A HOL implementation of Smallfoot

Thomas Tuerk

27th January 2008

Separation Logic

- Separation logic is an extension of Hoare Logic
- successfully used to reason about programs using pointers
- allows local reasoning
- scales nicely
- there are some implementations
  - Smallfoot (Calcagno, Berdine, O'Hearn)
  - Slayer (MSR, B. Cook, J. Berdine et al.)
  - ...
- there are formalisation in theorem provers
  - *Concurrent C-Minor Project*, Coq (Appel et al.)
  - *Types, Bytes, and Separation Logic*, Isabelle/HOL (Tuch, Klein, Norrish)

Motivation

- there are a lot of slightly different separation logics
  - classically a state consists of stack + heap
  - but: how does the heap look like
  - read- / write-permissions for stack-variables ?
  - which predicates are supported?
- all tools / formalisations I know of are designed for one specific programming language
- in contrast, I would like to design a general framework
  - keep the core as abstract as possible
  - this should lead to simplicity
  - instantiate this core to different specific programming languages
- Main questions: what's the essence of separation logic? How to formalise it into a theorem prover?

Work done up to this point

- formalisation of *Abstract Separation Logic*
- first case study: a tool similar to Smallfoot
  - combines ideas from *Abstract Separation Logic*, *Variables as Resource* and Smallfoot
  - parser for Smallfoot example files
  - completely automatic verification
  - interactive proofs are possible as well
  - most features of Smallfoot are supported
  - data content is supported

Abstract Separation Logic

- Abstract Separation Logic is an abstract version
- introduced by Calcagno, O'Hearn and Yang in *Local Action and Abstract Separation Logic*
- abstraction helps to concentrate on the essential part
- embedding in a theorem prover becomes easier
- can be instantiated to different variants of separation logic
- therefore, it may be used as a basis for a separation logic framework in HOL

Introduction to Abstract Separation Logic

Separation Logic on Heaps

- heaps
- disjoint union of heaps $\uplus$
- h_1, h_2 have disjoint domains
- $h \models P_1 * P_2$ iff $\exists h_1, h_2.\ (h = h_1 \uplus h_2)\ \wedge\ h_1 \models P_1\ \wedge\ h_2 \models P_2$

Abstract Separation Logic

- abstract states
- abstract separation combinator $\circ$
- $s_1 \circ s_2$ is defined
- $s \models P_1 * P_2$ iff $\exists s_1, s_2.\ (s = s_1 \circ s_2) \wedge s_1 \models P_1\ \wedge\ s_2 \models P_2$

Separation Combinator

A **separation combinator** $\circ$ is a partially defined function such that:

- $\circ$ is **associative** $\forall x\, y\, z.\ (x \circ y) \circ z = x \circ (y \circ z)$
- $\circ$ is **commutative** $\forall x\, y.\ x \circ y = y \circ x$
- $\circ$ is **cancellative** $\forall x\, y\, z.\ (x \circ y = x \circ z) \Rightarrow y = z$
- forall elements there is a **neutral element** $\forall x. \exists u_x.\ u_x \circ x = x$

Hoare Triples and Actions

- consider partial correctness
- an action is a function from states to either a special failure state $\top$ or a set of states
- $\emptyset$ used to model actions that diverge
- $\{P\}$ action $\{Q\}$ iff forall states s such that $s \models P$ the action does not fail and $t \models Q$ forall $t \in$ action(s)

Local Actions / Frame Rule

Frame Rule

$$\frac{\{P\}\ \text{action}\ \{Q\}}{\{P * R\}\ \text{action}\ \{Q * R\}}$$

- frame rule is essential for separation logic
- it's important for local reasoning
- it does not hold for arbitrary actions
- actions that respect the frame rule are called **local**
- just local actions will be considered in the following

Programs

- c for every local action c
- `p ; q`
- `p + q`
- `p*`
- `p || q`
- `with l do p`
- `l.p`

Notice that `skip` and `assume c` for intuitionist conditions c are local actions.

Conditional execution and loops can be mimiced using non-determistic choice and `assume`.

Smallfoot

- "Smallfoot is an automatic verification tool which checks separation logic specifications of concurrent programs which manipulate dynamically-allocated recursive data structures."(Smallfoot documentation)
- developed by Cristiano Calcagno, Josh Berdine, Peter O'Hearn
- uses low-level imperative programming language that supports
  - pointers
  - local and global variables
  - dynamic memory allocation/deallocation
  - conditional execution, while-loops and recursive procedures
  - parallelism

Smallfoot II

`mergesort.sf`

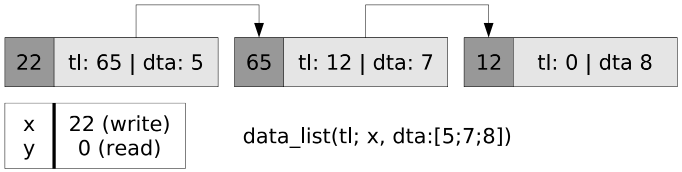
```
split(r;p) [list(p)] {
  local t1,t2;
  if(p == NULL) r = NULL;
  else {
    t1 = p->tl;
    if(t1 == NULL) {
      r = NULL;
    } else {
      t2 = t1->tl;
      split(r;t2);
      p->tl = t2;
      t1->tl = r;
      r = t1;
    }
  }
} [list(p) * list(r)]
```

```
merge(r;p,q)
   [list(p) * list(q)] {
   ...
} [list(r)]

mergesort(r;p) [list(p)] {
  local q,q1,p1;
  if(p == NULL) r = p;
  else {
    split(q;p);
    mergesort(q1;q);
    mergesort(p1;p);
    merge(r;p1,q1);
  }
} [list(r)]
```

Formalisation in HOL

- implemented as an instantiation of Abstract Separation Logic
- states are pairs of a heap and a stack
- the heap maps locations to named arrays
- the stack maps variables to value + permission
- stack uses ideas from Variables as Resource (Parkinson, Bornat, Calcagno)



Demo

Calculation of a frame - intuition

```
{pre}
   call_function fun_pre fun_post;
   prog
{post}
```

search `frame` such that

```
pre |= frame * fun_pre

{frame * fun_post}
   prog
{post}
```

- often the elimination of common parts of `pre` and `fun_pre` is the first step in the search
- keep these common parts in `context`

Calculation of a frame

```
SMALLFOOT_PROP_IMPLIES ...
  context pre fun_pre frame_prog_pred

means there is a frame such that

context * pre |= context * fun_pre * frame
frame_prog_pred frame holds
```

- additionally, one needs to take care of read / write-permissions

Calculation of a frame

```
val SMALLFOOT_PROP_IMPLIES_def = Define `
   SMALLFOOT_PROP_IMPLIES (strong_rest:bool) (wpb,rpb) wpb'
      sfb_context sfb_split sfb_imp sfb_restP =

~(sfb_restP = EMPTY) ==>
?sfb_rest. sfb_restP sfb_rest /\
   ((smallfoot_prop___COND (wpb,rpb)
      (BAG_UNION sfb_context (BAG_UNION sfb_split sfb_imp))) ==>

   (!s. smallfoot_prop___PROP (wpb,rpb) (BAG_UNION sfb_split sfb_context) s ==>
   (smallfoot_prop___COND (BAG_DIFF wpb wpb',
                           BAG_DIFF rpb wpb') sfb_rest /\
   smallfoot_prop___PROP (wpb,rpb)
      (BAG_UNION sfb_imp (BAG_UNION sfb_rest sfb_context)) s)))`
```

Consequence Conversions

- **conversions** are ML-functions that given a term `t` return a theorem `|- t = t_eq`.
- **consequence conversions** are ML-function that gievn a boolean term `t` return a theorem
  - `|- t_strong ==> t`,
  - `|- t = t_eq` or
  - `|- t ==> t_weak`.
- **directed consequence conversions** are consequence conversions with an additional direction argument to decide, whether to strengthen or weaken the input
- library `ConseqConv` contains useful consequence conversions and infrastructure for consequence conversions

Quantifier Instantiation Heuristics

- given a term `?x. P x` there are 3 reasons to instantiate `x` with a concrete value `i`:
  1. `P i`
  2. `!i'. ~(i = i') ==> ~(P i')`
  3. `!i'. P i' ==> P i`
- dual to these reasons there are three reasons for all-quantification
- `quantHeuristicsLib` is a library that supports instantiating quantifiers based on heuristics that come up with these guesses

Quantifier Instantiation Heuristics II

- a quantifier heuristic is an ML-function that given a term `P x` with a free variable `x` returns a list a **guesses** on how to instantiate `x`
- a guess consists of
  - the instantiation `i`
  - a list of free variables in `i` that should remain quantified
  - one of the 6 reasons or an *I-just-feel-like-it* reason
  - possibly a justification in form of a HOL-theorem
- if a justification is given, equivalence can be proved
- otherwise an implication is introduced

Quantifier Instantiation Heuristics III

- library knows about common boolean operators
- there is support for equations
- informations from type-base are used automatically
- all default heuristics come with a justifying theorem and are therefore safe
- user heuristics can be added very easily