S, \lambda\, s \bullet \emptyset, \lambda\, s\, v \bullet s)$, whose view type is unitary. Intuitively, $\mathbf{1}$ characterises the entirety of S, and $\mathbf{0}$ characterises none of it, and cannot distinguish any states. Consequently, we have $\mathbf{0} \preceq X$ and $X \preceq \mathbf{1}$, since these are the least and most distinguishing lenses, respectively.

For variable blocks, we need expansion and contraction of the state space, for both lenses and expressions. For lenses, we define the composition and quotient:

Definition 5.2 (Lens Composition and Quotient).

$$X\,;\ Y \triangleq \begin{pmatrix} V_X, S_Y, \mathit{get}_X \circ \mathit{get}_Y, \\ (\lambda\, s\, v \bullet \mathit{put}_Y\, s\, (\mathit{put}_X\, (\mathit{get}_Y\, s)\, v)) \end{pmatrix} \qquad \textit{when } S_X = V_Y$$

$$X \,/\, Y \triangleq \begin{pmatrix} V_X, V_Y, \mathit{get}_X \circ \mathit{create}_Y, \\ (\lambda\, s\, v \bullet \mathit{get}_Y\, (\mathit{put}_X\, (\mathit{create}_Y\, s)\, v)) \end{pmatrix} \qquad \textit{when } S_X = S_Y$$

$X\,;\ Y$ has been previously defined [8]. It selects a subregion V_1, characterised by $X : V_1 \Longrightarrow V_2$, of a larger region V_2, characterised by $Y : V_2 \Longrightarrow S$. Intuitively, Y denotes a sub-space of S, X is a variable of this sub-space, and so $X\,;\ Y \preceq Y$. We sometimes write $\mathit{obj}{:}\mathit{attr}$ for the composition $\mathit{attr}\,;\ \mathit{obj}$.

We believe the quotient operator, $X \,/\, Y$ is novel[2]. Provided that $X : V_1 \Longrightarrow S$ is constructed by composition of $Y : V_2 \Longrightarrow S$ and $Z : V_1 \Longrightarrow V_2$, we have it that $X \,/\, Y = Z$. The get function first creates an arbitrary state and populates the V_2 region with the incoming state. It then uses the get_X function to obtain the V_1 element. The assumption is that all the information needed to construct a V_1 can be obtained from V_2. The put function creates an S element, uses put_X to update this with $v : V_1$, and finally applies get_Y to obtain a V_2 element. Again, the assumption is that put_X will only manipulate data within V_1.

Lens quotient gives rise to some useful, and intuitive, properties.

Theorem 5.3. $(X\,;\ Y) \,/\, Y = X \qquad (X \,/\, X) = \mathbf{1} \qquad X \,/\, \mathbf{1} = X$

The first identity gives the intuition of quotient: it removes the second element of a composition. The second identity shows that if we remove a lens from itself, then only a residual $\mathbf{1}$ remains. The third identity shows that removal of $\mathbf{1}$ has no effect, because of course $X\,;\ \mathbf{1} = X$.

In addition, we need to expand and contract the state space of expressions:

[2] The similarly named *quotient lens* of Foster et al. [9] is a rather different concept.

Definition 5.4. *We fix* $X : S_1 \Longrightarrow S_2$*, expressions* $e : [A, S_1]\mathsf{expr}$ *and* $f : [B, S_2]\mathsf{expr}$*, and define:* $e \uparrow X \triangleq e \circ \mathsf{get}_X$ *and* $f \downarrow X \triangleq f \circ \mathsf{create}_X$

Here, X is a lens that describes how S_1 is embedded into a larger space S_2. The first operator, $e \uparrow X$, extends the state space of e to be S_2, and the second, $f \downarrow X$, restricts it to be S_1. These operators coerce an expression to have a different type, for use in a context with a different state space. They satisfy several theorems.

Theorem 5.5 (State Space Extension and Restriction).

$$(f\ e_1 \cdots e_n) \uparrow A = f\ (e_1 \uparrow A \cdots e_n \uparrow A) \qquad (f\ e_1 \cdots e_n) \downarrow A = f\ (e_1 \downarrow A \cdots e_n \downarrow A)$$
$$x \uparrow A = A{:}x \qquad x \downarrow A = x \,/\, A$$
$$A \bowtie B \Rightarrow B \sharp (e \uparrow A) \qquad (e \uparrow A) \downarrow A = e$$

Both extension and restriction distribute through function application in the obvious way. Extension of a lens expression entails a lens composition, and restriction entails a lens quotient. If we extend an expression's state space, $e \uparrow A$, then the resulting expression does not depend on a lens B that is independent of A. The reason is that the original state space of e is characterised by A. Finally, we have it that restriction is the inverse of extension. The converse theorem does not hold, because restricting a state space may result in a loss of information.

We can define $e^{\blacktriangleleft} \triangleq e \uparrow \mathit{fst}$, $e_{\blacktriangleleft} \triangleq e \downarrow \mathit{fst}$, and $e^{\blacktriangleright} \triangleq a \uparrow \mathit{snd}$, that characterise relational preconditions and postconditions. Specifically, $e^{\blacktriangleleft}$ lifts an expression on S to one on $S \times S$, thus turning a predicate into a relation. We can characterise initial and final variables, $x^{\blacktriangleleft}$ and $x^{\blacktriangleright}$, in the style of notations like Z. We can also define weakest preconditions, $P\ \mathit{wp}\ b \triangleq (P \fatsemi b^{\blacktriangleleft})_{\blacktriangleleft}$, and also the Hoare triple, $\{\, p \,\}\ Q\ \{\, r \,\} \triangleq (p^{\blacktriangleleft} \Rightarrow q^{\blacktriangleright}) \sqsubseteq Q$. These definitions admit, as theorems, the usual laws [27, 28]. For example, we have the assignment law, $\{\, p[v/x] \,\}\ x := v\ \{\, p \,\}$, for any lens x, and the more general $\{\, \sigma \dagger p \,\}\ \langle \sigma \rangle\ \{\, p \,\}$.

6 Dynamic and Collection Lenses

In this section we give semantics to the notation $x[i]$, which refers to the ith element of a collection x. We model x with a lens that points to a collection, such as a list, and i with an index expression. The generality of the lens axioms means that we can define $x[i]$ itself to be a type of lens, which we call the collection lens. Consequently, we can manipulate it like any other lens, employing the theorems of §4. In order to define this, we first need to define dynamic lenses:

Definition 6.1. *We fix sets* A *and* B *that denote elements and collections, and a set* I *of indices. We assume a family of* I*-indexed lenses* $F : I \to (A \Longrightarrow B)$ *and an expression* $e : B \to I$*. A dynamic lens is defined as follows:*

$$\mathsf{dyn\text{-}lens}\ F\ e \triangleq (A, B, \lambda\, s \bullet \mathsf{get}_{F(e\ s)}\ s, \lambda\, s\ v \bullet \mathsf{put}_{F(e\ s)}\ s\ v)$$

Intuitively, a dynamic lens points to the eth element of the indexed lens F. Since e is an expression, it can change value, and consequently the current index depends on the state space. The *get* and *put* function both instantiate the indexed lens with e applied to the current state, and then apply its respective *get* and *put* function. From this definition, we can prove the following closure theorem:

Theorem 6.2. *We assume that, for all $i : I$, e does not refer to $F\,i$, that is $(F\,i) \sharp e$, and $F\,i$ is a partial or total lens. We can then show that* dyn-lens *$F\ e$ is a partial or total lens, respectively.*

The intuition is that $F\,i$ must satisfy the lens axioms, for all indexes, and the index expression e should not itself refer to the $F\,i$, to avoid self references. From this definition, we can now define collection lenses:

Definition 6.3 (Collection Lenses). *We fix $F : I \to (A \Longrightarrow B)$, a lens indexed by the set I. Then, given a lens $x : B \Longrightarrow S$, for some state space S, and an index expression $e : S \to I$, a collection lens is defined as follows:*

$$x[e] \triangleq \mathit{dyn\text{-}lens}\,(\lambda\, i : I \bullet F\,i\,;\ x)\ e$$

The collection lens, $x[e]$, is a dynamic lens where the underlying indexed lens F is applied after selection of the collection location in the lens x. It is clear that Theorem 6.2 can be applied here too, provided that x is also a total lens. The intuition of the collection lens is perhaps clearer if we consider a concrete example where $F = \mathit{fun\text{-}lens}$. In this case, we can prove the following identity:

$$(x[i] := e) = (x := x(i := e))$$

An assignment to $x[i]$ frames the remainder of the state, and thus x takes its original value with the i index updated. We can also derive the identity:

$$(x[i] := e \fatsemi x[j] := f) \ = \ (x[j] := f \fatsemi x[i] := e)$$

whenever $i \neq j$, $x[i] \sharp f$, and $x[j] \sharp e$, by using the generalised assignment laws.

In Isabelle/UTP, we make F an overloaded polymorphic constant that associates a suitable indexed lens to a collection type. In many situations, $x[e]$ is a partial lens, since it is only meaningful when x is a collection where the key e is defined. For example, the assignment $(arr[j-1], arr[j]) := (arr[j], arr[j-1])$ in Example 1.1 is meaningful only when $j < \#arr$. Thus, when verifying programs with collection lenses, it is necessary to guard them with definedness predicates.

7 Symmetric Lenses

Symmetric lenses [16] stand in contrast to the lenses that were introduced in §3, which are "asymmetric" because, once a view has been extracted from a source, it is not possible to reconstruct the source from the view alone [16]. Symmetric lenses are effectively lenses of type $V \times C \Longrightarrow S$, where C is the "complement"

of V with respect to S – the remainder of S once V is removed. In general, it is not possible to compute the complement of an asymmetric lens. Symmetric lenses thus capture the notion of partitioning the state into disjoint regions. These regions are represented by two lenses, which we refer to as the *view* and the *coview*, and for a given symmetric lens $\mathcal{X}$, we write $\mathcal{V}_\mathcal{X}$ and $\mathcal{C}_\mathcal{X}$ to represent them. Such a partitioning of the state space is fundamental to framing of certain variables, and allows us to distinguish the global and local store.

To characterise symmetric lenses, we must capture both the disjointness of the view and coview, and the fact that, taken together, they cover the state space. Coverage is captured by first combining the view and coview into a pairing $\mathcal{V}_\mathcal{X} \oplus \mathcal{C}_\mathcal{X}$, and requiring that this covers the state space. Such a definition is provided for by the concept of *bijective lenses*, defined below.

Definition 7.1. *A partial bijective lens satisfies L1, and also* $\mathit{put}\, s\, v = \mathit{put}\, s'\, v$. *A (total) bijective lens satisfies L1, and also* $\mathit{put}\, s\, (\mathit{get}\, s') = s'$.

For total bijective lenses, we require that getting the view of s' and putting it into s replaces the whole of s with s'. For a partial lens, *get* may return an incorrect value for states outside its domain, so a partial bijective lens is characterised by $\mathit{put}\, s\, v = \mathit{put}\, s'\, v$. This captures the property that *put* replaces the state space, without constraining *get*. A bijective lens fulfils all the axioms of a partial or total lens, but it is sufficient to require L1, so the overall definition of a bijective lens is as shown above. We can now define symmetric lenses:

Definition 7.2. *A (partial) symmetric lens* $\mathcal{X} \triangleq (\mathcal{V}, \mathcal{C})$ *over a state space* S *is a pair of (partial) total lenses,* $\mathcal{V} : V_1 \Longrightarrow S$ *and* $\mathcal{C} : V_2 \Longrightarrow S$ *such that (1)* $\mathcal{V} \bowtie \mathcal{C}$*, and (2)* $\mathcal{V} \oplus \mathcal{C}$ *is a (partial) bijective lens. We denote the set of symmetric lenses between* $V_1 \times V_2$ *and* S *with the notation* $[V_1, V_2] \Longleftrightarrow [S]$.

As an example of a symmetric lens, consider $\mathcal{X} \triangleq (\mathit{fst}_A, \mathit{snd}_B)$. These lenses are clearly independent, and $\mathit{fst}_A \oplus \mathit{snd}_B$ provides a view of the entire product, so it is a bijective lens. Thus, $\mathcal{X}$ is a (total) symmetric lens.

A more interesting example is the list symmetric lens, the view and coview of which are the head and tail of a list. Formally, they are the head lens, $\mathit{hd}_A : A \Longrightarrow [A]\mathit{list}$, and the tail lens, $\mathit{tl}_A : [A]\mathit{list} \Longrightarrow [A]\mathit{list}$. The head lens is defined in terms of the list lens: $\mathit{hd}_A \triangleq \mathit{list\text{-}lens}_A(0)$. The tail lens is defined as $\mathit{tl}_A \triangleq ([A]\mathit{list}, [A]\mathit{list}, \mathit{tl}, \lambda\, xs\; v \bullet \mathit{hd}\, xs \frown v)$. It gets the tail of the list, and puts xs as the tail of the new list, preserving the old head. These lenses are independent, since they operate on different parts of a list. The head lens, as an instance of the list lens, is a partial lens, since it is not defined for an empty list. The list symmetric lens is thus an example of a partial symmetric lens, since putting a list head and tail replaces the whole list. We note that the tail lens has the same view and source types. This is an important property for allowing variable blocks based on such symmetric lenses to be recursed on [18], as we discuss in §8.

Another symmetric lens is induced by record state spaces, each of which induces two regions: the *base* region, which consists of the defined fields $(x_0 \cdots x_m)$, and the *extension* region, with any additional fields $(y_0 \cdots y_n)$. These can be characterised by $\mathit{base} : \mathit{rec\text{-}typ} \Longrightarrow [\alpha]\mathit{rec\text{-}typ\text{-}ext}$ and $\mathit{more} : \alpha \Longrightarrow [\alpha]\mathit{rec\text{-}typ\text{-}ext}$,

where $\textbf{\textit{base}} \bowtie \textbf{\textit{more}}$ and $\textbf{\textit{base}} \oplus \textbf{\textit{more}}$ is a bijective lens. Consequently, for a record we define $\textbf{\textit{all}} \triangleq (\textbf{\textit{base}}, \textbf{\textit{more}})$, which forms a total symmetric lens. Moreover, we have it that $x_i \preceq \textbf{\textit{base}}$, for $0 \leq i \leq m$, and $y_j \preceq \textbf{\textit{more}}$, for $0 \leq j \leq n$.

The polymorphic nature of a record lens means that type coercions can be handled easily, as the following theorem shows.

Theorem 7.3. $x_i \,;\, \textbf{\textit{base}} = x_i$ *and* $x_i \,/\, \textbf{\textit{base}} = x_i$ *whenever* $x_i \preceq \textbf{\textit{base}}$

Composition and quotient using the ***base*** lens corresponds to moving it into and out of an extended state space. Since $x_i : V_i \Longrightarrow [\alpha]\mathit{rec\text{-}typ\text{-}ext}$ is polymorphic, such a coercion yields the same lens but with a different type. These laws are important for when moving a global variable into a local scope in §8.

8 Variables Blocks

Having defined symmetric lenses, and demonstrated several models, we now use these to characterise local variable blocks. The basic idea is to implement operators analogous to begin and end from Back and von Wright [13, §5.6], that grow and shrink the state space with additional variables.

Here, however, we fix a symmetric lens $\mathcal{X} : [S_2, C] \Longleftrightarrow [S_1]$ to give a concrete semantics to scope expansion and contraction. Intuitively, S_2 is the global state space, S_1 extends S_2 with local variables, and C is the complement of S_2 wrt. S_1. Then we have it that $\mathcal{V}_\mathcal{X}$ characterises the global state region of S_1, and $\mathcal{C}_\mathcal{X}$ the local state region. The symmetric lens allows us to distinguish global and local variables, so that we can determine whether an assignment can be moved outside a block or not. Unlike [13], where types are implicit, we have to explicitly handle type coercion when a variable and expression moves between state spaces.

We give the following program that swaps two variables as a running example:

Example 8.1. $\mathit{swap}(x, y : \mathit{int}) \triangleq \mathbf{var}\ z : \mathit{int} \bullet (z := y \fatsemi y := x \fatsemi x := z)$

It creates a third variable, z, and then uses this as a temporary store in which to place the value of y. We show how this can be modelled and verified in Isabelle/UTP, with the aim of supporting the larger insertion sort example in §9. We first define substitutions that extend and contract the state space.

Definition 8.2 (Extension and Contraction Substitutions).

$$\textbf{\textit{ext}}_\mathcal{X} \triangleq (\!|\mathcal{V}_\mathcal{X} \mapsto \textbf{\textit{v}}, \mathcal{C}_\mathcal{X} \mapsto \varepsilon v \bullet v \in V_2|\!) \qquad \textbf{\textit{con}}_\mathcal{X} \triangleq (\!|\textbf{\textit{v}} \mapsto \mathcal{V}_\mathcal{X}|\!)$$

Here, $\textbf{\textit{ext}}_\mathcal{X} : S_2 \to S_1$ and $\textbf{\textit{con}}_\mathcal{X} : S_1 \to S_2$ are heterogeneous substitutions. Extension assigns the original state ($\textbf{\textit{v}} : S_2$) to the view lens ($\mathcal{V}_\mathcal{X}$), and assigns an arbitrary but fixed element of C to the coview lens. Effectively this extends the state space, retaining the values for the global variables, and assigning an arbitrary value to the local ones. Contraction, conversely, assigns the view lens to the entire state lens, leading to the loss of the local state. Extension and contraction satisfy the theorem below:

Theorem 8.3. *Any symmetric lens* $\mathcal{X}$ *satisfies* $\mathbf{con}_{\mathcal{X}} \circ \mathbf{ext}_{\mathcal{X}} = \mathbf{id}_{S_2}$ 

Specifically, if we extend a state space and then contract it, we always get the original state space back. The converse of this law does not hold since contracting a state space, of course, loses the local state stored in the coview. Moreover, the law only follows for total symmetric lenses since extending and then contracting using a partial lens can alter the state. It is now straightforward to define relations that open and close a block using the substitutions:

Definition 8.4 (Blocks). $\mathbf{open}_{\mathcal{X}} \triangleq \langle \mathbf{ext}_{\mathcal{X}} \rangle \fatsemi \mathcal{C}_{\mathcal{X}} := *$ $\quad \mathbf{close}_{\mathcal{X}} \triangleq \langle \mathbf{con}_{\mathcal{X}} \rangle$ 

Here, $\mathbf{open}_{\mathcal{X}} : [S_2, S_1]\mathit{rel}$ first extends the state space and then non-determinstically assigns a value to the coview, replacing the arbitrary but fixed value. Also, $\mathbf{close}_{\mathcal{X}}$ simply contracts the state space. We prove a useful law:

Theorem 8.5. *Any symmetric lens* $\mathcal{X}$ *satisfies* $\mathbf{open}_{\mathcal{X}} \fatsemi \mathbf{close}_{\mathcal{X}} = \mathit{II}$. 

Aside from being an important property of variable blocks, this law allows us to introduce a local variable block at any point in a program, which can facilitate step-wise refinement. We now prove three algebraic laws for variable blocks and assignments, which are adapted from [13, page 102].

Theorem 8.6 (Variable Block Laws). 

$$x := v \fatsemi \mathbf{open}_{\mathcal{X}} = \mathbf{open}_{\mathcal{X}} \fatsemi \mathcal{V}_{\mathcal{X}} : x := (v \upharpoonright \mathcal{V}_{\mathcal{X}}) \tag{1}$$

$$y := v \fatsemi \mathbf{close}_{\mathcal{X}} = \mathbf{close}_{\mathcal{X}} \quad \text{when } y \preceq \mathcal{C}_{\mathcal{X}} \tag{2}$$

$$x := v \fatsemi \mathbf{close}_{\mathcal{X}} = \mathbf{close}_{\mathcal{X}} \fatsemi (x \mathbin{/} \mathcal{V}_{\mathcal{X}}) := (v \downharpoonright \mathcal{V}_{\mathcal{X}}) \quad \text{when } x \preceq \mathcal{V}_{\mathcal{X}}, \mathcal{C}_{\mathcal{X}} \sharp v \tag{3}$$

An assignment to a global variable x can be pushed into a variable block (1). We have to coerce both the variable and the assigned expression using lens composition and the state space extension operators, respectively. An assignment to a local variable $y \preceq \mathcal{C}_{\mathcal{X}}$ at the end of a block is lost (2). An assignment to a global variable in a block can be moved past the end (3). Again, it is necessary to coerce the variable and expression, using lens quotient and state space restriction, this time to contract the state space. Moreover, this law only applies when the expression does not refer to local variables, given by the condition $\mathcal{C}_{\mathcal{X}} \sharp v$. These latter two laws show how the symmetric lens allows us to distinguish local and global variables. We can also derive a Hoare logic law for variable blocks:

Theorem 8.7. If $\{\, p \upharpoonright \mathcal{V}_{\mathcal{X}} \,\}\, S \,\{\, q \upharpoonright \mathcal{V}_{\mathcal{X}} \,\}$ then $\{\, p \,\}\, \mathbf{open}_{\mathcal{X}} \fatsemi S \fatsemi \mathbf{close}_{\mathcal{X}} \,\{\, q \,\}$ 

The intuition is that p and q must be augmented with additional variables in the enlarged state space, and references to global variables must be type cast.

We can now model the algorithm in Example 8.1. First, we need to create global and local state spaces and a suitable symmetric lens. The global state space can be described by $\mathit{global} \triangleq [x : \mathit{int}, y : \mathit{int}]$, as explained in §3. This gives rise to $\mathbf{base} : \mathit{global} \Longrightarrow [\alpha]\mathit{global\text{-}ext}$ and $\mathbf{more} : \alpha \Longrightarrow [\alpha]\mathit{global\text{-}ext}$, which together form a symmetric lens $\mathbf{all}$. Moreover, the local state can be described by

```
alphabet global = x :: int  y :: int

abbreviation swap :: "global hrel" where
"swap ≡ var z :: int in all⇩L • z := x ;; x := y ;; y := &z"

lemma swap_wp: "swap wp (x = Y ∧ y = X) = U(y = Y ∧ x = X)"
  by (simp add: vblock_def wp usubst unrest)

lemma swap_hoare: "{x = X ∧ y = Y} swap {x = Y ∧ y = X}" by (rel_auto)

lemma swap_alt_def: "swap = (x, y) := (y, x)" by (rel_auto)
```

Fig. 1. Modelling *swap*, and its properties in Isabelle/UTP

the record $local \triangleq global + [z : int]$, and so we specialise *global*'s base lens, ***base***, to have type $global \Longrightarrow local$. In Isabelle/UTP, *local* can be generated on-the-fly in a record block, to support the syntax given in Example 8.1. This approach, using extensible records for variable blocks, is used in Simpl [17].

An implementation is shown in Figure 1, along with several theorems. We construct *global* using the ***alphabet*** command, and then define *swap*, with a near identical representation to Example 8.1. The decorations $\&z$ and $\boldsymbol{U}(\cdot)$ are hints to parser with no semantic content. The machinery for creating *local* and instantiating the symmetric lens is hidden behind the **var** construct, though we have to explicitly state that we are using the ***all*** symmetric lens from the global state space. We then prove three theorems. The first one calculates a weakest precondition, the second a Hoare triple, and the final one shows that *swap* can actually be replaced by a simultaneous assignment, assuming this is supported.

While the use of records in blocks provides strong typing, the fact that the ***all*** symmetric lens changes the type of the state space means it cannot be used in recursive functions. This was previously observed by Back and Preoteasa [18]. To handle recursion, the symmetric lens must describe a global state with the same type as the overall state space (which includes both global and local). As mentioned previously, (hd_A, tl_A) is such a lens, and so is its converse (tl_A, hd_A). This symmetric lens creates variable blocks that push an arbitrary value onto the start of a list, creating a stack semantics.

The fact that the list symmetric lens forms a partial lens creates the need for domain checks when variables in the list are accessed. Such checks can be avoided by using a state space with a function from natural numbers to values instead of a list. We define head and tail lenses on such a state space as follows:

$$hd^f_A \triangleq (A, \mathbb{N} \to A, \lambda f \bullet f\ 0, \lambda f\ v\ n \bullet v \lhd n = 0 \rhd f\ n)$$
$$tl^f_A \triangleq (\mathbb{N} \to A, \mathbb{N} \to A, \lambda f\ n \bullet f\ (n+1), \lambda f\ v\ n \bullet f\ 0 \lhd n = 0 \rhd f\ (n-1))$$

These head and tail lenses are total lenses, since they are defined on a total function. We can thus define a total symmetric lens using them in a similar way to the list symmetric lens. These lenses can also be lifted to a state space with additional global state in a similar way to the list symmetric lens. We have mechanised these definitions in Isabelle/UTP and proved the resultant lens is indeed a total symmetric lens. We have also proved properties for a swap function using this symmetric lens as we did for the record symmetric lens. This shows the flexibility of our lens-based approach to local variables: list or function lenses

```
definition insert_elem :: "int list ⇒ local hrel" where
"insert_elem xs =
  while (0 < j ∧ arr!j < arr!(j-1)) invr @(I xs)
  do
    (arr[j-1], arr[j]) := (arr!j, arr!(j-1)) ;; j := j - 1
  od"

abbreviation insertion_sort :: "int list ⇒ global hrel" where
"insertion_sort xs ≡
  arr := xs ;;
  open⇘lv⇙ ;;
  (i := 1 ;;
  while (i < length arr)
  invr 0 < i ∧ i ≤ length arr ∧ sorted(nths arr {0..i-1}) ∧ perm arr xs
  do
    j := i ;; insert_elem xs ;; i := i + 1
  od) ;;
  close⇘lv⇙"
```

Fig. 2. Insertion Sort in Isabelle/UTP

can be used where support for recursion is required, while record lenses can be used where the added structure of Isabelle's type system is desired.

9 Insertion Sort

Here, we show how we have used the collected results of the previous sections to verify the insertion sort algorithm in Isabelle/UTP. We model the algorithm using both collection lenses and symmetric lens variable blocks, as shown in Figure 2. In order to ease verification, we split the algorithm into two definitions: one for the inner loop (insert-elem), and one for the outer loop (insertion-sort). Both are specified as functions that take the list to be sorted, xs, as a parameter.

The syntax of the program broadly follows that given in Example 1.1. The only significant deviation is that we have manually constructed a symmetric lens lv that uses an explicit local state space. This is so that i and j can be referred to as global names in the Isabelle theory, to enable description of the invariants. The outer program itself operates on a state space where only arr is present, and the other variables are introduced by ***open*** and ***close***.

As usual [1], our loop construct supports invariant annotation, using the invr keyword. The invariant of the inner loop ($I\ xs$) is not shown due to its complexity. The outer loop invariant states that (1) $0 < i \leq \#arr$, it is within the array bounds; (2) the array in the range $0 \cdots i-1$ is sorted; and (3) arr is a permutation of the original list xs. The function $sorted : [A]list \to bool$ determines that a list is sorted by a predefined total order on A, and $perm : [A]list \to [A]list \to bool$ states two lists have the same elements, including repetitions. Both of the latter functions are provided as part of the Isabelle/HOL library. Function $nths : [A]list \to [nat]set \to [A]list$ gives the elements of a list described by an index set.

The program is verified using Hoare logic as shown in Figure 3. The proof is quite long, due to the number of proof obligations, and some manual effort is required. This seems mainly due to missing lemmas, and so in future the proof should be more automated (cf. [12]). Nevertheless, for now we omit details of the proof steps. The first Hoare triple demonstrates that the inner loop preserves

```
lemma insert_elem_correct: "
    {0 < i ∧ i < length arr ∧ j = i ∧ sorted(nths arr {0..i-1}) ∧ perm arr xs}
    insert_elem xs
    {0 < i ∧ i < length arr ∧ sorted(nths arr {0..i}) ∧ perm arr xs}"
proof -  [57 lines]

lemma insertion_sort_correct:
  "{true} insertion_sort xs {sorted(arr) ∧ perm arr xs}"
proof - [26 lines]
```

Fig. 3. Insertion Sort Verification

the invariant of the outer loop. The outer loop shows that, when provided with a non-empty list a sorted permutation of *xs* is returned in *arr*.

10 Conclusions

In this paper we have shown how lenses support modelling and verification of algorithms in the Isabelle/UTP tool [11]. We introduced dynamic lenses, that allow us to handle collections, and symmetric lenses, that allow partitioning of the state space into disjoint regions. Collection lenses allow us to generically characterise a variety of different indexed collection types in Isabelle/UTP, including arrays and maps. Symmetric lenses [16] allow us to characterise state partitioning, and we have used them here to distinguish local and global variables scopes. Due to typed nature of our state spaces, coercions are necessary when moving between scopes, which we can also handle using lenses. Our conclusion is that algebraic characterisation in this way is both flexible and practical, as our verification of insertion sort demonstrates. Moreover, since our characterisation sits at the state space level, our results are applicable to paradigms beyond imperative programming, such as reactive [11] and hybrid programming [4, 19].

In future work, we will explore symmetric lenses and their properties further. We note that there are several different models for symmetric lenses, including extensible records, lists, and total functions, each with unique advantages. We can use extensible records to support variables with native type checking, but they cannot support recursion, as for example required by quicksort, due to *a priori* bounding of the state space. In contrast, list and function symmetric lenses overcome this limitation, but require a fixed element type. Our mechanisation thus allows us chose the best model for a particular circumstance. In the future, we will perform a detailed comparison of the different models.

Acknowledgements. This work is funded by the EPSRC projects CyPhyAssure[3] (Grant EP/S001190/1) and RoboTest (Grant EP/R025479/1).

References

1. Armstrong, A., Gomes, V., Struth, G.: Building program construction and verification tools from algebraic principles. Formal Aspects of Computing **28**(2) (2015)
2. Dongol, B., Gomes, V., Struth, G.: A program construction and verification tool for separation logic. In: MPC 2015. LNCS 9129, Springer (2015) 137–158
3. Gomes, V.B.F., Struth, G.: Modal Kleene algebra applied to program correctness. In: 21st. Intl. Symp. on Formal Methods (FM). LNCS 9995, Springer (2016)

[3] CyPhyAssure Project: https://www.cs.york.ac.uk/circus/CyPhyAssure/

4. Huerta y Munive, J.J., Struth, G.: Verifying hybrid systems with modal Kleene algebra. In: RAMICS. LNCS 11194, Springer (October 2018)
5. Barnett, M., Chang, B.Y., DeLine, R., Jacobs, B., Leino, R.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO. LNCS 4111, Springer (2005)
6. Filliâtre, J.C., Paskevich, A.: Why3 – where programs meet provers. In: Programming Languages and Systems (ESOP). LNCS 7792, Springer (2013)
7. Kozen, D.: Kleene algebra with tests. ACM TOPLAS **19**(3) (May 1992)
8. Foster, J., Greenwald, M., Moore, J., Pierce, B., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. **29**(3) (May 2007)
9. Foster, J., Pilkiewicz, A., Pierce, B.: Quotient lenses. In: Proc. 13th Intl. Conf. on Functional Programming (ICFP), ACM (2008)
10. Foster, S., Zeyda, F., Woodcock, J.: Unifying heterogeneous state-spaces with lenses. In: ICTAC. LNCS 9965, Springer (2016) 295–314
11. Foster, S., Cavalcanti, A., Canham, S., Woodcock, J., Zeyda, F.: Unifying theories of reactive design contracts. Theoretical Computer Science **802** (September 2020)
12. Bockenek, J., Lammich, P., Nemouchi, Y., Wolff, B.: Using Isabelle/UTP for the verification of sorting algorithms. In: Proc. Isabelle Workshop (FLoC). (July 2018)
13. Back, R.J., Wright, J.: Refinement Calculus: A Systematic Introduction. Springer (1998)
14. Foster, S., Zeyda, F., Woodcock, J.: Isabelle/UTP: A mechanised theory engineering framework. In: UTP. LNCS 8963, Springer (2014) 21–41
15. Foster, S., Zeyda, F., Nemouchi, Y., Ribeiro, P., Wolff, B.: Isabelle/UTP: Mechanised Theory Engineering for Unifying Theories of Programming. Archive of Formal Proofs (2019) https://www.isa-afp.org/entries/UTP.html.
16. Hofmann, M., Pierce, B., Wagner, D.: Symmetric lenses. In: Proc. 38th Intl. Symp. on Principles of Programming Languages (POPL), IEEE (2011) 371–384
17. Alkassar, E., Hillebrand, M., Leinenbach, D., Schirmer, N., Starostin, A.: The Verisoft approach to systems verification. In: VSTTE. LNCS 5295, Springer (2008)
18. Back, R.J., Preoteasa, V.: An algebraic treatment of procedure refinement to support mechanical verification. Formal Aspects of Computing **17**(1) (2005)
19. Foster, S.: Hybrid relations in Isabelle/UTP. In: UTP. LNCS 11885, Springer (2019) 130–153
20. Feliachi, A., Gaudel, M.C., Wolff, B.: Unifying theories in Isabelle/HOL. In: UTP. LNCS 6445, Springer (2010) 188–206
21. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall (1998)
22. Schirmer, N., Wenzel, M.: State spaces – the locale way. In: SSV 2009. Volume 254 of ENTCS. (2009) 161–179
23. Greenaway, G., Lim, J., Andronick, J., Klein, G.: Don't sweat the small stuff: Formal verification of C code without the pain. In: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM (June 2014)
24. Dongol, B., Hayes, I., Meinicke, L., Struth, G.: Cylindric Kleene lattices for program construction. In: MPC. LNCS 11825, Springer (2019) 192–225
25. Tarski, A.: On the calculus of relations. J. Symbolic Logic **6**(3) (1941) 73–89
26. Hoare, C.A.R., Hayes, I., He, J., Morgan, C., Roscoe, A., Sanders, J., Sørensen, I., Spivey, J., Sufrin, B.: The laws of programming. Commun. ACM **30**(8) (1987)
27. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10) (1969) 576–580
28. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM **18**(8) (1975) 453–457