A Separation Logic Framework in HOL

Thomas Tuerk

University of Cambridge Computer Laboratory
William Gates Building, JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom
http://www.cl.cam.ac.uk

Abstract. Separation Logic has proved to be a powerful technique for program verification. There are tools based on ideas of separation logic and even some implementations in theorem provers. However, these implementations all concentrate on the verification of programs written in one specific language. In contrast, I propose to build a framework inside the HOL theorem prover that concentrates on separation logic itself. This framework should be general enough to express different flavours of separation logic and it should be easily instantiable for different programming languages.
In this paper, a formalisation of *Abstract Separation Logic* in HOL is presented as a first step towards implementing the proposed framework. This presentation contains an initial case study – an implementation of a tool similar to Smallfoot – which gives some insights into the suitability of abstract separation logic for this purpose.

1 Introduction

Separation logic is an extension of Hoare logic that allows local reasoning about mutable data structures, especially structures using pointers. It's been introduced by O'Hearn, Reynolds and Yang in 2001 [7,9] based on earlier ideas by Burstall, Reynolds, Ishtiaq and O'Hearn. The main idea of separation logic is the usage of a *spatial conjunction* $p * q$ that asserts that the formulas p and q hold on *separate* parts of the state. This notion of separate parts allows local reasoning and an elegant solution to aliasing problems. Moreover, it enables separation logic to be extended to concurrent programs in a natural way [3].

1.1 Introductory Example

Classically, separation logic uses states consisting of a stack and a heap. Consider such states and, as an example, an update of the heap at the location stored in a stack-variable x with the value 2 ($[x] := 2$). In order to reason about this assignment, separation logic needs the precondition, that x is allocated ($x \mapsto \_$). Using this precondition, one can conclude that after executing the assignment x points to the value 2 ($x \mapsto 2$). This whole reasoning is captured by the validity of the following Hoare triple $\{x \mapsto \_\}\ [x] := 2\ \{x \mapsto 2\}$. In contrast to classical Hoare logic, the pre- and postcondition have to mention at least, that x is allocated. $\{x \mapsto \_\}\ [x] := 2\ \{\text{emp}\}$ for example does not hold. So, one cannot easily remove parts of the postcondition. On the other hand, the conditions can be safely extended by an arbitrary context using the spatial conjunction operator. This is due to the separation that this operator provides and locality properties of programming languages. This property gives rise to the following inference rules for extending the conditions with a frame and parallel composition, which capture the essence of local reasoning:

$$\frac{\{P\}\ prog\ \{Q\}}{\{P * R\}\ prog\ \{Q * R\}} \qquad \frac{\{P_1\}\ prog_1\{Q_1\} \qquad \{P_2\}\ prog_2\{Q_2\}}{\{P_1 * P_2\}\ prog_1||prog_2\ \{Q_1 * Q_2\}}$$

1.2 Existing Implementations of Separation Logic

Due to local reasoning, separation logic scales much better than classical Hoare logic to the verification of large programs. Moreover, even for simple, small pieces of programs separation logic proofs are often much more succinct and easier to read. Separation logic has become more and more popular during the last few

years. There are several implementations: Smallfoot [2], SLAyer and SpaceInvader are probably some of the best know examples. There are also formalisations inside theorem provers. There are several implementations of separation logic in Isabelle/HOL [10,11] and one in Coq [6], which is used and extended by the *Concurrent C-Minor Project* [1]. There is also related work by Andrew Ireland et al [5], but I learned about it too late to discuss it here.

1.3 Motivation for the Proposed Framework

The implementations mentioned above all focus on one specific programming language (mostly some C-like imperative language) and take design decisions with respect to this language. While in most cases a specific language is all you are interested in and while it allows better automation and perhaps simplified and more intuitive formalisations for this specific language, it makes reusing of these formalisations very difficult and distracts from the core features of separation logic.

The differences and problems with reuse start with simple design decisions like whether the heap is modelled as a function from integers to integers or whether it can contain arrays of integers. Is it perhaps even more appropriate to explicitly model the finiteness of memory by having 32- or 64-bit words instead of integers? Problems increase if you consider changes to the logic itself like equipping values on the stack with explicit read/write permissions [8]. Even worse, one may not use the classical stack/heap model at all and be interested in verifying assembler code that operates on a fixed set of registers and a chunk of memory instead of a heap and a stack. In short, there are a lot of different *flavours* of separation logic, which share a large common part, but may dramatically vary in detail.

Therefore, I suggest building a framework for separation logic inside the HOL theorem prover that concentrates on separation logic itself instead of a concrete programming languages.

1.4 Structure of this Paper

In the next section, I briefly state some ideas for such a separation logic framework in HOL. Then, a first step towards implementing such a framework is presented: a formalisation of *Abstract Separation Logic* in HOL. The paper will conclude with first experiences of instantiating abstract separation logic and plans for future work.

2 Ideas for a Separation Logic Framework

I suggest building a framework for separation logic inside the HOL theorem prover. The main part of this framework should be general, but it should provide support for instantiating it for different concrete languages and applications. I hope that these instantiations can be significantly different while still keeping a large common part. As case studies one could try to implement:

- a completely automated tool – similar to Smallfoot [2] – for a simple imperative language (this case study is mainly done);
- a tool to interactively verify programs written in a more complicated imperative language;
- an interactive *separation logic calculator* that may be used to formalise long, complicated hand-proofs found in separation logic papers;
- a tool to verify assembler programs.

I hope that splitting the formalisation into a general part and instantiations will add to clarity and help to keep a lot of proofs simple. For example, one should be able to neglect the complicated structure of states some separation logics use (e. g. [8]) and mainly work with abstractions. However, choosing these abstractions has a huge impact and is far from obvious. On the one hand, the abstractions should be abstract enough to model a large variety of separation logics and to keep the general part as simple as possible. On the other hand, they need to be concrete enough to be able to convince people that the formalisation of a concrete programming language in the framework is reasonable or even intuitive. I believe, that *Abstract Separation Logic* [4] is a good starting point to build such a framework.

3 Abstract Separation Logic

Abstract Separation Logic was introduced by Calcagno, O'Hearn and Yang [4]. While most separation logics operate on states consisting of a stack and a heap, abstract separation logic can use arbitrary states. A partial function $\bullet$ is used to combine these states. Two states s_1 and s_2 are *separate*, iff $s_1 \bullet s_2$ is defined. Using this notion, one can easily define the spatial conjunction operator $*$ as follows:

$$P * Q := \{s \mid \exists p, q.\ (p \bullet q = s)\ \wedge\ p \in P\ \wedge\ q \in Q\}$$

Intuitively, this means, that a state s satisfies $P * Q$ iff it can be split into two separate states p and q such that p satisfies P and q satisfies Q. Other standard separation logic constructs can be defined in a natural way as well. However, in order for these definitions to be useful, the combination function $\bullet$ has to satisfy some properties: a neutral element u has to exists, such that the set of states Σ forms a *separation algebra* with $\bullet$ and u, i. e. $(\Sigma, \bullet, u)$ is a cancellative, partial commutative monoid.

3.1 Programming Language

The programming language used by abstract separation logic is abstract as well. An *action* act is a function from a state s to a set of states S or a special failure state $\top$. If $act(s) = \top$, then an error may occur during the execution of the action starting in state s. This is used to model for example that an action might try to dereference a null-pointer or read an unallocated location on the heap. If $act(s) = S$, then no error will occur and after executing the state will be one of the states in S. If S is the empty set, the action does not terminate. Thus, $act(s) = S$ can be used to express nondeterminism and nontermination.

Based on this notion of actions, a Hoare triple $\{P\}\ act\ \{Q\}$ holds, iff for all states p that satisfy the *precondition* P the action does not fail, i. e. there is a set of states S with $act(p) = S$, and leads to a state that satisfies the *postcondition* Q, i. e. $S \subseteq Q$. Notice, that this describes partial correctness, since a Hoare triple is trivially satisfied, if act does not terminate, i. e. if S is empty.

As explained above, it is an essential feature of separation logic, that a specification can be safely extended by an arbitrary frame R. This is the essence of local reasoning. Therefore, abstract separation logic only uses actions that satisfy this property. These actions are called *local*. More explicitly, an action is called local, iff it satisfies the following inference rule for all P, Q and R:

$$\frac{\{P\}\ act\ \{Q\}}{\{P * R\}\ act\ \{Q * R\}}$$

Using only local actions is not a big restriction, since most programming languages just use local actions anyhow.

Abstract separation logic provides some operations to extend the set of local actions provided by the user to a programming language. This extension guarantees that all programs written in it are local actions themselves. There are e. g. a sequential composition operator $(;)$, a nondeterministic choice operator $(+)$ and a Kleene star operator $(^*)$. One important predefined local action is *assume* B for a predicate B. The predicate B has to be *intuitionistic* and its intuitionistic negation is denoted by $\neg_i B$. Intuitionistic predicates are not discussed here for reasons of brevity. Given a state s the action *assume* B

- skips, iff B holds in all extensions of s by a frame;
- diverges, iff B does not hold in any extension of s by a frame;
- fails otherwise.

For example, in a usual setting *assume* $x \neq 0$ skips, iff x is defined in the state s and does not contain the value 0, diverges iff x is defined and contains the value 0 and fails, iff x is not defined. Since abstract separation logic considers partial correctness, divergence has the effect of satisfying any specification. Therefore, conditional execution and loops can be defined using nondeterminism, Kleene star and assume:

$$\begin{aligned} &\text{if } B \text{ then } prog_1 \text{ else } prog_2 := (assume\ B;\ prog_1)\ +\ (assume\ \neg_i B;\ prog_2) \\ &\text{while } B \text{ do } body \qquad\quad := (assume\ B; body)^*;\ assume\ \neg_i B \end{aligned}$$

Notice, that the semantics of a while-loop can be seen as nondeterministically choosing a natural number n and unrolling the loop n times. If the wrong number for a particular input has been chosen, the execution is aborted by one of the assumptions. If the loop would not terminate, no such number exists and every choice is aborted by the assumptions. This captures the semantics of loops in a simple, abstract way, but it is not intuitive.

Abstract separation logic defines constructs for expressing parallel programs as well. There is a parallel composition operator || that executes two programs in parallel. Moreover, there are semaphore operations. However, for reasons of brevity these operations are not presented here. Neither is the exact semantics discussed. However, the part described here should give a glimpse of what constructs abstract separation logic provides and how its abstract programming language can be instantiated to a concrete one. For more details please refer to the original paper about abstract separation logic [4].

3.2 Inference Rules

Using the detailed semantics of abstract separation logic, one can prove high-level inference rules correct. The goal is to have sufficiently expressive inference rules to verify specification in this high-level view instead of using the detailed semantics all the time. Some important inference rules, that are valid in abstract separation logic are:

$$\frac{\{P\}\ p\ \{Q\}}{\{P * R\}\ p\ \{Q * R\}} \qquad \frac{\{P\}\ p_1\ \{Q\} \quad \{Q\}\ p_2\ \{R\}}{\{P\}\ p_1; p_2\ \{R\}}$$

$$\frac{\{P\}\ p\ \{P\}}{\{P\}\ p^*\ \{P\}} \qquad \frac{\{P\}\ p_1\ \{Q\} \quad \{P\}\ p_2\ \{Q\}}{\{P\}\ p_1 + p_2\ \{Q\}}$$

$$\frac{\{P_1\}\ p_1\ \{Q_1\} \quad \{P_2\}\ p_2\ \{Q_2\}}{\{P_1 * P_2\}\ p_1 || p_2\ \{Q_1 * Q_2\}} \qquad \frac{\{B \wedge P\}\ p_1\ \{Q\} \quad \{\neg_i B \wedge P\}\ p_2\ \{Q\}}{\{P\}\ \text{if } B \text{ then } p_1 \text{ else } p_2\ \{Q\}}$$

$$\frac{\{B \wedge P\}\ p\ \{P\}}{\{P\}\ \text{while}\ \ B\ \ \text{do}\ \ p\ \{\neg_i B \wedge P\}}$$

3.3 HOL implementation

I formalised abstract separation logic as described in the paper by Calcagno, O'Hearn and Yang [4]. This formalisation includes the specification logic, concurrent semantics, proofs for all inference rules and most lemmata found in this paper. There are also some extensions. The HOL sources can be found in the HOL repository[1].

The formalisation consists of a mixture of deep and shallow embeddings. In general, I tried to keep it as flexible as possible for instantiations and used shallow embeddings. However, the programs are defined by a deep-embedding. This embedding depends on several free type variables used to instantiate elementary actions, predicates etc. To instantiate the formalisation of abstract separation logic, one has to provide among other things a concrete type for states, a partial function $\bullet$ and a neutral element u for combining states, concrete types for elementary actions and predicates and functions assigning a semantic to these actions and predicates. All these things needed for an instantiation are collected in an environment term. One has to prove that this environment is valid, which includes for example that $\bullet$ forms a separation algebra or that all used elementary actions are local ones. A lot of functions defined in the formalisation take this environment as an argument and theorems use the validity of the environment as a precondition.

I also formalised some extensions. The largest extension are procedures. Their semantics are defined by replacing a function call with the body of the function. The formalisation can handle mutually recursive

[1] The HOL repository can be found at `http://hol.sourceforge.net`. The formalisation of abstract separation logic and case studies are located in `examples/separationLogic`.

definitions and there are some inference rules to eliminate recursion during the verification. The usual separation logic predicates like magic wand or septraction are predefined. Additionally, there are predicates like separation logic quantifiers or a definition scheme for recursive predicates. Moreover, there are also definitions and inference rules for commonly used imperative constructs like loops or conditional execution. However, there is still a lot to add and I expect the formalisation to grow with future instantiations.

4 First Instantiations and Resulting Experiences

After formalising abstract separation logic in HOL, I tried to instantiate it to the flavour of separation logic presented in *Variables as Resource in Hoare Logic* by Parkinson, Bornat and Calcagno [8]. This separation logic treats stack-variables as resources, i. e. it extends the separation idea from the heap to the stack. To this end, each variable on the stack is equipped with a read/write permission.

Proving that the resulting operation to combine states consisting of these extended stacks and classical heaps forms a separation algebra took some time but is straight-forward. However, there were some problems with formalising the programming language. The main problem is locality: like most languages the language used in *Variables as Resource* [8] has a concept of local variables. Consider for example the following pseudocode for traversing a list:

```
list_traverse(x) {
  local t;
  t := x;
  if (t != NULL) then {
     list_traverse(t.next_list_element);
     do something with data in t;
  } else {done}
}
```

The local variable `t` needs to address a new and different location on the stack in each recursive call of `list_traverse`. A common solution is to informally demand that a *fresh* variable `z` is chosen every time and `t` is replaced by `z`. However, there are some problems formalising this notion of freshness in an intuitive way.

Abstract separation logic does not provide a notion of locality itself. However, it contains nondeterministic choice and assumptions. These can be used to nondeterministically choose a variable that is not present in the current state. Being not present in the current state is sufficient; however, it is not straight-forward to see that it does not matter if a variable is chosen that will be used in the future by a different part of the program. This semantics for local variables is abstract, fits easily into the framework, leads to simple proofs for corresponding inference rules etc. Unfortunately, it is not easy to see that this semantics is really the one intended. It's not intuitive.

In my opinion it is preferable to use a semantics that explicitly hides the current value when the scope of a local variable is entered and restores it afterwards. However, hiding and restoring are not local actions. Moreover, these hiding and restoring operations would need to respect threads somehow, since using a local variable t in one thread should not influence the usage of another variable t in a different thread. It would require serious modifications to allow variables to have different values depending on the thread that interprets them. Even if one is willing to apply such modifications, it is not obvious how such changes might look like. Should one just add thread identifiers and leave the details completely to the instantiations? This would keep the abstract level nice and clean, but cause a large part of the formalisations to be repeated over and over again, since probably most instantiations need a concept similar to local variables. One could try to split the current global state into one global state and local states for each thread. That is a common view and would simplify instantiations. However, the abstract part of the formalisation would become much more complicated and harder to understand.

Currently, variables denote explicit locations on the stack. So one has to manipulate either the program by replacing the variable with a different one or one has to modify the stack. Both ways are causing problems as explained above. However, one could add an extra layer of abstraction: namespaces. A namespace is a

function mapping names to values. Variables could just be names that need a namespace to be interpreted as locations on the stack. In different parts of the code or different threads the same name could point to different locations. There would be just one global state instead of the complicated structure above. Admittedly, there would be many namespaces, but their structure is very simple and operations to manipulate namespaces could be very limited. Adding namespaces would not complicate the abstract level much, but provide the needed functionality. Moreover, the concept is abstract enough to be used for other purposes as well. It could for example solve very similar problems with locally defined recursive functions.

During the discussions about the semantics of local variables, it turned out that it might be beneficial to make the model of parallel computation more suited for real world programming languages. To this end, one could switch to C-like fork/join parallelism. However, this would involve huge modifications. A rather minor change that turned out to be probably useful is to change synchronisation to conditional critical regions.

5 Current Work

Apart from these problems with locality, I did not experience major trouble with the first instantiation. However, I did not formalise the formalism described in *Variables as Resource* [8] exactly. The formalism described there allows locally defined mutually recursive functions. This causes problems in the current setting that could be overcome using namespaces as described above. To keep things simple at the beginning, I started formalising a mixture of the logic described in this paper with the one used by Smallfoot [2] as a case study.

Smallfoot is a completely automated tool that allows to verify specifications of programs written in a simple imperative language. This language know mutually recursive functions, but no local function definitions. Compared with other tools and implementations the separation logic used is weak. Thus, it's a good starting example. I use the Smallfoot syntax, but a semantics that is much closer to the one used by *Variables as Resource* [8] than the one used by Smallfoot. In particular, the stack variables are treated as resource, i. e. they are equipped with permissions. The goal of this case study is to develop a tool, that parses Smallfoot input files and is able to verify the specifications completely automatic inside the HOL theorem prover. This goal has nearly been achieved.

The tool is already able to verify some input files completely automatic. However, some features of Smallfoot are still missing. The biggest missing feature are conditional critical regions. Support for semaphores is available at the abstract level, but there is no support for any kind of synchronisation in the Smallfoot instantiation yet. Moreover, there is no support for existantically quantified specifications. As existential quantification is used internally however, it is simple to add this feature. It would be straight-forward to add some more predicates used by Smallfoot like double linked as well. I guess that it would take a few more weeks to implement all missing features of Smallfoot.

6 Conclusion and Future Work

I already took some initial steps towards implementing a framework for separation logic inside the HOL theorem prover:

- I formalised abstract separation logic as described in the original paper [4].
- This formalisation contains extensions of abstract separation logic by
  - procedure calls,
  - standard separation logic operators,
  - common imperative programming constructs.
- I formalised a big part of the logic described in *Variables as Resource* [8].
- As a case study, I implemented a completely automated tool, that uses a language very similar to the one used by Smallfoot:
  - I instantiated the framework to the needed specification and programming languages.

- There are proofs of specialised inference rules.
- A parser for reading Smallfoot specifications has been implemented.
- Specialised tactics have been implemented to verify these parsed specifications.

While this completed work can just be considered as initial steps towards the proposed framework, it already took a significant amount of effort. There are about 30000 lines of proofs and about 6000 lines of ML for the automation.

Currently, I try to complete the Smallfoot case study. Once it is completed I plan to modify the abstract part, i. e. the formalisation of abstract separation logic according to what I learned:

– As motivated above, I plan to add namespaces.
– Replacing the parallel composition provided by abstract separation logic with C-like fork/join parallelism is another planned task.

In general the abstract part should become more powerful, better suited for real programming languages while still keeping as abstract as possible. More ideas for modifications may arise during the current case study, but even now, a large part of the current formalisation is due to be modified.

After finishing these modifications I plan to try some more case studies like the ones described in the introduction and iterate the process of adapting the underlying abstract semantics.

Acknowledgements

I would like to thank Matthew Parkinson, Mike Gordon, Alexey Gotsman, Magnus Myreen and Viktor Vafeiadis for a lot of discussions, comments and criticism.

References

1. A.W. Appel and S. Blazy. Separation logic for small-step Cminor. In K. Schneider and J. Brandt, editors, *International Conference on Theorem Proving in Higher Order Logics (TPHOL)*, volume 4732 of *LNCS*, pages 5–21, Kaiserslautern, Germany, 2007. Springer.
2. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.
3. Stephen Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007.
4. Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS '07: Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science*, pages 366–378, Washington, DC, USA, 2007. IEEE Computer Society.
5. Andrew Ireland et al. Core: Cooperative reasoning for automatic software verification. http://www.macs.hw.ac.uk/core.
6. Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Towards formal verification of memory properties using separation logic. In *22nd Workshop of the Japan Society for Software Science and Technology, Tohoku University, Sendai, Japan, September 13–15, 2005*. Japan Society for Software Science and Technology, Sep. 2005.
7. Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL'01*, volume 2142, pages 1–19, 2001.
8. Matthew Parkinson, Richard Bornat, and Cristiano Calcagno. Variables as resource in hoare logics. In *LICS '06: Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science*, pages 137–146, Washington, DC, USA, 2006. IEEE Computer Society.
9. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.
10. Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 97–108, New York, NY, USA, 2007. ACM.
11. Tjark Weber. Towards mechanized program verification with separation logic. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *CSL*, volume 3210 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2004.