A Program Construction and Verification Tool for Separation Logic

Brijesh Dongol[1], Victor B. F. Gomes[2], and Georg Struth[2]

[1] Department of Computer Science, Brunel University
Brijesh.Dongol@brunel.ac.uk
[2] Department of Computer Science, University of Sheffield
{v.gomes,g.struth}@sheffield.ac.uk

Abstract. An algebraic approach to the design of program construction and verification tools is applied to separation logic. The control-flow level is modelled by power series with convolution as separating conjunction. A generic construction lifts resource monoids to assertion and predicate transformer quantales. The data domain is captured by concrete store-heap models. These are linked to the separation algebra by soundness proofs. Verification conditions and transformation or refinement laws are derived by equational reasoning within the predicate transformer quantale. This separation of concerns makes an implementation in the Isabelle/HOL proof assistant simple and highly automatic. The resulting tool is itself correct by construction; it is explained on three simple examples.

1 Introduction

Separation logic is an approach to program verification that has received considerable attention over the last decade. It is designed for local reasoning about a system's states or resources, by isolating the part of a system that is affected by an action from the remainder. This capability is provided by its separating conjunction operator together with the frame inference rule, which makes local reasoning modular. A key application is the verification of programs with pointers [35, 32]; but the method has also been used for modular concurrency verification [30, 38, 6].

Separation logic is currently supported by a large number of tools, some of which are discussed in Section 9. Implementations in higher-order interactive proof assistants [40, 37, 8, 23] are particularly relevant to this article. In comparison to automated tools or tools for decidable fragments, they can express more program properties, but are less effective for proof search. Ultimately, an integration of these different proof methods is desirable.

This article adds to this tool chain (and presents another implementation in the Isabelle/HOL theorem-proving environment [29]). However, our approach is different in several respects. It focusses almost entirely on making the control-flow layer as simple as possible and on separating it cleanly from the data layer. This supports the integration of various data models and modular reasoning about these two layers, with assignment laws providing an interface. To achieve this separation of concerns, we develop an algebraic approach to separation logic which aims to combine the simplicity of the original

logical approaches [32] with the abstractness and elegance of O'Hearn and Pym's categorical logic of bunched implications [31] in a way suitable for formalisation in Isabelle. Our approach is based on *power series* [15], which have found applications in formal language and automata theory [5, 16]. Their use in the context of separation logic is a contribution in itself.

In a nutshell, a *power series* is a function $f : M \to Q$ from a partial monoid M into a quantale Q. Defining addition of power series by lifting addition pointwise from the quantale, and multiplication as *convolution*

$$(f \otimes g)\, x = \sum_{x=y\circ z} f\, y \odot g\, z,$$

where $\cdot$ acts on M, $\odot$ on Q and $\otimes$ on Q^M, it turns out that the function space Q^M of power series itself forms a quantale [15]. If M is commutative (a *resource monoid* [7]) and Q formed by the booleans $\mathbb{B}$ (with $\odot$ as meet), one can interpret power series as assertions or predicates over M. Separating conjunction then arises as a special case of convolution, and, in fact, as a language product over resources. The function space $\mathbb{B}^M$ is the assertion quantale of separation logic. The approach generalises to power series over program states modelled by store-heap pairs, which is needed for programming applications.

Using lifting results for power series again, we construct the quantale-like algebraic semantics of predicate transformers over assertion quantales in the style of boolean algebras with operators, following a previous approach by O'Hearn and Yang [41]. We characterise the monotone predicate transformers and derive the inference rules of Hoare logic for partial correctness (without assignment) within this subalgebra. We also derive the frame rule of separation logic on the subalgebra of local monotone predicate transformers. We use these rules for automated verification condition generation. Formalising Morgan's specification statement [27] on the transformer quantale yields tools for program construction and refinement with a frame refinement law with minimal effort. The predicate transformer approach, instead of the more common state transformer one [7], fits well into the power series approach and simplifies the development.

The formalisation of the algebraic hierarchy from resource monoids to predicate transformer algebras benefits from the functional programming approach imposed by Isabelle and its integration of automated theorem provers and SMT-solvers via the Sledgehammer tool. These are optimised for equational reasoning, which makes the entire development highly automatic. In addition, Isabelle's reconstruction of proof outputs provided by the external tools makes our tools correct by construction.

At the data-domain level, we currently use Isabelle's extant infrastructure for the store, the heap and pointer-based data structures. An interface to the control-flow algebra is provided by the standard assignment laws of separation logic and their refinement counterparts. Isabelle's concrete data domain models are linked formally with our abstract separation algebra by soundness proofs. Algebraic facts are then picked up automatically by Isabelle for reasoning in the concrete model. Our verification examples show that, at the concrete layer, proofs may require some user interaction, but an integration of domain-specific provers and solvers for the data domain is an avenue of future work. The entire technical development has been formalised in Isabelle; all proofs have

been formally verified. We therefore show only some example proofs which demonstrate the simplicity of algebraic reasoning. The complete executable Isabelle theories can be found online[3].

2 Partial Monoids, Quantales and Power Series

This section presents the algebraic structures that underlie our approach to separation logic. Further details on power series and lifting constructions can be found in [15].

A *partial semigroup* is defined as a set S with a composition $\cdot : D \to S$ for some $D \subseteq S \times S$ that satisfies the usual associativity law in the sense that if either side is defined then so is the other side and both are equal [4]. A *partial monoid* M is an obvious extension by an unit 1 such that $x \cdot 1 = x$ for all $(x, 1) \in D$ and $1 \cdot x = x$ for all $(1, x) \in D$. A partial monoid M is *commutative* if $x \cdot y = y \cdot x$ for all $(x, y) \in D$. Henceforth $\cdot$ is used for a general and $*$ for a commutative multiplication.

A *quantale* (or *standard Kleene algebra*) is a structure $(Q, \leq, \cdot, 1)$ such that $(Q, \leq)$ is a complete lattice, $(Q, \cdot, 1)$ is a monoid and the distributivity axioms

$$(\sum_{i \in I} x_i) \cdot y = \sum_{i \in I} x_i \cdot y, \qquad x \cdot (\sum_{i \in I} y_i) = \sum_{i \in I} (x \cdot y_i)$$

hold, where $\sum X$ denotes the supremum of a set $X \subseteq Q$. Similarly, we write $\bigsqcap X$ for the infimum of X, and 0 for the least and U for the greatest element of the lattice. The monotonicity laws

$$x \leq y \Rightarrow z \cdot x \leq z \cdot y, \qquad x \leq y \Rightarrow x \cdot z \leq y \cdot z$$

follow from distributivity. The two annihilation laws $x \cdot 0 = 0 = 0 \cdot x$ follow from $\sum_{i \in \emptyset} x_i = \sum \emptyset = 0$. A quantale is *commutative* and *partial* if the underlying monoid is. It is *distributive* if

$$x \sqcap (\sum_{i \in I} y_i) = \sum_{i \in I} (x \sqcap y_i), \qquad x + (\bigsqcap_{i \in I} y_i) = \bigsqcap_{i \in I} (x + y_i).$$

A *boolean quantale* is a complemented distributive quantale. The boolean quantale $\mathbb{B}$ of the booleans, where multiplication coincides with join, is an important example.

We call a *power series* a function $f : M \to Q$, from a partial monoid M into a quantale Q. For $f, g : M \to Q$ and a family of functions $f_i : M \to Q$, $i \in I$ we define

$$(f \cdot g)\ x = \sum_{x = y \cdot z} f\ y \cdot g\ z, \qquad (\sum_{i \in I} f_i)\ x = \sum_{i \in I} f_i\ x,$$

where $y, z \in M$. The composition $f \cdot g$ is called *convolution*; the multiplication symbol is often overloaded to be used on M, Q and the function space Q^M.

The idea behind convolution is simple: element x is split into y and z, the functions f and g are applied in parallel to y and z to calculate the values $f\ y$ and $g\ z$, and their

[3] `https://github.com/vborgesfer/sep-logic`

results are composed to form a value for the summation with respect to all possible splits of x.

Because x ranges over M, the constant $\bot \notin M$ is excluded as a value; undefined splittings of x do not contribute to convolutions. In addition, $(f+g)\ x = f\ x + g\ x$ arises as a special case of the supremum. Finally, we define the power series $\mathbb{0} : M \to Q$ and $\mathbb{1} : M \to Q$ as

$$\mathbb{0} = \lambda x.\ 0, \qquad \mathbb{1} = \lambda x.\ \begin{cases} 1, & \text{if } x = 1, \\ 0, & \text{otherwise.} \end{cases}$$

The quantale structure lifts from Q to the function space Q^M of power series.

Theorem 2.1 ([15]). *Let M be a partial monoid. If Q is a boolean quantale, then so is $(Q^M, \leq, \cdot, \mathbb{1})$. If M and Q are commutative, then so is Q^M.*

The power series approach generalises from one to n dimensions [15]. For separation logic, the two-dimensional case with power series $f : S \times M \to Q$ from set S and partial commutative monoid M into the commutative quantale Q is needed. Now

$$(f * g)\ (x, y) = \sum_{y = y_1 * y_2} f\ (x, y_1) * g\ (x, y_2), \qquad (\sum_{i \in I} f_i)\ (x, y) = \sum_{i \in I} f_i\ (x, y).$$

The convolution $f * g$ acts solely on the second coordinate. Finally, we define two-dimensional units as

$$\mathbb{0} = \lambda x, y.\ 0, \qquad \mathbb{1} = \lambda x, y.\ \begin{cases} 1, & \text{if } y = 1, \\ 0, & \text{otherwise.} \end{cases}$$

Theorem 2.2 ([15]). *Let S be a set and M a partial commutative monoid. If Q is a commutative boolean quantale, then so is $Q^{S \times M}$.*

We have implemented partial monoids and quantales by using Isabelle's type class and locale infrastructure, building on existing libraries for monoids, quantales and complete lattices. The implementation of power series uses Isabelle's well developed libraries for functions. This makes proofs in this setting simple and highly automatic.

3 Assertion Quantales

In language theory, power series have been introduced for modelling formal languages. Here, M is the free monoid X^* and Q can be taken as a semiring $(Q, +, \cdot, 0, 1)$, because there are only finitely many ways of splitting words into prefix/suffix pairs in convolutions. Infinite suprema in the definition of convolution are therefore not needed. In the particular case of the boolean semiring $\mathbb{B}$, where composition $\cdot$ is meet $\sqcap$, power series $f : X^* \to \mathbb{B}$ are interpreted as characteristic functions (i.e., predicates) that indicate whether a word is in a set. In this case, sets are languages, and hence, convolution specialises to

$$(f \cdot g)\ x = \sum_{x = yz} f\ x \sqcap g\ y,$$

identifying predicates with their extensions to the language product

$$p \cdot q = \{yz \mid y \in p \wedge z \in q\}.$$

More generally, we consider power series $S \to \mathbb{B}$ from a partial monoid S into the boolean quantale $\mathbb{B}$ and set up a connection with separation logic. There, one is interested in modelling assertions or predicates over the memory heap. The heap can be represented abstractly by a so-called *resource monoid* [7], which is a partial commutative monoid. By analogy to the language case, an assertion p of separation logic is a boolean-valued function from a resource monoid M, hence a power series $p : M \to \mathbb{B}$. Thus Theorem 2.1 applies.

Corollary 3.1. *The assertions $\mathbb{B}^M$ over resource monoid M form a commutative boolean quantale with convolution as separating conjunction.*

The logical structure of the assertion quantale $\mathbb{B}^M$ is as follows. The predicate $\mathbb{0}$ is a contradiction whereas $\mathbb{1}$ holds of the empty resource 1 and is false otherwise. The operations $\sum$ and $\bigsqcap$ correspond to existential and universal quantification; their finite cases yield conjunctions and disjunctions. The order $\leq$ is implication. Convolution becomes

$$(p * q)\, x = \sum_{x=y*z} p\, y \sqcap q\, z,$$

and it gives a simple algebraic account of separating conjunction. By $x = y*z$, resource x is separated into resources y and z. By $p\ y \sqcap q\ z$, the value of predicate p on y is conjoined with that of q on z. Finally, the supremum is true if one of the conjunctions holds for some splitting of x.

As for languages, one can again identify predicates with their extensions. Then

$$p * q = \{y * z \in M \mid y \in p \wedge z \in q\},$$

and separating conjunction becomes a language product over resources (cf. [21]). The analogy to language theory is even more striking when considering the paradigmatic kind of resource: multisets or Parikh vectors over a finite set X. These form the free commutative monoids over X.

Applications of separation logic, however, require program states which are store-heap pairs. Now Theorem 2.2 applies.

Corollary 3.2. *The assertions $\mathbb{B}^{S\times M}$ over set S (the store) and resource monoid M form a commutative boolean quantale with convolution as separating conjunction. For all $p, q : S \times M \to \mathbb{B}$, $s \in S$ and $h \in M$,*

$$(p * q)\ (s, h) = \sum_{h=h_1*h_2} p\ (s, h_1) \sqcap\ q\ (s, h_2).$$

Written in language-product style, therefore,

$$p * q = \{(s, h_1 * h_2) \in S \times M \mid (s, h_1) \in p \wedge (s, h_2) \in q\}.$$

The definition of convolution and the associated lifting is obviously flexible enough to encompass situations where pairs are extended to tuples or where the store as well as the heap are split by convolution. Isabelle also supports uncurried representations of such tuples and translations between them. The constructive functional approach of power series is very convenient for the functional programming style of Isabelle.

Quantales carry a rich algebraic structure. Their distributivity laws give rise to continuity or co-continuity properties. Therefore, many functions constructed from the quantale operations have adjoints as well as fixpoints, which can be iterated to the first limit ordinal. This is well known in denotational semantics and important for our approach to program verification. In particular, separating conjunction $*$ distributes over arbitrary suprema in $\mathbb{B}^M$ and $\mathbb{B}^{S\times M}$ and therefore has an upper adjoint: the *magic wand* operation $-\!\!*$, which is widely used in separation logic. In the quantale setting, the adjunction gives us theorems for the magic wand for free. This and other residuals that arise on the assertion quantale of separation logic have been studied, for instance, in [9].

One can think of the power series approach to separation logic as a simpler account of the category-theoretical approach in O'Hearn and Pym's logic of bunched implication [31] in which convolution generalises to coends and the quantale lifting is embodied by Day's construction [12]. For the design of verification tools and our implementation in Isabelle, the simplicity of the power series approach is certainly an advantage.

4 Predicate Transformer Quantales

Our algebraic approach to separation logic is based on predicate transformers (cf. [3]). This is in contrast to most previous state-transformer-based approaches and implementations [7, 23, 37], with [41, 20] being exceptions. First of all, predicate transformers are more amenable to algebraic reasoning [3]—simply because their source and target types are both at powerset level. Second, the approach is coherent and easily implementable within our framework. Predicate transformers can be seen once more as power series and instances of Theorem 2.1 describe their algebras.

A *state transformer* $f_R : A \to 2^B$ is often associated with a relation $R \subseteq A \times B$ from set A to set B by defining

$$f_R\, a = \{b \mid (a, b) \in R\}.$$

It can be lifted to a function $\langle R\rangle : 2^A \to 2^B$ defined by

$$\langle R\rangle X = \bigcup_{a\in X} f_R\, a$$

for all $X \subseteq A$. More importantly, state transformers are lifted to *predicate transformers* $[R] : 2^B \to 2^A$ by defining

$$[R]\, Y = \{x \mid f_R\, x \subseteq Y\}$$

for all $Y \subseteq B$. The modal box and diamond notation is justified by the correspondence between diamond operators and Hoare triples as well as box operators and weakest

liberal precondition operators in the context of modal semirings and modal Kleene algebras [26]. In fact we obtain the adjunction

$$\langle R\rangle X \subseteq Y \Leftrightarrow X \subseteq [R]Y$$

from the above definitions.

Predicate transformers in $(2^A)^{2^B}$ form complete distributive lattices [3]. In the power series setting, this follows from Theorem 2.1 in two steps, ignoring the monoidal structure. Since $\mathbb{B}$ forms a complete distributive lattice, so do $2^B \cong \mathbb{B}^B$ and $2^A \cong \mathbb{B}^A$ in the first step, and so does $(2^A)^{2^B}$ in the second one.

In addition, predicate transformers in $(2^A)^{2^A}$ form a monoid under function composition with the identity function as the unit. Such predicate transformers form a distributive *near-quantale*, which is a quantale such that the left distributivity law, $x \cdot (\sum_{i\in I} y_i) = \sum_{i\in I}(x \cdot y_i)$, need not hold. The monotone predicate transformers in $(2^A)^{2^A}$, which satisfy $p \leq q \Rightarrow f\ p \leq f\ q$, form a distributive *pre-quantale* [3], which is a near-quantale in which the left monotonicity law, $x \leq y \Rightarrow z \cdot x \leq z \cdot y$, holds. In these cases, the monoidal parts of the lifting are not obtained with the power series technique. The monoidal operation on predicate transformers is function composition and not convolution. Of course it is associative and the identity function is a unit of composition.

Adapting these results to separation logic requires the consideration of assertion quantales $\mathbb{B}^M$ or $\mathbb{B}^{S\times M}$ with store S and resource monoid M instead of the powerset algebra over a set A. Instead of lifting these quantales, we only lift their boolean algebra reduct, disregarding separation conjuction by not lifting it to a convolution on predicate transformers, which is not needed for separation logic. The quantale structure of predicate transformers is again obtained by considering function composition as the monoidal operation. This yields the following result.

Theorem 4.1. *Let S be a set, M a resource monoid and $\mathbb{B}^{S\times M}$ an assertion quantale. The monotone predicate transformers over $\mathbb{B}^{S\times M}$ form a distributive pre-quantale.*

The proof consists of showing that the predicate transformers over $\mathbb{B}^{S\times M}$ form a near quantale and checking that the monotone predicate transformers form a subalgebra of this near-quantale. In fact, the unit predicate transformer has to be monotone—which is the case—and the quantale operations of suprema, infima and composition have to preserve monotonicity. This is implied by properties such as

$$[R \cup S] = [R] \sqcap [S], \qquad [R; S] = [R] \cdot [S] = \lambda x.\ [R]\ ([S]\ x).$$

Monotone predicate transformers can be used to derive the standard inference rules of Hoare logic as verification conditions (Section 5) and the usual rules of Morgan's refinement calculus (Section 6). Derivation of the frame rule of separation logic, however, requires a smaller class of predicate transformers defined as follows.

A state transformer f is *local* [7] if

$$f\ (x * y) \leq (f\ x) * \{y\}$$

whenever $x * y$ is defined. Intuitively, this means that the effect of such a state transformer is restricted to a part of the heap; see [7] for further discussion. Analogously, and similarly to [20], we call a predicate transformer F *local* if

$$(F\ p) * q \leq F\ (p * q).$$

It is easy to show that the two definitions are compatible.

Lemma 4.2. *State transformer f_R is local iff predicate transformer $[R]$ is local.*

The final theorem in this section establishes the local monotone predicate transformers as a suitable algebraic framework for separation logic.

Theorem 4.3. *Let S be a set and M a resource monoid. The local monotone predicate transformers over the assertion quantale $\mathbb{B}^{S \times M}$ form a distributive pre-quantale.*

The fact that the local monotone predicate transformers form a complete lattice has been observed previously [41]. Once again it must be checked that the zero predicate transformer is local—which is the case—and that the quantale operations preserve locality and monotonicity.

We have implemented the whole approach in Isabelle; all theorems have been formally verified, mainly using Theorem 2.1 for the lifting to predicate transformers. Apart from the local case, ours is not the first Isabelle formalisation of predicate transformers; it is based on previous work by Preoteasa [34].

5 Verification Conditions

The pre-quantale of local monotone predicate transformers supports the derivation of verification conditions by equational reasoning. A standard set of such conditions are the inference rules of Hoare logic. For sequential programs, Hoare logic provides one inference rule per program construct, and these can be applied by and large in non-deterministic fashion to simple while-programs. This suffices to eliminate the control structure of a program and generate verification conditions at the data level.

The quantale setting also guarantees that the finite iteration F^* of a predicate transformer is well defined. Writing $\mathbf{skip}$ for the quantale unit (the identity function) we obtain the greatest fixpoint laws

$$\mathbf{skip} \sqcap F \cdot F^* = F^*, \qquad G \leq H \sqcap F \cdot G \Rightarrow G \leq F^* \cdot H,$$
$$\mathbf{skip} \sqcap F^* \cdot F = F^*, \qquad G \leq H \sqcap G \cdot F \Rightarrow G \leq H \cdot F^*,$$

from the explicit definition

$$F^* = \bigsqcap_{i \in \mathbb{N}} F^i$$

by iteration to the first infinite ordinal where $F^0 = \mathbf{skip}$ and $F^{n+1} = F \cdot F^n$, as usual. This supports a shallow algebraic embedding of a simple while language with the standard intermediate language for the verification of while-programs.

First we lift predicates to predicate transformers [3]:

$$[p] = \lambda q.\, \overline{p} + q,$$

where $\overline{p}$ denotes the boolean complement of p. With predicates modelled as relational subidentities, this definition is justified by the lifting from the previous section: $(s, s) \in [p]\ q$ iff $(s, s) \in p \Rightarrow (s, s) \in q$.

Second, we change notation to use descriptive while program syntax for predicate transformers. We now write ; for function composition, and we encode the algebraic semantics of conditionals and while loops as

$$\mathbf{if}\ p\ \mathbf{then}\ F\ \mathbf{else}\ G\ \mathbf{fi} = [p] \cdot F \sqcap [\overline{p}] \cdot G,$$
$$\mathbf{while}\ p\ \mathbf{do}\ F\ \mathbf{od} = ([p] \cdot F)^* \cdot [\overline{p}].$$

Third, we provide the usual assertions notation for programs via Hoare triple syntax:

$$\{p\}\ F\ \{q\} \Leftrightarrow p \leq F\ q.$$

Box notation shows that

$$\{p\}\ [R]\ \{q\} \Leftrightarrow p \leq [R]q$$

for relational program R. Thus $[R]q = wlp(R, q)$ is the well-known weakest liberal precondition of program R and postcondition q. This encoding is standard in the context of Kleene algebras with domain [26]. It also explains our slight abuse of relational or imperative notation for predicate transformers: e.g. we write $[R]; [S]$ instead of $[R] \cdot [S]$ because the latter expression is equal to $[R; S]$, as indicated in the previous section.

Proposition 5.1. *Let $p, q, r, p', q' \in \mathbb{B}^{S \times M}$ be predicates. Let F, G, H be monotone predicate transformers over $\mathbb{B}^{S \times M}$, with H being local. Then the rules of propositional Hoare logic (no assignment rule) and the frame rule of separation logic are derivable.*

$$\begin{aligned}
p \leq q\ &\Rightarrow \{p\}\ \mathbf{skip}\ \{q\},\\
p \leq p' \wedge q' \leq q\ \wedge \{p'\}\ F\ \{q'\}\ &\Rightarrow \{p\}\ F\ \{q\},\\
\{p\}\ F\ \{r\}\ \wedge \{r\}\ G\ \{q\}\ &\Rightarrow \{p\}\ F; G\ \{q\},\\
\{p \sqcap r\}\ F\ \{G\}\ \wedge \{p \sqcap \overline{r}\}\ G\ \{q\}\ &\Rightarrow \{p\}\ \mathbf{if}\ r\ \mathbf{then}\ F\ \mathbf{else}\ G\ \mathbf{fi}\ \{q\},\\
\{p \sqcap q\}\ F\ \{p\}\ &\Rightarrow \{p\}\ \mathbf{while}\ q\ \mathbf{do}\ F\ \mathbf{od}\ \{\overline{b} \sqcap p\},\\
\{p\}\ H\ \{q\} &\Rightarrow \{p * r\}\ H\ \{q * r\}.
\end{aligned}$$

Proof. We derive the frame rule as an example. Suppose $p \leq H\ q$. Then, by isotonicity of $*$ and locality, $p * r \leq (H\ q) * r \leq H(q * r)$. □

The remaining derivations are just as simple and fully automatic in Isabelle. In fact, these laws can be derived within the predicate transformer quantale [26, 20], but pragmatically, in the context of verification condition generation, this abstraction leads to less applicable Isabelle tactics.

Beyond the simple fixpoints studied in this section, the quantale setting guarantees the existence of least fixpoints of arbitrary monotone functions. A verification condition

for parameterless recursive procedures can therefore be derived as well, supporting the verification of more general recursive programs. A Hoare logic for recursive imperative programs has already been implemented in the quantale setting in Isabelle[4], however, it remains to be combined with the assertion quantale of separation logic.

6 Refinement Laws

To demonstrate the power of the predicate transformer approach to separation logic we now outline its applicability to local reasoning in program construction and transformation. We show that the standard laws of Morgan's refinement calculus [27] plus an additional framing law for resources can be derived and programmed in Isabelle with little effort. It only requires defining one single additional concept—Morgan's specification statement—which is definable in every predicate transformer quantale.

Formally, for predicates $p, q \in \mathbb{B}^{S \times M}$, we define the *specification statement* as

$$[\![p, q]\!] = \sum \{F \mid p \leq F\ q\}.$$

It models the most general predicate transformer or program that links postcondition q with precondition p. It is easy to see that

$$\{p\}\ F\ \{q\} \Leftrightarrow F \leq [\![p, q]\!],$$

which entails the characteristic properties

$$\{p\}\ [\![p, q]\!]\ \{q\}, \qquad \{p\}\ F\ \{q\} \Rightarrow F \leq [\![p, q]\!]$$

of the specification statement: program $[\![p, q]\!]$ relates precondition p with postcondition q whenever it terminates; and it is the largest program with that property. It is easy to check that specification statements over the pre-quantale of local monotone predicate transformers are themselves local and monotone.

Like Hoare logic, Morgan's basic refinement calculus provides one refinement law per program construct. Once more we ignore assignments at this stage. We also switch to standard refinement notation with refinement order $\sqsubseteq$ being the converse of $\leq$.

Proposition 6.1. *For $p, q, r, p', q' \in \mathbb{B}^{S \times M}$, and predicate transformer F the following refinement laws are derivable in the algebra of local monotone predicate transformers.*

$$\begin{aligned}
p \leq q \Rightarrow [\![p, q]\!] &\sqsubseteq \mathbf{skip}, \\
p' \leq p \wedge q \leq q' \Rightarrow [\![p, q]\!] &\sqsubseteq [\![p', q']\!], \\
[\![p, q]\!] &\sqsubseteq [\![p, r]\!]; [\![r, q]\!], \\
[\![p, q]\!] &\sqsubseteq \mathbf{if}\ b\ \mathbf{then}\ [\![b \sqcap p, q]\!]\ \mathbf{else}\ [\![\bar{b} \sqcap p, q]\!]\ \mathbf{fi}, \\
[\![p, \bar{b} \sqcap p]\!] &\sqsubseteq \mathbf{while}\ b\ \mathbf{do}\ [\![b \sqcap p, p]\!]\ \mathbf{od}, \\
[\![p * r, q * r]\!] &\sqsubseteq [\![p, q]\!], \\
[\![0, 1]\!] &\sqsubseteq F, \\
F &\sqsubseteq [\![1, 0]\!].
\end{aligned}$$

[4] `http://www.dcs.shef.ac.uk/~victor/verification`

Proof. Using the frame rule, we derive the framing law, the sixth law in Proposition 6.1, as an example:

$$\{p\}\ [\![p, q]\!]\ \{q\} \Rightarrow \{p * r\}\ [\![p, q]\!]\ \{q * r\} \Leftrightarrow [\![p * r, q * r]\!] \sqsubseteq [\![p, q]\!].$$

The first step uses the frame rule from Proposition 5.1, the second one the Galois connection for the specification statement. The proofs of the other refinement laws are equally simple, using the corresponding Hoare rules in their proofs. They are fully automatic in Isabelle. A refinement law for recursive programs can be derived as well. □

The entire theory hierarchy discussed so far, from partial monoids to predicate transformer quantales, has been formalised in modular fashion as algebraic components in Isabelle/HOL, much of which was highly automatic and required only a moderate effort. It benefits, to a large extent, from Isabelle's integrated first-order theorem proving, SMT-solving and counterexample generation technology. These tools are highly optimised for equational reasoning, interacting efficiently with the algebraic layer.

7 Data Domain Integration

This section describes the integration of the data domain layer into our Isabelle tools for program construction and verification. It uses an important Isabelle feature, namely that the mathematical structures formalised in Isabelle are all polymorphic. We can therefore instantiate the abstract algebras for the control flow with various concrete models by soundness proofs, that is, quantales with predicate transformers, predicate transformers with binary relations and functions which update program states. In particular, abstract resource monoids are linked with various concrete models for resources, including store-heap pairs.

Our data domain integration can build on excellent Isabelle libraries and decades-long experience in reasoning with functions and relations, all sorts of data structures and data types. In particular, for program construction and verification with separation logic, Isabelle already provides support for reasoning with pointers and the heap [25, 40]. This is predominantly based on set theory.

As previously mentioned, program states in separation logic are store-heap pairs (s, h). Program stores are implemented in Isabelle as records of program variables, each of which has a *retrieve* and an *update* function. On the one hand, this approach is polymorphic and supports variables of any Isabelle type. For instance, Isabelle's built-in list data type and list libraries can be used to reason about list-based programs. On the other hand, Isabelle records are static, which makes it difficult to accommodate dynamic features such as variable scoping, as considered in the framing laws of Morgan's refinement calculus.

Heaps have been modelled in Isabelle as partial functions on $\mathbb{N}$ [25, 40]; they therefore have type $\mathbf{nat} \to \mathbf{nat\ option}$.

We implement assignments first as functions from states to states,

$$(`x := e) = \lambda(s, h).\ (x\_update\ s\ e, h),$$

where $`x$ is a program variable, $x\_update$ the update function for $`x$, (s, h) a state and e an evaluated expression of the same type as $`x$.

Next, we implement the typical commands for heap manipulation of separation logic. For heap allocation, we use Hilbert's ε operator, where $\varepsilon\ x.\ P\ x$ denotes some x such that $P\ x$ provided it exists. Heap allocation is then programmed as

$$(`x := \mathbf{cons}\ e) = \lambda(s, h).\ \mathbf{let}\ n = \varepsilon\ y.\ (\forall x \in dom\ h.\ x < y\ )$$
$$\mathbf{in}\ (x\_update\ s\ n, h[n \mapsto e]),$$

where $dom\ h$ is the domain of the heap h, expression e is of natural number type, and $h[n \mapsto e]$ maps n to e and is the same as h for all other parameters, namely, $h[n \mapsto e] = \lambda m.\ \mathbf{if}\ m = n\ \mathbf{then}\ e\ \mathbf{else}\ h(m)$. In a similar fashion, we implement deallocation and mutation as

$$(\mathbf{dispose}\ e) = \lambda(s, h).\ (s, h[e := None]),$$
$$(@e := e') = \lambda(s, h).\ (s, h[e \mapsto e']),$$

where the expressions e and e' evaluate to natural numbers and $h[e := None]$ removes e from the domain of h.

We lift these atomic commands to predicate transformers as

$$[f] = \lambda q.\ q \cdot f,$$

where $\cdot$ denotes function composition, as usual. This definition is consistent with the definition of lifting in Section 4. As previously, we generally do not write the lifting brackets explicitly, identifying program pseudocode with predicate transformers to simplify verification notation. It then remains to show that these atomic commands are local and monotonic.

With this infrastructure in place we can prove Hoare's assignment rule and Reynolds' local axioms for allocation, deallocation and mutation of separation logic [35] in the concrete heap model. We write $q[e/`x]$ for the substitution of variable $`x$ by expression e in q, write $e \mapsto e'$ for the singleton heap mapping e to e', write $e \mapsto -$ for the singleton heap mapping e to any value, and write $\mathbf{emp}$ for the empty heap.

Proposition 7.1. *Suppose $p \leq q[e/`x]$ holds. Then the following local rules are derivable in the store-heap model:*

$$\{p\}\ `x := e\ \{q\},$$
$$\{\mathbf{emp}\}\ `x := \mathbf{cons}\ e\ \{`x \mapsto e\},$$
$$\{e \mapsto -\}\ \mathbf{dispose}\ e\ \{\mathbf{emp}\},$$
$$\{e \mapsto -\}\ @e := e'\ \{e \mapsto e'\}.$$

Variants of these rules can easily be derived. For example, global rules, obtained by a simple application of the frame rule, emphasise that anything in the heap different from the mutated location is left unchanged:

$$\{r\}\ `x := \mathbf{cons}\ e\ \ \{(`x \mapsto e) * r\},$$
$$\{(e \mapsto -) * r\}\ \ \mathbf{dispose}\ e\ \{r\},$$
$$\{(e \mapsto -) * r\}\ @e := e'\ \{(e \mapsto e') * r\}.$$

Applying the weakening rule and the identity $p * (p \mathrel{-\!\!*} q) \leq q$ yields another global mutation rule for backward reasoning, which is more suitable for automation

$$\{(e \mapsto -) * ((e \mapsto e') \mathrel{-\!\!*} q)\} \ @e := e' \ \{q\}.$$

Note that the magic wand operation $\mathrel{-\!\!*}$ has been discussed briefly at the end of Section 3. The resulting set of control-flow and data-domain inference rules for separation logic allows us to program the Isabelle proof tactic *hoare*, which generates verification conditions automatically and eliminates the entire control structure when the invariants of while loops are annotated.

One can also use the assignment rules to derive their refinement counterparts:

$$\begin{aligned} p \leq q[e/`x] &\Rightarrow [\![p, q]\!] \sqsubseteq (`x := e), \\ q' \leq q[e/`x] &\Rightarrow [\![p, q]\!] \sqsubseteq [\![p, q']\!]; \ (`x := e), \\ p' \leq p[e/`x] &\Rightarrow [\![p, q]\!] \sqsubseteq (`x := e); \ [\![p', q]\!]. \end{aligned}$$

The second and third laws are called the *following* and *leading* refinement law for assignments [27]. They are useful for program construction. We have derived analogous laws for heap allocation, deallocation and mutation. We have also programmed the tactic *refinement*, which automatically tries to apply all the rules of this refinement calculus in construction steps of pointer programs.

8 Examples

To show our approach at work, we present three examples, among them the obligatory correctness proof of the classical in situ linked-list reversal algorithm. The post-hoc verification of this algorithm in Isabelle has been considered before [25, 40]. However, we follow Reynolds [35], who gave an informal annotated proof, and reconstruct his proof step-by-step in refinement style. As usual for verification with interactive theorem provers, functional specifications are related to imperative data structures. The former are defined recursively in functional programming style and hence amenable to proof by induction. Such detailed functional specifications of data structures used in separation logic are usually not amenable to pure first-order reasoning and therefore beyond the scope of first-order tools such as SMT solvers.

First, we define two inductive predicates on the heap. The first one creates a contiguous heap from a position e using Isabelle's functional lists as its representation, i.e., by induction on the structure of the list,

$$\begin{aligned} e\ [\mapsto]\ [\,] &= \mathbf{emp}, \\ e\ [\mapsto]\ (t\#ts) &= (e \mapsto t) * (e + 1\ [\mapsto]\ ts), \end{aligned}$$

where $[\,]$ denotes the empty list, $t\#ts$ denotes concatenation of element t with list ts, $e \mapsto t$ is again a singleton heap predicate and $\mathbf{emp}$ states that the heap is empty.

The second predicate indicates whether a heap, starting from position i, contains the linked list represented as a functional list:

$$\begin{aligned} list\ i\ [\,] &= (i = 0) \wedge \mathbf{emp}, \\ list\ i\ (j\#js) &= (i \neq 0) \wedge (\exists k.\ i\ [\mapsto]\ [j, k] \ * \ list\ k\ js). \end{aligned}$$

This is Reynolds' definition; it uses separating conjunction instead of plain conjunction.

Example 1: Constructing a linked list reversal algorithm. We reconstruct Reynolds' classical proof relative to the standard recursive function rev for functional list reversal. The initial specification statement is

$$[\![list\ `i\ A_0, list\ `j\ (rev\ A_0)]\!],$$

where A_0 is the input list and $`i$ and $`j$ are pointers to the head of the list on the heap.

The main idea behind Reynolds' proof is to split the heap into two lists, initially A_0 and an empty list, and then iteratively swing the pointer of the first element of the first list to the second list. The full proof is shown in Figure 1; we now explain its details.

In (1), we strengthen the precondition, splitting the heap into two lists A and B, and inserting a variable $`j$ initially assigned to 0 (or *null*). The equation

$$(rev\ A_0) = (rev\ A)\ @\ B$$

then holds of these lists, where $@$ denotes the append operation on linked lists. Justifying this step in Isabelle requires calling the *refinement* tactic from Section 6, which applies the leading law for assignment. This obliges us to prove that the lists A and B *de facto* exist, which is discharged automatically by calling Isabelle's *force* tactic. In fact, 8 out of the 10 proof steps in our construction are essentially automatic: they only require calling *refinement* followed by Isabelle's *force* or *auto* provers.

The new precondition generated then becomes the loop invariant of the algorithm. It allows us to refine our specification statement to a while loop in step (2), where we iterate $`i$ until it becomes 0. Calling the *refinement* tactic applies the while law for refinement. From step (3) to (10), we refine the body of the while loop and do not display the outer part of the program.

Because now $`i \neq 0$, the list $A$ has at least an element $a$. We can thus expand the definition of $list$ in step $(3)$. Next, we assign the value pointed to by $`i + 1$ to $`k$—our first list now starts at $`k$ and $`i$ points to $[a, `k]$. Isabelle then struggles to discharge the generated proof goal automatically. In this predicate, the heap is divided in three parts. One needs to prove first that $`i+1$ really points to the same value when considering just the first part of the heap or the entire heap. After that, the proof is automatic.

Step (5) performs a mutation on the heap, changing the cell $`i+1$ to $`j$, consequently $`i$ now points to $[a, `j]$. Because $*$ is commutative, we can strengthen the precondition accordingly in step (6). We now work backwards, folding the definition of $list$ in step (7) and removing the existential of a in step (8). This step again requires interaction: we need to indicate to Isabelle how to properly split the heap. This amounts to finding the right summand in the convolution expressing separating conjunction, or alternatively finding the existential split in the corresponding logical formulation. Last, to establish the invariant, we only need to swap the pointers $`j$ to $`i$ and $`i$ to $`k$ in steps (9) and (10). The resulting algorithm is highlighted in Figure 1 and shown in Figure 2. □

Example 2: Verifying a list deallocation algorithm. We verify an algorithm for list deallocation in a post-hoc fashion. It has been annotated in the standard way by a precondition, a postcondition and a loop invariant. The latter simply states that there exists

$[\![\ list\ `i\ A_0,\ list\ `j\ (rev\ A_0)\ ]\!]$
$\sqsubseteq$ (1)
$`j := 0;$
$[\![\ \exists A\ B.\ (list\ `i\ A * list\ `j\ B\ ) \wedge (rev\ A_0) = (rev\ A)\ @\ B,\ list\ `j\ (rev\ A_0)\ ]\!]$
$\sqsubseteq$ (2)
$`j := 0;$
while $`i \neq 0$ **do**
$[\![(\exists A\ B.\ (list\ `i\ A * list\ `j\ B) \wedge (rev\ A_0) = (rev\ A)\ @\ B) \wedge `i \neq 0,$
$\exists A\ B.\ (list\ `i\ A * list\ `j\ B) \wedge (rev\ A_0) = (rev\ A)\ @\ B\ ]\!]$
od

$[\![(\exists A\ B.\ (list\ `i\ A * list\ `j\ B) \wedge (rev\ A_0) = (rev\ A)\ @\ B) \wedge `i \neq 0,$
$\exists A\ B.\ (list\ `i\ A * list\ `j\ B) \wedge (rev\ A_0) = (rev\ A)\ @\ B\ ]\!]$
$\sqsubseteq$ (3)
$[\![(\exists a\ A\ B\ k.\ (`i\ [\mapsto]\ [a,k] * list\ k\ A * list\ `j\ B) \wedge (rev\ A_0) = (rev\ (a\#A))\ @\ B) \wedge `i \neq 0,$
$\exists A\ B.\ (list\ `i\ A * list\ `j\ B) \wedge (rev\ A_0) = (rev\ A)\ @\ B\ ]\!]$
$\sqsubseteq$ (4)
$`k := @(`i + 1);$
$[\![(\exists a\ A\ B.\ (`i\ [\mapsto]\ [a,`k] * list\ `k\ A * list\ `j\ B) \wedge (rev\ A_0) = (rev\ (a\#A))\ @\ B) \wedge `i \neq 0,$
$\exists A\ B.\ (list\ `i\ A * list\ `j\ B) \wedge (rev\ A_0) = (rev\ A)\ @\ B\ ]\!]$
$\sqsubseteq$ (5)
$`k := @(`i + 1);$
$@(`i + 1) := `j;$
$[\![(\exists a\ A\ B.\ (`i\ [\mapsto]\ [a,`j] * list\ `k\ A * list\ `j\ B) \wedge (rev\ A_0) = (rev\ (a\#A))\ @\ B) \wedge `i \neq 0,$
$\exists A\ B.\ (list\ `i\ A * list\ `j\ B) \wedge (rev\ A_0) = (rev\ A)\ @\ B\ ]\!]$
$\sqsubseteq$ (6)
$`k := @(`i + 1);$
$@(`i + 1) := `j;$
$[\![(\exists a\ A\ B.\ (list\ `k\ A * `i\ [\mapsto]\ [a,`j] * list\ `j\ B) \wedge (rev\ A_0) = (rev\ (a\#A))\ @\ B) \wedge `i \neq 0,$
$\exists A\ B.\ (list\ `i\ A * list\ `j\ B) \wedge (rev\ A_0) = (rev\ A)\ @\ B\ ]\!]$
$\sqsubseteq$ (7)
$`k := @(`i + 1);$
$@(`i + 1) := `j;$
$[\![(\exists a\ A\ B.\ (list\ `k\ A * list\ `i\ (a\#B)\ ) \wedge (rev\ A_0) = (rev\ A)\ @\ (a\#B)\ ) \wedge `i \neq 0,$
$\exists A\ B.\ (list\ `i\ A * list\ `j\ B) \wedge (rev\ A_0) = (rev\ A)\ @\ B\ ]\!]$
$\sqsubseteq$ (8)
$`k := @(`i + 1);$
$@(`i + 1) := `j;$
$[\![(\exists A\ B.\ (list\ `k\ A * list\ `i\ B) \wedge (rev\ A_0) = (rev\ A)\ @\ B) \wedge `i \neq 0,$
$\exists A\ B.\ (list\ `i\ A * list\ `j\ B) \wedge (rev\ A_0) = (rev\ A)\ @\ B\ ]\!]$
$\sqsubseteq$ (9)
$`k := @(`i + 1);$
$@(`i + 1) := `j;$
$`j := `i;$
$[\![(\exists A\ B.\ (list\ `k\ A * list\ `j\ B) \wedge (rev\ A_0) = (rev\ A)\ @\ B) \wedge `i \neq 0,$
$\exists A\ B.\ (list\ `i\ A * list\ `j\ B) \wedge (rev\ A_0) = (rev\ A)\ @\ B\ ]\!]$
$\sqsubseteq$ (10)
$`k := @(`i + 1);$
$@(`i + 1) := `j;$
$`j := `i;$
$`i := `k$

Fig. 1. *In situ* list reversal by refinement. The first block shows the refinement up to the introduction of the while-loop. The second block shows the refinement of the body of that loop.

```
⟦ list ‘i A0, list ‘j (rev A0) ⟧
  ⊑
‘j := 0;
while ‘i ≠ 0 do
  ‘k := @(‘i + 1);
  @(‘i + 1) := ‘j;
  ‘j := ‘i;
  ‘i := ‘k
od
```

Fig. 2. *In situ* reversal list algorithm

a list xs starting from the pointer ‘x while ‘x is not equal to 0. The Isabelle code is shown in Figure 3. Calling the *hoare* tactic generates several verification conditions for the data domain, which are easily discharged by *auto* and other Isabelle proof tools. □

```
lemma list-dealloc: ⊢ ⦃ list ‘x xs ⦄
  while ‘x ≠ 0
  inv ⦃ ∃xs. list ‘x xs ⦄
  do
    ‘y := ‘x;
    ‘x := @(‘x + 1);
    dispose ‘y;
    dispose (‘y + 1)
  od
  ⦃ emp ⦄
apply hoare
apply (auto simp: mono-def dispose-comm-def list-i-null, frule list-i-not-null-var)
by (auto simp: is-singleton-def, auto intro!: list-cong-ex heap-div-the heap-ortho-div2)
```

Fig. 3. List deallocation algorithm

Example 3: Explicit application of the frame rule. The frame rule is hidden as part of the global mutation rule in Example 1 and is not required in Example 2. Therefore we present a third (somewhat artificial) example designed to demonstrate its use. The following Isabelle code fragment shows the Hoare triple used for verification.

$$\{\ x\ [\mapsto]\ [\text{-}, j]\ *\ \mathit{list}\ j\ \mathit{as}\ \}\ @x := a\ \{\ x\ [\mapsto]\ [a, j]\ *\ \mathit{list}\ j\ \mathit{as}\ \}$$

Calling the *hoare* tactic for verification condition generation was sufficient for proving the correctness of this simple example automatically. Internally, the frame and the global mutation rule have been applied. □

In addition, we have performed a post-hoc verification of the list reversal algorithm (Figure 1) using two different approaches. The first, previously taken by Weber [40], uses Reynolds' list predicate, as we have used it in the above refinement proof. The

second follows Nipkow in using separating conjunction in the pre- and postcondition, but not in the definition of the list predicate. Since our approach is modular with respect to the underlying data model, it was straightforward to replay Nipkow's proof in our setting. The degree of proof automation with our tool is comparable to Nipkow's proof.

In summary, our approach supports the program construction and verification of pointer-based programs with separation logic, but larger case studies need to be performed to assess the performance of our tool. The algebraic approach has been used, apart from soundness proofs and the certification of the tool itself, predominantly in the derivation and implementation of tactics for program construction and verification. In the future, a Sledgehammer-style integration of optimised provers and solvers for the data level seems desirable for increasing the general degree of automation. This can be obtained, first of all, by integrating decision procedures for data types and data structures in Isabelle. Alternatively, it is sometimes possible to represent such data structures algebraically and use automated reasoning for their analysis. Dang and Möller [10], for instance, have shown how pointer structures with a generalised notion of separating conjunction can be modelled within modal Kleene algebras. Implementing this approach in Isabelle seems promising.

9 Related Work

This section discusses two main lines of related work: tool support for separation logic and similar algebraic approaches.

Numerous tools supporting separation logic have been created for various purposes. Some tools (e.g., *Predator* [17], *JStar* [14] and *VeriFast* [36]) are able to reason automatically about shape properties of real-word programming languages like C and Java. These are often highly optimised by using decision procedures and SMT solver at the data domain level. However, soundness of these tools is not guaranteed as they have not been verified relative to a small core, as provided by an LCF-style proof assistant such as Isabelle/HOL. Additionally, these tools have different degrees of interactivity and generally do not allow the user to prove any remaining verification conditions by interactively; they also cannot deal with higher-order aspects of data types and the store.

Other verification tools have been built on top of proof assistants such as Coq or Isabelle/HOL. These are generally less automatic, but allow more precise properties of the store and heap to be proved. *Smallfoot* [37], for example, has been implemented within HOL4; it supports concurrent separation logic in an approach based on [7]. Several formalisations of separation logic have been obtained in Coq, of which *YNot* [8] seems to provide the highest level of proof automation. Formalisations in Isabelle/HOL include that of Kolanski and Klein [23], which is targeted towards a subset of C, and Weber [40], which uses a shallow embedding of a simple imperative language similar to ours, but without allocation and deallocation laws. None of these tools has a lightweight middle layer formed by an algebraic semantics, which provides more modularity and flexibility when changing the programming language, logic or even the semantics. In particular, none of these tools supports program construction and refinement.

An early algebraic approach to separation logic is O'Hearn and Pym's logic of bunched implication, which describes the assertion quantale of separation logic in a

category-theoretic setting [31]. Another source of inspiration is the abstract separation logic of Calgagno, O'Hearn and Yang [7], where the role of the resource monoid, the power set lifting to an assertion algebra, and the role of locality for deriving the frame rule have been elaborated within a state transformer approach. Our predicate transformer approach based on convolution seems conceptually simpler; it is certainly more suitable for implementation. Aspects of predicate transformers and the role of locality have also been investigated in [20], but a coherent approach has not been developed.

The assertion quantale structure of separation logic has also been observed by Dang, Höfner and Möller [9], and several subclasses of assertions are studied by these authors. In addition, a relational characterisation of separating conjunction is given, and a so-called frame property, which seems similar to locality, is used for deriving the frame rule in an approach based on Kleene algebra with tests [24] and a relational semantics. The precise relationship to the approach of [7] remains to be explored. It seems to be conceptually more involved and less straightforward to implement in Isabelle than ours. Using ideas from concurrent Kleene algebra [21, 20], Dang and Möller have extended their approach to concurrent separation logic [11] with a relational semantics that seems compatible with our predicate transformer approach. This might support an extension of our formalisation to separation-based concurrency verification.

Another approach to concurrency verification is the Views framework [13, 39], which aims to provide a metatheory consisting of several parameters. Specific instantiations of these parameters gives rise to formalisms such as Owicki-Gries [33] and rely/guarantee [22]. One of the parameters of the Views framework is a resource semigroup or monoid, which supports the parallel or concurrent application of predicates to a resource, as in [21]. However, there is no coherent algebraic approach and more interesting algebraic structures, such as quantales, are never explored. Instead, reasoning proceeds by using the operational semantics of their simple language, and hence, the approach, and even the aims, of the Views framework differ significantly from ours.

Finally, we could have adapted modal Kleene algebra [26], which has already been formalised in Isabelle [19], to program our predicate transformer approach. We could have defined this algebra over the assertion quantale instead of the usual boolean algebra, with the definition of locality (Section 4) providing the interaction of separating conjunction with the modal box operator. In practice, however, this leads to more complex and less automatic Isabelle proofs due to the increased distance between the abstract algebraic level and the concrete store-heap model. An optimisation is left for future work.

10 Conclusion

A principled approach to the design of program verification and construction tools for separation logic with the Isabelle theorem proving environment has been presented. The general approach has been used previously for implementing tools for the construction and verification of simple while programs [2] and rely-guarantee based concurrent programs [1]. It aims at a clean separation of concerns between the control flow and the data domain of programs and focusses on developing a lightweight algebraic layer from which verification conditions or transformation and refinement laws can be developed

by simple equational reasoning. Previously, in the case of while programs, this layer has been provided by Kleene algebras with tests; in the rely-guarantee case, new algebraic foundations based on concurrent Kleene algebras were required.

Our approach to separation logic is a conceptual reconstruction of separation logic beyond a mere implementation as well, which forms a contribution in its own right. To make an Isabelle implementation as small, automatic and modular as possible, we have once more aimed at finding a conceptually minimalist setting from which powerful verification conditions as well as transformation and refinement laws can be derived. Though strongly inspired by abstract separation logic [7] and the logic of bunched implications [31], we use a different combination of simplicity and mathematical abstraction. In contrast to the logic of bunched implication, we use power series instead of higher categories, and in contrast to abstract separation logic we follow [41] in using predicate transformers in the style of boolean algebras with operators instead of state transformers. These design choices allow us to use power series, quantales and generic lifting constructions throughout the approach, which leads indeed to a very small and highly automated Isabelle implementation. A particular feature of this approach is the view on separating conjunction as a notion of convolution over resources.

Our tool prototype has so far allowed us to verify some simple pointer-based programs with a relatively high degree of automation. So far it is certainly useful for educational and research purposes, but extensions and optimisations beyond the mere proof of concept are desirable. This includes the consideration of recursive procedures [20], for variable framing laws in our refinement calculus or of error states [7], the development of more sophisticated proof tactics, and the integration of tools and techniques for automatic data-level reasoning in Sledgehammer style.

Other opportunities for future work lie in the integration of categorial approaches to data type constructions [18, 28], the consolidation with Preoteasa's approach to predicate transformers in Isabelle [34], in a further abstraction of the control-flow layer by defining modal Kleene algebras over assertion quantales [26] for which some Isabelle infrastructure already exists [19], in a combination with our rely-guarantee tool into RGSep-style tools for concurrency verification [38], and in the exploration of the language connection of separating conjunction in terms of representability and decidability results.

Acknowledgements. We are grateful for support by EPSRC grant EP/J003727/1 and the CNPq. The third author would like to thank Tony Hoare, Peter O'Hearn and Matthew Parkinson for discussions on concurrent Kleene algebra and separation logic.

References

[1] A. Armstrong, V. B. F. Gomes, and G. Struth. Algebraic principles for rely-guarantee style concurrency verification tools. In C. B. Jones, P. Pihlajasaari, and J. Sun, editors, *FM 2014*, volume 8442 of *LNCS*, pages 78–93. Springer, 2014.

[2] A. Armstrong, V. B. F. Gomes, and G. Struth. Lightweight program construction and verification tools in Isabelle/HOL. In D. Giannakopoulou and G. Salaün, editors, *SEFM 2014*, volume 8702 of *LNCS*, pages 5–19. Springer, 2014.

[3] R.-J. Back and J. von Wright. *Refinement calculus*. Springer, 1999.

[4] V. Bergelson, A. Blass, and N. Hindman. Partition theorems for spaces of variable words. *Proc. London Mathematical Society*, 68(3):449–476, 1994.
[5] J. Berstel and C. Reutenauer. *Les séries rationnelles et leurs langagues*. Masson, 1984.
[6] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In J. Palsberg and M. Abadi, editors, *POPL*, pages 259–270. ACM, 2005.
[7] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS 2007*, pages 366–378. IEEE Computer Society, 2007.
[8] A. Chlipala, J. G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. In G. Hutton and A. P. Tolmach, editors, *ICFP 2009*, pages 79–90. ACM, 2009.
[9] H.-H. Dang, P. Höfner, and B. Möller. Algebraic separation logic. *J. Log. Algebr. Program.*, 80(6):221–247, 2011.
[10] H.-H. Dang and B. Möller. Transitive separation logic. In W. Kahl and T. G. Griffin, editors, *RAMiCS 2012*, volume 7560 of *LNCS*, pages 1–16. Springer, 2012.
[11] H.-H. Dang and B. Möller. Concurrency and local reasoning under reverse interchange. *Science of Computer Programming*, 85:204–223, 2014.
[12] B. Day. On closed categories of functors. In *Reports of the Midwest Category Seminar IV*, volume 137 of *Lecture Notes in Mathematics*, pages 1–38. Springer, 1970.
[13] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. J. Parkinson, and H. Yang. Views: compositional reasoning for concurrent programs. In R. Giacobazzi and R. Cousot, editors, *POPL*, pages 287–300. ACM, 2013.
[14] D. Distefano and M. J. Parkinson. jstar: towards practical verification for java. In G. E. Harris, editor, *OOPSLA 2008*, pages 213–226. ACM, 2008.
[15] B. Dongol, I. J. Hayes, and G. Struth. Convolution, separation and concurrency. *CoRR*, abs/1312.1225, 2014.
[16] M. Droste, W. Kuich, and H. Vogler. *Handbook of Weighted Automata*. Springer, 1st edition, 2009.
[17] K. Dudka, P. Peringer, and T. Vojnar. Byte-precise verification of low-level list manipulation. In F. Logozzo and M. Fähndrich, editors, *SAS 2013*, volume 7935 of *LNCS*, pages 215–237. Springer, 2013.
[18] P. H. B. Gardiner, C. E. Martin, and O. de Moor. An algebraic construction of predicate transformers. *Science of Computer Programming*, 22(1-2):21–44, 1994.
[19] W. Guttmann, G. Struth, and T. Weber. Automating algebraic methods in Isabelle. In S. Qin and Z. Qiu, editors, *ICFEM 2011*, volume 6991 of *LNCS*, pages 617–632. Springer, 2011.
[20] C. A. R. Hoare, A. Hussain, B. Möller, P. W. O'Hearn, R. Lerchedahl Petersen, and G. Struth. On locality and the exchange law for concurrent processes. In J.-P. Katoen and B. König, editors, *CONCUR 2011*, volume 6901 of *LNCS*, pages 250–264. Springer, 2011.
[21] T. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent kleene algebra and its foundations. *J. Log. Algebr. Program.*, 80(6):266–296, 2011.
[22] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.
[23] G. Klein, R. Kolanski, and A. Boyton. Mechanised separation algebra. In L. Beringer and A. P. Felty, editors, *ITP 2012*, volume 7406 of *LNCS*, pages 332–337. Springer, 2012.
[24] D. Kozen. On Hoare logic and Kleene algebra with tests. *ACM TOCL*, 1(1):60–76, 2000.
[25] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199(1-2):200–227, 2005.
[26] B. Möller and G. Struth. Algebras of modal operators and partial correctness. *Theoretical Computer Science*, 351(2):221–239, 2006.
[27] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1998.

[28] D. A. Naumann. Beyond fun: Order and membership in polytypic imperative programming. In J. Jeuring, editor, *MPC 1998*, volume 1422 of *LNCS*, pages 286–314. Springer, 1998.

[29] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[30] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007.

[31] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[32] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *CSL 2001*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.

[33] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6:319–340, 1976.

[34] V. Preoteasa. Algebra of monotonic boolean transformers. In A. da S. Simão and C. Morgan, editors, *SBMF 2011*, volume 7021 of *LNCS*, pages 140–155. Springer, 2011.

[35] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[36] J. Smans, B. Jacobs, and F. Piessens. Verifast for java: A tutorial. In D. Clarke, J. Noble, and T. Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *LNCS*, pages 407–442. Springer, 2013.

[37] T. Tuerk. *A Separation Logic Framework for HOL*. PhD thesis, Computer Laboratory, University of Cambridge, 2011.

[38] V. Vafeiadis. *Modular Fine-Grained Concurrency Verificaiton*. PhD thesis, Computer Laboratory, University of Cambridge, 2007.

[39] S. van Staden. Constructing the views framework. In D. Naumann, editor, *UTP 2014*, volume 8963 of *LNCS*, pages 62–83. Springer, 2014.

[40] T. Weber. Towards mechanized program verification with separation logic. In J. Marcinkowski and A. Tarlecki, editors, *CSL 2004*, volume 3210 of *LNCS*, pages 250–264. Springer, 2004.

[41] H. Yang and P. W. O'Hearn. A semantic basis for local reasoning. In M. Nielsen and U. Engberg, editors, *FOSSACS 2002*, volume 2303 of *LNCS*, pages 402–416. Springer, 2002.