A Formalisation of Smallfoot 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. In this paper a general framework for separation logic inside the HOL theorem prover is presented. This framework is based on Abstract Separation Logic. It contains a model of an abstract, imperative programming language as well as an abstract specification logic for this language. While the formalisation mainly follows the original definition of Abstract Separation Logic, it contains some additional features. Most noticeably is added support for procedures.
As a case study, the framework is instantiated to build a tool that is able to parse Smallfoot specifications and verify most of them completely automatically. In contrast to Smallfoot this instantiation can handle the content of data-structures as well as their shape. This enables it to verify fully functional specifications. Some noteworthy examples that have been verified are parallel mergesort and an interactive filter-function for single linked lists.

1 Motivation

Separation logic is an extension of Hoare logic that allows local reasoning [7, 9]. It is used to reason about mutable data structures in combination with low level imperative programming languages that use pointers and explicit memory management. Thanks to local reasoning, it scales better than classical Hoare logic to the verification of large programs and can easily be used to reason about parallelism. There are several implementations: Smallfoot [2], SLAyer[1] and SpaceInvader [5] are probably some of the best know examples. Moreover, there are formalisations inside theorem provers [1, 6, 10, 11].

The problem, as I see it, is that all these tools and formalisations focus on one concrete setting. They fix the programming languages, their exact semantics, the supported specifications etc. However, there are a lot of different possible design choices and the tools differ in these. I'm therefore building a general framework for separation logic in HOL that can be instantiated to a variety of different separation logics. By building such a framework, I hope to be able to concentrate on the essence of separation logic as well as keeping the formalisation clean and easy.

In this paper, the results of these efforts to build a separation logic framework in HOL are presented. The framework is based on *Abstract Separation*

[1] `http://research.microsoft.com/SLAyer/`

Logic [4], an abstract, high level variant of separation logic. It consists of both an abstract, imperative programming language and an abstract specification logic for this language. Both the abstract language and the specification logic are designed to be instantiated to a concrete programming language and a concrete language for specifications.

As a case study, I instantiated this framework to build a tool similar to Smallfoot [2], one of the oldest and best documented separation logic tools. Smallfoot is able to automatically prove specifications about programs written in a simple, low-level imperative language with support for parallelism. The tool, called Holfoot, combines ideas from Abstract Separation Logic, *Variables as Resource in Hoare Logic* [8] and Smallfoot. It is able to parse Smallfoot-specifications and prove nearly all of them completely automatically inside the HOL theorem prover. In addition to Smallfoot, specifications can talk about the content of data-structures as well as their shape. Proving the resulting fully functional specifications exploits the fact that Holfoot is implemented inside HOL. All existing libraries and proof tools can be used, while a substantial amount of automation is still available to reason about the structure of the program.

Reasoning about the data-content as well as the shape of data-structures is one of the main challenges of separation logic tools at the moment. To help the communication within the community and in general to further the progress of the field, a benchmark collection called *A Heap of Problems*[2] was created. It collects interesting examples, usually with at least a natural-language description, a C-implementation and some pseudo-code. Often implementations for a specific separation logic tool are available as well.Moreover, there are proofs of the examples using different tools and techniques.

Here, I would like to highlight just two of these benchmark examples: mergesort, whose verification needs some knowledge about orderings and permutations, and filtering of a single linked list, whose iterative version uses a very complicated loop invariant. Both examples can easily be verified using Holfoot. The tool is able to reason automatically about the shape part of the problem, leaving the user to reason about properties of the data-content, i. e. about the essence of these algorithms. Fully functional specifications of simpler algorithms like reversing or copying of a single-linked-list, determining its length or a recursive filter function can even be verified completely automatically. For more examples and discussions about them, please have a lock at the *A Heap of Problems* webpage.

It took considerable effort to build this framework and instantiate it. This work cannot be presented here in detail due to space limitations. Therefore, the next section, will present a high level view on Holfoot. It is intended to give a glimpse of the features and power of this tool. Semantic foundations and implementation details are not discussed. This high level presentation of Holfoot is followed by a detailed description of the formalisation of Abstract Separation Logic in HOL. This description explains the semantic background

[2] `http://wiki.heap-of-problems.org`.

of Holfoot. However, it is barely scratched, how the Abstract Separation Logic framework is instantiated to build Holfoot. The paper ends with a section about future work and some conclusions.

2 Formalisation of Smallfoot

Smallfoot [2] is one of the oldest and best documented separation logic tools. It is able to automatically prove specifications about programs written in a simple, low-level imperative language, which is designed to resemble C. This language contains pointers, local and global variables, dynamic memory allocation/deallocation, conditional execution, while-loops and recursive procedures with call-by-value and call-by-reference arguments. Moreover, there is support for parallelism with conditional critical regions that synchronise the access to so-called resources. Smallfoot-specifications are concerned with the shape of memory. Common specifications, for example, say that some stack-variable points to a single linked list in memory. However, nothing is e. g. said about the length of the list or about its data-content.

Smallfoot comes with a selection of example specifications. There are common algorithms about single linked lists like copying, reversing or deallocating them. Another set of examples contains similar algorithms for trees. There is an implementation of mergesort, some code about queues, circular-lists, buffers and similar examples. Holfoot[3] is able to parse Smallfoot-specifications and prove most of the mentioned examples completely automatically inside the HOL theorem prover.

While some features like local variables or procedures with call-by-value arguments took some effort, and while it turned out to be useful to use explicit permission for stack-variables, it was nevertheless possible to formalise Smallfoot based on Abstract Separation Logic in a natural way. As far as I know, this is the first time Abstract Separation Logic has been used to implement a separation logic tool. The formalisation of Smallfoot illustrates that Abstract Separation Logic is powerful and flexible enough to model languages and specifications used by well-known separation logic tools. Moreover, it demonstrates that it is possible to automate reasoning in this framework. While Holfoot is slower than Smallfoot, it provides the additional assurance of a formal proof inside HOL. That this is really valuable, is underlined by the fact, that an error in Smallfoot was detected while building Holfoot. Due to a bug in its implementation, Smallfoot handles call-by-value parameters like call-by-reference ones.

However, besides a formal foundation and much higher trust in the tool, another advantage of Holfoot is, that it is straightforward to use all the libraries and proof-tools HOL provides. Smallfoot specifications talk about the shape of data-structures. The Smallfoot-specification of mergesort for example states that mergesort returns a single linked list. It does not guarantee anything about the content of this list, much less that mergesort really sorts lists. In fact,

[3] Holfoot as well as a collection of examples can be found in the HOL-repository

to prove a fully functional specification of mergesort, substantial knowledge about permutations of lists, orderings and sorted lists is needed. Here, the existing infrastructure of HOL is very useful.

Once the formalisation of the features provided by Smallfoot was completed, it was straight-forward to extend it with support for the content of data-structures. This allows the verification of fully functional specifications. Holfoot is able to automatically verify fully functional specifications of simple algorithms like list-reversal, list-copy or list-length:

```
list_copy(z;c) [data_list(c,data)] {
  local x,y,w,d;
  if (c == NULL) {z=NULL;}
  else {
    z=new(); z->tl=NULL;
    x = c->dta; z->dta = x;
    w=z;
    y=c->tl;
    while (y != NULL) [
       data_lseg(c,
         ``_data1++[_cdate]``,y) *
       data_list(y,_data2) *
       data_lseg(z,_data1,w) *
       w |-> tl:0,dta:_cdate *
       ``data:num list =
         _data1 ++ _cdate::_data2``] {
      d=new(); d->tl=NULL;
      x=y->dta; d->dta=x;
      w->tl=d; w=d;
      y=y->tl;
    }
  }
} [data_list(c,data) * data_list(z,data)]
```

```
list_reverse(i;) [data_list(i,data)] {
  local p, x;
  p = NULL;
  while (i != NULL) [
       data_list(i,_idata) *
       data_list(p,_pdata) *
       ``(data:num list) =
         (REVERSE _pdata) ++ _idata``] {
    x = i->tl; i->tl = p; p = i; i = x;
  }
  i = p;
} [data_list(i,``REVERSE data``)]
```

```
list_length(r;c) [data_list(c,cdata)] {
  local t;
  if (c == NULL) {r = 0;} else {
    t = c->tl;
    list_length(r;t);
    r = r + 1;
  }
} [data_list(c,cdata) *
   r == ``LENGTH (cdata:num list)``]
```

The syntax of the this pseudo-code used by Smallfoot and Holfoot is indented to be close to C. However, there are some uncommon features: the arguments of a procedure before the semicolon are call-by-reference arguments, the others call-by-reference ones. So the argument `z` of `list_copy` is a call-by-reference argument, whereas `c` is a call-by-value argument. The pre- and post-conditions of procedures are denoted in brackets around the procedure's body. Similarly, loops are annotated with their invariant. In specifications, a variable name that starts with an underscore denotes an existentially quantified variable. For example, `data1`, `data2` and `cdate` are existentially quantified in the loop-invariant of copy. This invariant requires that `data` can somehow be split into these three. How it is split changes from iteration to iteration. Finally, everything within quotation marks is regarded as a HOL term. So, `REVERSE` or `LENGTH` are not part of the Smallfoot formalisation but functions from HOL's list library.

While these simple algorithms can be handled completely automatically, more complicated ones like the aforesaid mergesort need user interaction. However, even in these interactive proofs, there is a clear distinction between reasoning about the content and about the shape. While the shape can mostly be handled automatically, the user is left to reason about properties of the content. Let's consider the following specification of parallel mergesort:

```
merge(r;p,q) [data_list(p,pdata) *
    data_list(q,qdata) *
    ``(SORTED $<= pdata) /\
      (SORTED $<= qdata)``] {
  local t, q_date, p_date;
  if (q == NULL) r = p;
  else if(p == NULL) r = q;
  else {
    p_date = p->dta;
    q_date = q->dta;
    if (q_date < p_date) {
      t = q; q = q->tl;
    } else {
      t = p; p = p->tl;
    }
    merge(r;p,q);
    t->tl = r; r = t;
  }
} [data_list(r,_rdata) *
   ``(SORTED $<= _rdata) /\
     (PERM (pdata ++ qdata) _rdata)``]

split(r;p) [data_list(p,data)] {
  local t1,t2;
  if (p == NULL) r = NULL;
  else {
    t1 = p->tl;
    if (t1 == NULL) r = NULL;
    else {
      t2 = t1->tl;
      split(r;t2);
      p->tl = t2;
      t1->tl = r;
      r = t1;
    }
  }
} [data_list(p,_pdata) *
   data_list(r,_rdata) *
   ``PERM (_pdata ++ _rdata) data``]

mergesort(r;p) [data_list(p,data)] {
  local q,q1,p1;
  if (p == NULL) r = p;
  else {
    split(q;p);
    mergesort(q1;q) || mergesort(p1;p);
    merge(r;p1,q1);
  }
} [data_list(r,_rdata) *
   ``(SORTED $<= _rdata) /\
     (PERM data _rdata)``]
```

Holfoot can automatically reduce this fully functional specification of mergesort to a small set of simple verification conditions. These verification conditions are just concerned with permutations and sorted lists. The whole structure of the program and the shape of the data-structures can be handled automatically. Some of the remaining verification conditions are very simple as for example `SORTED $<= x::xs ==> SORTED xs`. Others require some knowledge about permutations like `PERM (x::(xs ++ ys)) l ==> PERM (x::(xs ++ y::ys)) (y::l)`. However, most of them can easily be handled by automated proof tools for permutations and orderings. The only remaining verification conditions are of the form

```
SORTED $<= x::xs  /\  SORTED $<= y::ys  /\ SORTED $<= l  /\
y < x  /\  PERM l (x::xs++ys)  ==> SORTED $<= y::l
```

Their proof needs a combination of properties of permutations and sorted lists. Thus, the standard proof tools fail and a tiny manual proof is required. The following proof-script is sufficient to prove the given specification of mergesort:

```
val thm = smallfoot_verbose_prove(mergesort-specification-filename,
   SMALLFOOT_VC_TAC THEN
   ASM_SIMP_TAC (arith_ss++PERM_ss) [SORTED_EQ, SORTED_DEF, transitive_def] THEN
   REPEAT STRIP_TAC THEN (
      IMP_RES_TAC PERM_MEM_EQ THEN
      FULL_SIMP_TAC list_ss [] THEN
      RES_TAC THEN ASM_SIMP_TAC arith_ss []
   ));
```

After parsing and preprocessing the specification stored in the given file, verification conditions are generated using SMALLFOOT_VC_TAC. This single call is sufficient to eliminate the whole program structure and leave just the described verification conditions. The next line calls some proof-tools for permutations and sorted lists and is able to discharge most of the verification conditions. The rest of the proof-script handles the remaining verification conditions which are all of the aforesaid form.

As this example illustrates, human interaction is often only needed to reason about the essence of an algorithms and HOL provides powerful tools to aid this reasoning. This shows the power of Holfoot and with it the flexibility and power of the whole framework.

3 Formalisation of Abstract Separation Logic

In the previous section, a high-level view of Holfoot and with it of the framework and its capabilities was presented. In this section its semantic foundations – *Abstract Separation Logic* [4] – will be explained. This explanation follows closely the HOL formalisation[4].

Abstract Separation Logic abstracts from both the concrete states and the concrete programming language. Instead of using a concrete model of memory consisting usually of a stack and a heap, Abstract Separation Logic uses an abstract set of states Σ. A partial function $\circ$, called *separation combinator*, is used to combine states.

Definition 1 (Separation Combinator). A *separation combinator* $\circ$ is a partially defined function that satisfies the following properties:

- $\circ$ is partially associative
- $\circ$ is partially commutative
- $\circ$ is cancellative, i. e.
  $\forall s_1, s_2, s_3.\ \text{Defined}(s_1 \circ s_2)\ \wedge\ (s_1 \circ s_2 = s_1 \circ s_3) \Longrightarrow (s_2 = s_3)$ holds
- for all states s there exists a neutral element u_s with $u_s \circ s = s$

Definition 2 (Separateness, Substates). This definition of separation combinators induces notions of *separateness* ($\#$) and *substates* ($\preceq$).

$$s_1 \# s_2 \ \text{ iff } \ s_1 \circ s_2 \text{ is defined} \qquad s_1 \preceq s_3 \ \text{ iff } \ \exists s_2.\ s_3 = s_1 \circ s_2$$

Definition 3 ($*$, *emp*). Predicates are as usual elements of the powerset of states $P(\Sigma)$. This allows to define the spatial conjunction operator $*$ of separation logic and its neutral element *emp* as follows:

$$\begin{aligned} P * Q &:= \{s \mid \exists p, q.\ (p \circ q = s)\ \wedge\ p \in P\ \wedge\ q \in Q\} \\ \mathit{emp} &:= \{u \mid \exists s.\ u \circ s = s\} \end{aligned}$$

$*$ forms together with *emp* a commutative monoid. Other standard separation logic constructs can be defined in a natural way as well. There is a shallow embedding of the most common constructs available in the framework. Additional constructs can be added easily.

In order to instantiate the framework, one has to provide a concrete set of states Σ and a concrete separation combinator $\circ$.

[4] The sources can be found in the HOL - repository at Sourceforge in the subdirectory `examples/separationLogic`.

Example 4. Heaps, modelled as finite partial functions, are commonly used with separation logic. In this model, Σ is the set of all heaps and $\circ$ is given by

$$h_1 \circ h_2 = \begin{cases} h_1 \uplus h_2 & \text{iff } \mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset \\ \textit{undefined} & \text{otherwise} \end{cases}$$

In this setting, two heaps are disjoint ($h_1 \ \# \ h_2$) iff their domains are disjoint. The combination of two separate heaps ($h_1 \circ h_2$) is their disjoint union. The empty heap is the neutral element for all heaps.

3.1 Actions

The programming language used by Abstract Separation Logic is abstract as well. Its elementary constructs are actions.

Definition 5 (Action). An *action act* : $\Sigma \to P(\Sigma)^\top$ is a function from a state to a set of states or a special failure state $\top$.

If executing an action *act* in a state s results in $\top$, then an error may occur during the execution of the action. Otherwise, if $\mathit{act}(s)$ results in a set of states S, no error can occur and executing the action will nondetermistically lead to one of the states in S. The empty set can be used to model actions that do not terminate. Actions can be combined to form new actions. The most common combination is consecutive execution:

$$(\mathit{act}_1; \mathit{act}_2)(s) = \begin{cases} \top & \text{if } \mathit{act}_1(s) = \top \\ \top & \text{if } \exists s'.\ s' \in \mathit{act}_1(s) \ \wedge \ \mathit{act}_2(s') = \top \\ \displaystyle\bigcup_{s' \in \mathit{act}_1(s)} \mathit{act}_2(s') & \text{otherwise} \end{cases}$$

Another common combination is nondeterministic choice:

$$\left(\bigsqcup_{\mathit{act} \in \mathit{act\text{-}set}} \mathit{act} \right)(s) = \begin{cases} \top & \text{if } \exists \mathit{act} \in \mathit{act\text{-}set}.\ \mathit{act}(s) = \top \\ \displaystyle\bigcup_{\mathit{act} \in \mathit{act\text{-}set}} \mathit{act}(s) & \text{otherwise} \end{cases}$$

$$\mathit{act}_1 + \mathit{act}_2 = \bigsqcup_{\mathit{act} \in \{\mathit{act}_1,\ \mathit{act}_2\}} \mathit{act}$$

Definition 6 (Semantic Hoare Triples). For predicates P, Q and an action *act*, a *semantic Hoare triple* $\ll \ P \ \gg \mathit{act} \ll \ Q \ \gg$ holds, iff for all states p that satisfy the *precondition* P the action does not fail, i. e. $\forall p \in P.\ \mathit{act}(p) \neq \top$, and leads to a state that satisfies the *postcondition* Q, i. e. $\forall p \in P.\ \mathit{act}(p) \subseteq Q$. Notice, that this describes partial correctness, since a Hoare triple is trivially satisfied, if *act* does not terminate, i. e. if $\mathit{act}(s) = \emptyset$ holds.

Local reasoning is an essential feature of separation logic. It allows to extend a specification with an arbitrary context:

$$\frac{\ll P \gg \ \mathit{act} \ \ll Q \gg}{\ll P * R \gg \ \mathit{act} \ \ll Q * R \gg}$$

In order to provide local reasoning, only those actions are considered whose specifications can be safely extended using this inference rule. These actions are called *local*.

Definition 7 (Local Actions). An action *act* is called *local*, iff for all states s, s_1, s_2 with $s = s_1 \circ s_2$ and $act(s_1) \neq \top$ the evaluation of the action on the extended state does not fail ($act(s) \neq \top$) and $act(s) \subseteq act(s_1) * \{s_2\}$ holds.

The *skip* action defined by $skip(s) := \{s\}$ is a simple example of a local action. Other examples are $diverge(s) := \emptyset$ or $fail(s) := \top$. Sequential composition and nondeterministic choice preserve locality. The set of local actions forms together with the following order a complete lattice.

Definition 8 (Order of Actions). $act_1 \sqsubseteq act_2$ iff act_2 allows more behaviour than act_1, i. e. iff $\forall s.\ (act_2(s) = \top) \ \vee \ (act_1(s) \subseteq act_2(s))$ holds. Notice that this is equivalent to $\forall P, Q.\ \ll P \gg\ act_2 \ll Q \gg \Longrightarrow \ll P \gg\ act_1 \ll Q \gg$.

This lattice of local actions is used to define a *best local action* as an infimum of local actions in this lattice. The HOL formalisation contains the corresponding definitions and theorems. However, here the discussion of this lattice is skipped. Instead an equivalent, high level characterisation is used.

Definition 9 (Best Local Action). Given a precondition P and a postcondition Q the *best local action* $bla[P, Q]$ is the most general local action that satisfies $\ll P \gg\ bla[P, Q] \ll Q \gg$. This means:

- $bla[P, Q]$ is a local action
- $\ll P \gg\ bla[P, Q] \ll Q \gg$ holds
- $bla[P, Q]$ is more general than any local actions *act* with $\ll P \gg\ act \ll Q \gg$, i. e. $act \sqsubseteq bla[P, Q]$

One common use of the best local action *bla* are the materialisation and annihilation actions. $materialise(P) := bla[emp, P]$ can be used to materialise some new part of the state that satisfies the predicate P. Similarly, $annihilate(P) := bla[P, emp]$ is used to annihilate some part of the state that satisfies P. Notice, that for certain P the annihilation $annihilate(P)$ behaves unexpectedly. If there is more than one substate that satisfies P, then $annihilate(P)$ diverges. Therefore, usually just *precise* predicates are used with annihilation:

Definition 10 (Precise Predicates). A predicate P is called *precise* iff for every state there is at most one substate that satisfies P.

As shown by the examples of materialisation and annihilation, *bla* is useful to define local actions. Often it is however necessary to relate the pre- and postcondition. For example, the postcondition of an action that increments the value of a variable needs to refer to the old value of this variable. This leads to the following extension of best local actions:

Definition 11 (Quantified Best Local Action). Given two functions $P_{(\cdot)}$ and $Q_{(\cdot)}$ that map some argument type to predicates the *quantified best local action* (*qbla*) is the most general local action that satisfies

$$\forall arg.\ \ll P_{arg} \gg\ qbla[P_{(\cdot)}, Q_{(\cdot)}] \ll Q_{arg} \gg$$

Another useful local action is *assume*. Given a predicate, *assume* skips if the predicate holds and diverges if it does not hold. In the next section, *assume* is used in combination nondeterministic choice and Kleene star to model conditional execution and loops. In order to be a local action, the predicate has to be intuitionistic, though.

Definition 12 (Intuitionistic Predicate). A predicate P is called *intuitionistic*, iff $P * true = P$ holds. This means that iff P holds for a state s, then it holds for all superstates $s' \succeq s$ as well. The *intuitionistic negation* $\neg_i P$ holds in a state s, if P does not hold for all superstates $s' \succeq s$. P is called *decided* in a set of states S, iff $\forall s \in S.\ s \in P\ \vee s \in \neg_i P$ holds.

For an intuitionistic predicate P the local action *assume*(P) can be defined as

$$assume(P)(s) = \begin{cases} \{s\} & \text{if } s \in P \\ \emptyset & \text{if } s \in \neg_i P \\ \top & \text{otherwise} \end{cases}$$

3.2 Programs

This notion of local actions is extended to an abstraction of an imperative programming language. The basic constructs of this language are local actions. Besides local actions, the language contains the usual control structures like conditional execution and while-loops. Additionally, nondeterminism, concurrency and semaphores are supported. The definition of the semantics of this language follow ideas from Brooks [3] about Concurrent Separation Logic. Programs are translated to a set of traces that capture all possible interleavings during concurrent execution. The semantics of a program is given by nondeterministic choice between the semantics of its traces. As an additional layer of abstraction proto-traces are used between programs and traces.

Definition 13 (Proto-Trace). The set of *proto-traces PTr* is inductively defined to be the smallest set with

- $act \in PTr$ for all local actions *act*
- $pt_1\ ;\ pt_2 \in PTr$ (sequential composition) for $pt_1, pt_2 \in PTr$
- $pt_1 \,||\, pt_2 \in PTr$ (parallel composition) for $pt_1, pt_2 \in PTr$
- proccall$(name, arg) \in PTr$ (procedure call) for all procedure-names *name* and all arguments *arg*
- $l.pt \in PTr$ (lock declaration) for a lock l and $pt \in PTr$
- *with* l *do* $pt \in PTr$ (critical region) for a lock l and $pt \in PTr$

Definition 14 (Program). A program is a set of proto-traces. The set of all programs is denoted by *Prog*.

Definition 15 (Atomic Action). An *atomic action* is either a local action, a check $\mathit{check}(\mathit{act}_1, \mathit{act}_2)$ for local actions $\mathit{act}_1, \mathit{act}_2$ or a lock operation $P(l)$ or $V(l)$ for a lock l.

Definition 16 (Trace). A trace is a list of atomic actions. Let ϵ denote the empty trace. The concatenation of two traces t_1, t_2 is denoted as $t_1 \cdot t_2$.

To define the traces of a program, an environment is needed that fixes the semantics of procedure calls.

Definition 17 (Procedure Environment). A *procedure environment* is a finite map $\mathit{penv} : \mathit{procedure\text{-}names} \overset{\text{fin}}{\rightharpoonup} \mathit{arguments} \to \mathit{Prog}$ from procedure-names to a function from procedure arguments to programs.

Definition 18 (Traces of Proto-traces). Given an procedure environment *penv*, the traces of a proto-trace t after unfolding procedures n-times with respect to *penv* (denoted as $T^n_{\mathit{penv}}(t)$) are given by:

$$\begin{aligned}
T^n_{\mathit{penv}}(\mathit{act}) &= \{\mathit{act}\} \\
T^n_{\mathit{penv}}(\mathit{pt}_1 \ ; \ \mathit{pt}_2) &= \{t_1 \cdot t_2 \mid t_1 \in T^n_{\mathit{penv}}(\mathit{pt}_1) \ \wedge \ t_2 \in T^n_{\mathit{penv}}(\mathit{pt}_2)\} \\
T^n_{\mathit{penv}}(\mathit{pt}_1 \mid\mid \mathit{pt}_2) &= \bigcup_{t_1 \in T^n_{\mathit{penv}}(\mathit{pt}_1), t_2 \in T^n_{\mathit{penv}}(\mathit{pt}_2)} t_1 \ \mathit{zip} \ t_2 \\
T^n_{\mathit{penv}}(\text{proccall}(\mathit{name}, \mathit{arg})) &= \begin{cases} \{\mathit{fail}\} & \text{if } \mathit{name} \notin \mathit{dom}(\mathit{penv}) \\ \emptyset & \text{if } \mathit{name} \in \mathit{dom}(\mathit{penv}) \wedge n = 0 \\ \bigcup_{\mathit{pt} \in \mathit{penv}(\mathit{name},\mathit{arg})} T^{n-1}_{\mathit{penv}}(\mathit{pt}) & \text{otherwise} \end{cases} \\
T^n_{\mathit{penv}}(l.\mathit{pt}) &= \{\textit{remove-locks(l,t)} \mid t \in T^n_{\mathit{penv}}(\mathit{pt}) \ \wedge \ t \text{ is } l\text{-synchronised}\} \\
T^n_{\mathit{penv}}(\mathit{with}\ l\ \mathit{do}\ \mathit{pt}) &= \{P(l) \cdot t \cdot V(l) \mid t \in T^n_{\mathit{penv}}(\mathit{pt})\}
\end{aligned}$$

In this definition, *remove-locks(l,t)* removes all atomic actions concerned with the lock l, i. e. $P(l)$ and $V(l)$, from the trace t. A trace is *l-synchronised*, iff the lock-actions $P(l)$ and $V(l)$ are properly aligned. Finally, the auxiliary function *zip* builds all interleavings of two traces. It is given by

$$\begin{aligned}
\textit{add-check}(a_1, a_2, t) &= \begin{cases} \mathit{check}(a_1, a_2) \cdot t & \text{if } a_1 \text{ and } a_2 \text{ are local actions} \\ t & \text{otherwise} \end{cases} \\
\epsilon \ \mathit{zip} \ t = t \ \mathit{zip} \ \epsilon &= \{t\} \\
(a_1; t_1) \ \mathit{zip} \ (a_2; t_2) &= \begin{array}{l} \big\{\textit{add-check}(a_1, a_2, t) \mid t \in \{a_1; u \mid u \in t_1 \ \mathit{zip} \ (a_2; t_2)\} \cup \\ \qquad\qquad\qquad\qquad\qquad \{a_2; u \mid u \in (a_1; t_1) \ \mathit{zip} \ t_2\}\big\} \end{array}
\end{aligned}$$

Finally, the traces of a proto-trace *pt* and a program p with respect to *penv* are defined as

$$T_{\mathit{penv}}(\mathit{pt}) = \bigcup_{n \in \mathbb{N}} T^n_{\mathit{penv}}(\mathit{pt}) \qquad T_{\mathit{penv}}(p) = \bigcup_{\mathit{pt} \in p} T_{\mathit{penv}}(\mathit{pt})$$

It remains to define the semantics of traces. Local actions in traces are just interpreted by themselves. Checks are added to enforce race-freedom. The semantics of lock actions is however more complicated.

One central idea behind Concurrent Separation Logic is to split the state into parts for each thread and each lock: a lock protects a part of the state. If a thread holds a lock, it can access this state, otherwise it cannot. Therefore, a precise predicate called *lock invariant* is associated with each lock. This invariant abstracts the part of the state that is protected by the lock. *materialise* and *annihilate* actions are used to make this abstracted state accessible/inaccessible.

Definition 19 (Semantics of Atomic Actions). The semantics of an atomic action with respect to a *lock-environment* $\mathit{lenv} : \mathit{locks} \to P(\Sigma)$ is given by

$$\begin{aligned}
[\![\mathit{act}]\!]_{\mathit{lenv}} &= \mathit{act} \\
[\![\mathit{check}(\mathit{act}_1, \mathit{act}_2)]\!]_{\mathit{lenv}}(s) &= \begin{cases} \{s\} & \text{if } \exists s_1, s_2.\ s = s_1 \circ s_2 \wedge \\ & \qquad \mathit{act}_1(s_1) \neq \top \ \wedge\ \mathit{act}_2(s_2) \neq \top \\ \top & \text{otherwise} \end{cases} \\
[\![P(l)]\!]_{\mathit{lenv}} &= \mathit{materialise}(\mathit{lenv}(l)) \\
[\![V(l)]\!]_{\mathit{lenv}} &= \mathit{annihilate}(\mathit{lenv}(l))
\end{aligned}$$

Notice, that the semantics of an atomic action is a local action.

Definition 20 (Semantics of Traces, Programs). The semantics of a trace with respect to a lock-environment is the sequential combination of the semantics of its atomic actions. The semantics of a program is given by the nondeterministic choice between the semantics of its traces.

$$[\![\epsilon]\!]_{\mathit{lenv}} = \mathit{skip} \qquad [\![a \cdot t]\!]_{\mathit{lenv}} = [\![a]\!]_{\mathit{lenv}} \ ; \ [\![t]\!]_{\mathit{lenv}} \qquad [\![\mathit{prog}]\!]_{(\mathit{penv},\mathit{lenv})} = \bigsqcup_{t \in T_{\mathit{penv}}(\mathit{prog})} [\![t]\!]_{\mathit{lenv}}$$

Notice that the semantics of a program is a always a local action. This allows concepts for actions to be easily lifted to programs:

Definition 21 (Hoare triple). A *Hoare triple* $\rhd_{(\mathit{penv},\mathit{lenv})}\ \{P\}\ \mathit{prog}\ \{Q\}$ holds, iff $\ll P \gg [\![\mathit{prog}]\!]_{(\mathit{penv},\mathit{lenv})} \ll Q \gg$ holds. If a Hoare triple holds for all environments, it is written as $\{P\}\ \mathit{prog}\ \{Q\}$.

Definition 22 (Program Abstractions). A program p_2 is an abstraction of a program p_1 with respect to some environment *env* (denoted as $p_1 \sqsubseteq_{\mathit{env}} p_2$), iff $[\![p_1]\!]_{\mathit{env}} \sqsubseteq [\![p_2]\!]_{\mathit{env}}$ holds.

3.3 Programming Constructs

In the previous section a concept of programs has been introduced. However, these programs hardly resemble the usual programs written in imperative languages. Common constructs like loops or conditional execution are missing. However, these can be easily defined.

Every proto-trace *pt* can be regarded as the program $\{pt\}$. This immediately enriches the programming language with procedure calls and local actions. In particular, one can use *skip*, *fail*, *assume*, *diverge*, *bla* and *qbla* as programs. A lot of instructions can easily be defined using *bla* or *qbla*. Given some suitable definitions for a state containing a stack, one could for example define an instruction that increments a variable as $x{+}{+} = \mathit{qbla}[\lambda c.\ x = c, \lambda c.\ x = (c+1)]$.

The HOL-formalisation uses a shallow embedding of local actions. So, any function $f : \Sigma \to P(\Sigma)^\top$ can be used as a program. However, to enforce that just local actions are used, f is implicitly replaced by *fail*, if it is not local. The other constructs for proto-traces can be lifted to programs as well:

$$\begin{aligned}
p_1 \ ;\ p_2 &= \{pt_1 \ ;\ pt_2 \mid pt_1 \in p_1 \cup \{\mathit{diverge}\} \ \wedge\ pt_2 \in p_2 \cup \{\mathit{diverge}\}\} \\
p_1 \mid\mid p_2 &= \{pt_1 \mid\mid pt_2 \mid pt_1 \in p_1 \ \wedge\ pt_2 \in p_2\} \\
l.p &= \{l.pt \mid pt \in p\} \\
\mathit{with}\ l\ \mathit{do}\ p &= \{\mathit{with}\ l\ \mathit{do}\ pt \mid pt \in p\}
\end{aligned}$$

Some other constructs that are not available for proto-traces can be defined using the fact that programs are just sets of proto-traces. A simple example is nondeterministic choice: $p_1 + p_2 := p_1 \cup p_2$. In combination with *assume* and sequential composition of programs, this can be used to define conditional execution:

$$\text{if } B \text{ then } p_1 \text{ else } p_2 = (\mathit{assume}(B); p_1)\ +\ (\mathit{assume}(\neg_i B); p_2)$$

This definition of conditional execution might seem weird. Remember however, that the framework is just interested in partial correctness. Therefore, it is fine to nondetermistically choose between paths and then diverge, if the wrong choice has been made.

Loops can be defined in a similar manner. However, to define loops, Kleene star is needed:

$$p^0 = \mathit{skip} \qquad p^{n+1} = p \ ;\ p^n \qquad p^* = \bigcup_{n \in \mathbb{N}} p^n$$
$$\text{while } B \text{ do } p = (\mathit{assume}(B); p)^* \ ;\ \mathit{assume}(\neg_i B)$$

This time, one chooses nondetermistically, how often one needs to go around the loop. If the wrong number of iterations is picked, the trace is aborted by one of the *assume* statements.

Notice the definition of Kleene star. It is represented as a shallow embedding in HOL. Moreover, it uses nondeterministic choice over an infinite set of proto-traces. This simple example illustrates how flexible and powerful the combination of shallow and deep embeddings is. Depending on the needs of a concrete instantiation this power and flexibility can be used to define more constructs.

3.4 Inference Rules

Using the semantics of Abstract Separation Logic as presented above, one can deduce high-level inference rules. These inferences are used to verify specification on a high-level of abstraction instead of breaking every proof down to the semantic foundations. Some important inference rules, that are valid in Abstract Separation Logic are:

$$\frac{\begin{array}{c}P_2 \Rightarrow P_1 \qquad Q_1 \Rightarrow Q_2 \\ \triangleright_{\mathit{env}}\ \{P_1\}p\{Q_1\}\end{array}}{\triangleright_{\mathit{env}}\ \{P_2\}\ p\{Q_2\}} \qquad \frac{\begin{array}{c}p_1 \sqsubseteq p_2 \\ \triangleright_{\mathit{env}}\ \{P\}p_2\{Q\}\end{array}}{\triangleright_{\mathit{env}}\ \{P\}\ p_1\{Q\}}$$

$$\frac{\triangleright_{\mathit{env}}\ \{P\}\ p\ \{Q\}}{\triangleright_{\mathit{env}}\ \{P * R\}\ p\ \{Q * R\}} \qquad \frac{\triangleright_{\mathit{env}}\ \{P\}\ p_1\ \{Q\} \qquad \triangleright_{\mathit{env}}\ \{Q\}\ p_2\ \{R\}}{\triangleright_{\mathit{env}}\ \{P\}\ p_1\ ;\ p_2\ \{R\}}$$

$$\frac{B \text{ is decided in } P}{\triangleright_{\mathit{env}}\ \{P\}\ \mathit{assume}(B)\ \{P\ \wedge\ B\}} \qquad \frac{}{\triangleright_{\mathit{env}}\ \{P_{\mathit{arg}}\}\ \mathit{qbla}[P_{(\cdot)}, Q_{(\cdot)}]\{Q_{\mathit{arg}}\}}$$

$$\frac{\triangleright_{\mathit{env}}\ \{P\}\ p\ \{P\}}{\triangleright_{\mathit{env}}\ \{P\}\ p^*\ \{P\}} \qquad \frac{\triangleright_{\mathit{env}}\ \{P\}\ p_1\ \{Q\} \qquad \triangleright_{\mathit{env}}\ \{P\}\ p_2\ \{Q\}}{\triangleright_{\mathit{env}}\ \{P\}\ p_1 + p_2\ \{Q\}}$$

$$\frac{\begin{array}{c}\mathit{name} \in \mathit{dom}(\mathit{penv}) \\ \triangleright_{(\mathit{penv},\mathit{lenv})}\ \{P\}\ \mathit{penv}(\mathit{name}, \mathit{arg})\ \{Q\}\end{array}}{\triangleright_{(\mathit{penv},\mathit{lenv})}\ \{P\}\ \text{proccall}(\mathit{name}, \mathit{arg})\{Q\}} \qquad \frac{\begin{array}{c}B \text{ is decided in } P \\ \triangleright_{\mathit{env}}\ \{B \wedge P\}\ p_1\ \{Q\} \\ \triangleright_{\mathit{env}}\ \{\neg_i B \wedge P\}\ p_2\ \{Q\}\end{array}}{\triangleright_{\mathit{env}}\ \{P\}\ \text{if } B \text{ then } p_1 \text{ else } p_2\ \{Q\}}$$

$$\frac{\begin{array}{c}B \text{ is decided in } P \\ \triangleright_{\mathit{env}}\ \{B \wedge P\}\ p\ \{P\}\end{array}}{\triangleright_{\mathit{env}}\ \{P\}\ \text{while } B \text{ do } p\ \{\neg_i B \wedge P\}} \qquad \frac{\begin{array}{c}\triangleright_{\mathit{env}}\ \{P_1\}\ p_1\ \{Q_1\} \\ \triangleright_{\mathit{env}}\ \{P_2\}\ p_2\ \{Q_2\}\end{array}}{\triangleright_{\mathit{env}}\ \{P_1 * P_2\}\ p_1 \mid\mid p_2\ \{Q_1 * Q_2\}}$$

$$\frac{\begin{array}{c}\mathit{lenv}(l) = r \\ \triangleright_{(\mathit{penv},\mathit{lenv})}\ \{P\}\ p\ \{Q\}\end{array}}{\triangleright_{(\mathit{penv},\mathit{lenv})}\ \{P * r\}\ l.p\ \{Q * r\}} \qquad \frac{\begin{array}{c}\mathit{lenv}(l) = r \\ \triangleright_{(\mathit{penv},\mathit{lenv})}\ \{P * r\}\ p\ \{Q * r\}\end{array}}{\triangleright_{(\mathit{penv},\mathit{lenv})}\ \{P\}\ \mathit{with}\ l\ \mathit{do}\ p\{Q\}}$$

These inference rules are very useful. However, the reader might notice, that there is a problem with recursive functions. The inference rule that handles procedure-calls replaces the call with the definition of the procedure. This is fine for non-recursive functions. However, an implicit induction is needed for recursive functions.

Definition 23 (Procedure Specification). A *procedure specification* consists of a lock-environment *lenv*, a procedure-environment *penv* and specification functions $P_{(\cdot,\cdot,\cdot)}$, $Q_{(\cdot,\cdot,\cdot)}$. It holds, iff all procedures satisfy their specification in the given environment:

$$\forall f \in \mathit{dom}(\mathit{penv}), \arg, x.\ \triangleright_{(\mathit{penv},\mathit{lenv})}\ \{P_{(f,\arg,x)}\}\text{proccall}(\mathit{name}, \mathit{arg})\{Q_{(f,\arg,x)}\}$$

To prove that a procedure specification holds, it is sufficient to show that assuming that all procedures satisfy their specification, their bodies satisfy the specification. One does not need to show that possible recursions terminate, since Abstract Separation Logic just talks about partial correctness.

There is tool-support in the HOL-formalisation to handle procedure specifications. To prove a procedure specification, it is sufficient to prove the specifications of all procedure bodies, where a procedure call $\mathit{proccall}(\mathit{name}, \mathit{arg})$ has been replaced by $\mathit{qbla}[P_{(\mathit{name},\mathrm{arg},\cdot)}, Q_{(\mathit{name},\mathrm{arg},\cdot)}]$. This means that the resulting Hoare triples do not contain procedure calls any more. Therefore, the resulting Hoare triples do not depend on the procedure environment.

Making the Hoare triples independent from the lock-environment as well is not necessary, but often useful. The lock-operations $l.p$ and $\mathit{with}\ l\ \mathit{do}\ p$ can be eliminated by introducing $\mathit{annihilate}(\mathit{lenv}(l))$ and $\mathit{materialise}(\mathit{lenv}(l))$ at appropriate places in p. This moves the knowledge about lock invariants from the environment to the program itself, making the environment redundant.

Loops can be eliminated in a similar manner. Given a loop-invariant $I_{(\cdot)}$ such that for all x an intuitionistic predicate B is decided in I_x, a while-loop $\mathit{while}\ B\ \mathit{do}\ p$ can be abstracted by $\mathit{qbla}[I_{(\cdot)}, I_{(\cdot)} \wedge \neg_i B]$, if it can be proved that the body of the loop really satisfies the invariant.

After these preprocessing steps, one usually just needs to reason about programs consisting of local actions and conditional-execution, for which the presented inference rules are very useful.

3.5 Holfoot

The instantiation of the Abstract Separation Logic framework to Holfoot consists of two steps. First, the framework is instantiated to use a stack that maps variables to permissions and values. This instantiation is based on ideas from *Variables as Resource in Hoare Logic* [8]. The concrete type of the variables and values is not specified. Similarly, the stack is just a part of an abstract state. Nevertheless, this instantiation is sufficient to reason about pure expressions, assignments, local variables, etc. In a second step, this setting is instantiated to Holfoot.

Holfoot represents stack-variables with strings and uses natural numbers as values. Furthermore the abstract component of the state is instantiated to a heap from locations (represented by natural numbers without zero) and tags (represented as strings) to values (natural numbers). Using this concrete representation of a state, it is easy to define actions on these states. For example, the field-lookup action `v = e->t` is defined as

```
val holfoot_field_lookup_action_def = Define `
  (holfoot_field_lookup_action v e t) ((st,h):holfoot_state)) =
     if (~(var_res_sl___has_write_permission v st) \/
         IS_NONE (e st)) then NONE else
     let loc = (THE loc_opt) in (
       if (~(loc IN FDOM h) \/ (loc = 0)) then NONE else
       SOME {var_res_ext_state_var_update v ((h ' loc) t) (st,h)})`;
```

This action fails, if there is no write permission on the variable `v` or if the expression `e` fails to be evaluated in the current state (for example because a read permission on a variable it uses is missing). Otherwise, it checks whether the location pointed to by `e` is in the heap. If it is, the value of `v` is updated by the value found in the heap at that location indexed by tag `t`. Otherwise, i. e. if the location is not in the heap, the action fails.

Similarly to actions, it is straightforward to define predicates. For example, `e1 |-> L` is defined as:

```
val holfoot_ap_points_to_def = Define `
  holfoot_ap_points_to e1 L = \(st,h):holfoot_state.
    let loc_opt = (e1 st) in (IS_SOME (loc_opt) /\
    let (loc = THE loc_opt) in (~(loc = 0) /\  ((FDOM h)= {loc}) /\
    (FEVERY (\(tag,exp). IS_SOME (exp st) /\ (THE (exp st) = (h' loc) tag)) L))`;
```

This definition of `|->` is used to define predicates for single linked lists. The data content of these lists is represented by lists of natural numbers. Therefore HOL's list libraries can be used to reason about the data content.

Since actions and predicates are shallowly embedded, it is easy to extend Holfoot with new actions and predicates. Moreover, the automation has been designed with extensions in mind.

4 Conclusion and Future Work

The main contribution of this work is the formalisation of Abstract Separation Logic and demonstrating that Abstract Separation Logic is powerful and flexible enough to be used as a basis for separation logic tools. The formalisation of Abstract Separation Logic contains some minor extensions like the addition of procedures. However, it mainly follows the original definitions [4].

The Smallfoot case study demonstrates the potential of Abstract Separation Logic. However, it is interesting in its own right as well. The detected bug in Smallfoot shows that high-assurance implementations of even comparatively simple tools like Smallfoot are important. Moreover, Holfoot is one of the very few separation logic tools that can reason about the content of data-structures as well as the shape. Combining separation logic with reasoning about data-content is currently one of the main challenges for separation logic tools. As the example of parallel mergesort demonstrates, Holfoot can answer this challenge by combining the power of the interactive prover HOL with the automation separation logic provides.

In the future, I will try to improve the level of automation. Moreover, I plan to add a concept of arrays to Holfoot. This will put my claim that Holfoot is easily extensible to a test, since it requires adding new actions for allocating / deallocating blocks of the heap as well as adding predicates for arrays. However, the main purpose of adding arrays is reasoning about pointer arithmetic. It will be interesting to see, how HOL can help to verify algorithms that use pointer-arithmetic and how much automation is possible.

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.

Bibliography

[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] J. Berdine, C. Calcagno, and P.W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.

[3] S. Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007.

[4] C. Calcagno, P.W. O'Hearn, and H. 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] D. Distefano, P.W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer-Verlag, April 2006.

[6] N. Marti, R. Affeldt, and A. 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] P.W. O'Hearn, J.C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of 15th Annual Conference of the European Association for Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, September 2001.

[8] M. Parkinson, R. Bornat, and C. 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] J.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] H. Tuch, G. Klein, and M. 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] T. 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.