Mechanised Separation Algebra

Gerwin Klein, Rafal Kolanski, and Andrew Boyton

[1] NICTA, Sydney, Australia[⋆]
[2] School of Computer Science and Engineering, UNSW, Sydney, Australia
{first-name.last-name}@nicta.com.au

Abstract. We present an Isabelle/HOL library with a generic type class implementation of separation algebra, develop basic separation logic concepts on top of it, and implement generic automated tactic support that can be used directly for any instantiation of the library. We show that the library is usable by multiple example instantiations that include common as well as more exotic base structures such as heap and virtual memory, and report on our experience using it in operating systems kernel verification.

Keywords: Isabelle, Separation Logic

1 Introduction

The aim of this work is to support and significantly reduce the effort for future separation logic developments in Isabelle/HOL by factoring out the part of separation logic that can be treated abstractly once and for all. This includes developing typical default rule sets for reasoning as well as automated tactic support for separation logic. We show that both of these can be developed in the abstract and can be used directly for instantiations.

The library supports users by enforcing a clear axiomatic interface that defines the basic properties a separation algebra provides as the underlying structure for separation logic. While these properties may seem obvious for simple underlying structures like a classical heap, more exotic structures such as virtual memory or permissions are less straight-forward to establish. The library provides an incentive to formalise towards this interface, on the one hand forcing the user to develop an actual separation algebra with actual separation logic behaviour, and on the other hand rewarding the user with supplying a significant amount of free infrastructure and reasoning support.

Neither the idea of separation algebra nor its mechanisation is new. Separation algebra was introduced by Calcagno et al [2] whose development we follow, transforming it only slightly to make it more convenient for mechanised instantiation. Mechanisations of separation logic in various theorem provers are plentiful, we have ourselves developed multiple versions [4,5] as have many others. Similarly a

[⋆] NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program number of mechanisations of abstract separation algebra exist, e.g. by Tuerk [6] in HOL4, by Bengtson et al [1] in Coq, or by ourselves in Isabelle/HOL [4].

The existence of so many mechanisations of separation logic is the main motivation for this work. A large portion of the effort for developing a new instance of separation logic consists of boilerplate definitions, deriving standard properties, and often re-invented automated tactic support. While separation algebra is used to justify the separation logic properties of specific developments [4], or to conduct a part of the development in the abstract before proceeding to the concrete [1, 6], the number of instantiations of these abstract frameworks so far tends to be one. In short, the library potential of separation algebra has not been exploited yet in a practically re-usable way. Such light-weight library support with generic interactive separation logic proof tactics is the contribution of this paper.

A particular feature of the library presented here is that it does not come with a programming language, state space, or a definition of hoare triples. Instead it provides support for instantiating your own language to separation logic. This is important, because fixing the language, even if it is an abstract generic language, destroys most of the genericity that separation algebra can achieve. We have instantiated our framework with multiple different language formalisations, including both deep and shallow embeddings.

In Sec 2 we show the main interface of the separation algebra class in Isabelle/HOL, describe how it differs from Calcagno et al, and which concepts and properties we developed on top. Sec 3 describes the generic tactic support, and Sec 4 describes our experience with four example instantiations.

2 Separation Algebra

This section gives a brief overview of our formulation of abstract separation algebra. The basic idea is simple: capture separation algebra as defined by Calcagno et al [2] with Isabelle/HOL type class axioms, develop separation logic concepts in the abstract as far as can be done without defining a programming language, and instantiate simply using Isabelle's type class instantiations. This leads to a lightweight formalisation that carries surprisingly far.

Calcagno et al define separation algebra as *a cancellative, partial commutative monoid* $(\Sigma, \cdot, u)$. *A partial commutative monoid is given by a partial binary operation where the unity, commutativity and associativity laws hold for the equality that means both sides are defined and equal, or both are undefined.*

For a concrete instance, think of the carrier set as a heap and of the binary operation as map addition. The definition induces separateness and substate relations, and is then used to define separating conjunction, implication, etc. Since the cancellative property is needed primarily for completeness and concurrency, we leave it out at the type class level. It can either be introduced in a second type class on top or provided as an explicit assumption where needed. The definition above directly translates to the following type class axioms.

$$x \oplus 0 = \lfloor x \rfloor \qquad x \oplus y = y \oplus x \qquad a ++ b ++ c = (a ++ b) ++ c$$

where *op* $\oplus$::*'a* $\Rightarrow$ *'a* $\Rightarrow$ *'a option* is the partial binary operator and *op ++*::*'a option* $\Rightarrow$ *'a option* $\Rightarrow$ *'a option* is the $\oplus$ operator lifted to strict partiality. From this the usual definitions of separation logic can be developed. However, as to be expected in HOL, partiality makes the $\oplus$ operator cumbersome to instantiate; especially the third axiom often leads to numerous case distinctions. Hence, we make the binary operator total, re-using Isabelle's standard *+* as syntax. Totality requires us to put explicit side conditions on the laws above and to make disjointness a parameter of the type class leading to further axioms. The full definition of separation algebra with a total binary operator is

class *sep_algebra = zero + plus +*
  fixes *op ##*::*'a* $\Rightarrow$ *'a* $\Rightarrow$ *bool*
  assumes *x ## 0* **and** *x ## y* $\Longrightarrow$ *y ## x* **and** *x + 0 = x*
  assumes *x ## y* $\Longrightarrow$ *x + y = y + x*
  assumes $[\![$*x ## y; y ## z; x ## z*$]\!]$ $\Longrightarrow$ *x + y + z = x + (y + z)*
  assumes $[\![$*x ## y + z; y ## z*$]\!]$ $\Longrightarrow$ *x ## y*
  assumes $[\![$*x ## y + z; y ## z*$]\!]$ $\Longrightarrow$ *x + y ## z*

This form is precisely as strong as Calcagno et al's formulation above in the sense that either axiom set can be derived from the other. While 7 axioms might seem to induce a higher burden than the 3 axioms above, the absence of lifting and type partiality made them clearly smoother to instantiate in our experience, in essence guiding the structure of the case distinctions needed in the first formulation.

Based on this type class, the definitions of basic separation logic concepts are completely standard, as are the properties we can derive for them. Some definitions are summarised below.

$$\begin{array}{ll}
P \wedge\!* Q & \equiv \lambda h.\ \exists x\ y.\ x \mathbin{\#\#} y \wedge h = x + y \wedge P\ x \wedge Q\ y \\
P \longrightarrow\!* Q & \equiv \lambda h.\ \forall h'.\ h \mathbin{\#\#} h' \wedge P\ h' \longrightarrow Q\ (h + h') \\
x \preceq y & \equiv \exists z.\ x \mathbin{\#\#} z \wedge x + z = y \\
\square & \equiv \lambda h.\ h = (0::'a) \\
\bigwedge\!* Ps & \equiv \mathsf{foldl}\ op \wedge\!* \square\ Ps
\end{array}$$

On top of these, we have formalised the standard concepts of pure, intuitionistic, and precise formulas together with their main properties. We note to Isabelle that separating conjunction forms a commutative, additive monoid with the empty heap assertion. This means all library properties proved about this structure become automatically available, including laws about fold over lists of assertions.

From this development, we can set up standard simplification rule sets, such as maximising quantifier scopes (which is the more useful direction in separation logic), that are directly applicable in instances.

The assertions we cannot formalise on this abstract level are maps-to predicates such as the classical $p \mapsto v$. These depend on the underlying structure and can only be done on at least a partial instantiation.

Future work for the abstract development could include a second layer introducing assumptions on the semantics of the programming language instance. It then becomes possible to define locality, the frame rule, and (best) local actions generically for those languages where they make sense, e.g. for deep embeddings.

3 Automation

This section gives a brief overview of the automated tactics we have introduced on top of the abstract separation algebra formalisation.

There are three main situations that make interactive mechanical reasoning about separation logic in HOL frameworks cumbersome. Their root cause is that the built-in mechanism for managing assumption contexts does not work for the substructural separation logic and therefore needs to be done manually.

The first situation is the application of simple implications and the removal of unnecessary context. Consider the goal $(P \wedge* \ p \mapsto v \wedge* \ Q)\ h \Longrightarrow (Q \wedge* \ P \wedge* \ p \mapsto -)\ h$. This should be trivial and automatic, but without further support requires manual rule applications for commutativity and associativity of $\wedge*$ before the basic implication between $p \mapsto v$ and $p \mapsto -$ can be applied. Rewriting with AC rules alleviates the problem somewhat, but leads to unpleasant side effects when there are uninstantiated schematic variables in the goal. In a normal, boolean context, we would merely have applied the implication as a forward rule and solved the rest by assumption, having the theorem prover take care of reordering, unification, and assumption matching.

In a substructural logic, we cannot expect to always be able to remove context, but at least the order of conjuncts should be irrelevant. In the usual case, we expect to apply a rule of the form $(P \wedge* \ Q)\ h \Longrightarrow (P' \wedge* \ Q)\ h$ either as a forward, destruction, or introduction rule where real implication is between P and P' and Q tells us it can be applied in any context. Our tactics `sep_frule`, `sep_drule`, and `sep_rule` try rotating assumptions and conclusion of the goal respectively until the rule matches. If P occurs as a top-level separation conjunct in the assumptions, this will be successful, and the rule is applied.

This takes away the tedium of positional adjustments and gives us basic rule application similar to the boolean case. The common case of reasoning about heap updates falls into this category. Heap update can usually be characterised by a rule such as $(p \mapsto - \wedge* \ Q)\ h \Longrightarrow (p \mapsto v \wedge* \ Q)\ (h(p \mapsto v))$ where h is a simple heap map. Given a conclusion that mentions an updated heap $h(p \mapsto v)$ over a potentially large separating conjunction that somewhere mentions a term of the form $p \mapsto v$, often with a schematic v to be instantiated by the heap update, then we can now make progress with a simple `sep_rule` application.

Note that while the application scenario is instance dependent, the tactic is not. It simply takes a rule as parameter.

The second situation is reasoning about heap values. Again, consider a simple heap instantiation of the framework. The rule to apply would be $(p \mapsto v \wedge* \ Q)\ h \Longrightarrow \mathsf{the}\ (h\ p) = v$. The idea is similar to above, but this time we extend Isabelle's substitution tactic to automatically solve the side condition of the substitution by rotating conjuncts appropriately after applying the equality. It is important for this to happen atomically to the user, because the equality will instantiate the rule only partially in h and p, while the side condition determines the value v. Again, the tactic is generic, the rule comes from the instantiation.

The third situation is clearing context to focus on the interesting implication parts of a separation goal after heap update and value reasoning are done. The idea is to automatically remove all conjuncts that are equal in assumption and conclusion as well as solve any trivial implications. The tactic *sep_cancel* that achieves this is higher-level than the tactics above, building on the same principles.

Finally, we supply a low-level tactic *sep_select n* that rotates conjunct *n* to the front while leaving the rest, including schematics, untouched.

With these basic tactics in place, higher-level special-purpose tactics can be developed much more easily in the future. The rule application and substitution tactics fully support backtracking and chaining with other Isabelle tactics.

One concrete area of future in automation is porting Kolanski's machinery for automatically generating mapsto-arrow variants [4], e.g. automatically lifting an arrow to its weak variant, existence variant, list variant, etc, including generating standard syntax and proof rules between them which could then automatically feed into tools like *sep_cancel*. Again, the setup would be generic, but used to generate rules for instances.

4 Instantiations

We have instantiated the library so far to four different languages and variants of separation logic.

For the first instance, we took an existing example for separation logic that is part of the Isabelle distribution and ported it to sit on top of the library. The effort for this was minimal, less than an afternoon for one person, and unsurprisingly the result was drastically smaller than the original, because all boilerplate separation logic definitions and syntax declarations could be removed. The example itself is the classic separation logic benchmark of list reversal in a simple heap of type *nat* ⇒ *nat option* on a language with a VCG, deeply embedded statements and shallowly embedded expressions.

The original proof went through almost completely unchanged after replacing names of definitions. We then additionally shrunk the original proof from 24 to 14 lines, applying the tactics described above and transforming the script from technical details to reasoning about semantic content.

While a nice first indication, this example was clearly a toy problem. A more serious and complex instance of separation logic is a variant for reasoning about virtual memory by Kolanski [4]. We have instantiated the library to this variant as a test case, and generalised the virtual memory logic in the process.

The final two instantiations are taken from the ongoing verification of user- and kernel-level system initialisation in the seL4 microkernel [3]. Both instantiations are for a shallow embedding using the nondeterministic state monad, but for two different abstraction levels and state spaces. In the more abstract, user-level setting, the overall program state contains a heap of type *obj_id* ⇒ *obj option*, where *obj* is a datatype with objects that may themselves contain a map *slot* ⇒ *cap option* as well as further atomic data fields. In this setting we would like to not just separate parts of the heap, but also separate parts of the maps within objects, and potentially their fields. The theoretical background of this is not new, but the technical implementation in the theorem prover is nontrivial.

The clear interface of separation algebra helped one of the authors with no prior experience in separation logic to instantiate the framework within a week, and helped an undergraduate student to prove separation logic specifications of seL4 kernel functions within the space of two weeks. This is a significant improvement over previous training times in seL4 proofs.

The second monadic instance is similar in idea to the one above, but technically more involved, because it works on a lower abstraction level of the kernel, where in-object maps are not partial, some of the maps are encoded as atomic fields, and the data types are more involved. To use these with separation logic, we had to extend the state space by ghost state that encodes the partiality demanded by the logic. The instantiation to the framework is complete, and we can now proceed with the same level of abstraction in assertions as above.

Although shallow embeddings do not directly support the frame rule, we have found the approach of baking the frame rule into assertions by appending `∧* P` to pre- and post-conditions productive. This decision is independent of the library.

5 Conclusion

We have presented early work on a lightweight Isabelle/HOL library with an abstract type class for separation algebra and generic support for interactive separation logic tactics. While we have further concrete ideas for automation and for more type class layers with deeper support of additional separation logic concepts, the four nontrivial instantiations with productive proofs on top that we could produce in a short amount of time show that the concept is promising.

The idea is to provide the basis for rapid prototyping of new separation logic variants on different languages, be they deep or shallow embeddings, and of new automated interactive tactics that can be used across a number of instantiations.

Acknowledgements We thank Matthias Daum for his comments on this paper.

References

1. J. Bengtson, J. Jensen, F. Sieczkowski, and L. Birkedal. Verifying object-oriented programs with higher-order separation logic in Coq. In M. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *ITP*, volume 6898 of *LNCS*, pages 22–38. Springer, 2011.
2. C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Proc. 22nd LICS*, pages 366–378. IEEE, 2007.
3. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *22nd SOSP*, pages 207–220. ACM, Oct 2009.
4. R. Kolanski. *Verification of Programs in Virtual Memory Using Separation Logic*. PhD thesis, School Comp. Sci. & Engin., Jul 2011.
5. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *34th POPL*, pages 97–108, Nice, France, Jan 2007. ACM.
6. T. Tuerk. A formalisation of smallfoot in HOL. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, LNCS, pages 469–484. Springer, Aug 2009.