Efficient verification of imperative programs using auto2

Bohua Zhan

Technical University of Munich

zhan@in.tum.de

April 16, 2018

Table of Contents

1. Introduction
2. Verification of functional programs
3. Separation logic
4. Case studies

Motivation

- Develop proof automation for verification of (sequential) imperative programs in the interactive theorem prover Isabelle.
- Goal: automate the mundane or technical parts of the proof, allowing the user to focus on the main ideas. Make the proof scripts easier to write, read, and maintain.
- Why interactive theorem proving?
  - Able to guide the computer through more complex reasoning.
  - Make use of large library of background mathematics.
  - Higher confidence in proofs due to small trusted kernel.

Auto2 prover

- Framework for proof automation implemented in Isabelle.
- Saturation based proof search: starting from list of assumptions, iteratively add derivable facts, until a contradiction is found.
- Easily extensible by the user through adding *proof steps* that represent reasoning rules and proof procedures.
- Designed to work effectively with case-analysis, induction, and equality reasoning.

Imperative HOL and its separation logic

- Imperative HOL in Isabelle: a simple language that represents imperative programs as monads.
- Separation logic for Imperative HOL, with several applications: work by Peter Lammich and Rene Meis.
- Ad-hoc refinement strategy: first verify a functional version of the program, then show that the imperative version refines the functional version using separation logic.
  - Independent from the refinement framework by Peter Lammich.
- Add proof steps to auto2 to support both stages of this process.

Table of Contents

1. Introduction
2. Verification of functional programs
3. Separation logic
4. Case studies

Verification of functional programs

- Directional use of lemmas.
- Normalization on natural numbers.
- Difference logic.

Directional use of lemmas

- In auto2, it is possible to register a lemma for use only in certain directions.
- Forward reasoning:

$$\mathit{sorted}\ (x\ \#\ \mathit{xs}) \Longrightarrow y \in \mathit{set}\ \mathit{xs} \Longrightarrow \underline{x \leq y}$$

- Backward reasoning:

$$\mathit{sorted}\ \mathit{xs} \Longrightarrow j < \mathit{length}\ \mathit{xs} \Longrightarrow \underline{i \leq j} \Longrightarrow \mathit{xs}\ !\ i \leq \mathit{xs}\ !\ j$$

- Rewrite rules:

$$i < \mathit{length}\ \mathit{xs} \Longrightarrow$$
$$\mathit{xs}[i := x]\ !\ j\ \underline{=\ (\mathit{if}\ i = j\ \mathit{then}\ x\ \mathit{else}\ \mathit{xs}\ !\ j)}$$

Normalization on natural numbers

- Subtraction is complicated because the intuitive normalization is possible only if the expression is *well-formed*. For example

```
(a::nat) - b + c = a + c - b
```

  does not hold in general in Isabelle (e.g. $a = 0, b = c = 1$).
- Solution: normalize expressions together with their *well-formedness conditions* (here $a \geq b$ for the term $a - b$).
- Example:

```
(j::nat) - k + i = j - 1 - (k - (i + 1))
```

  (under certain conditions. Used in the analysis of `rev_swap`).

Difference logic

- Work with inequalities of the form $a \leq b + n$ or $a + n \leq b$ on natural numbers, where n is a constant.
- Such inequalities can be encoded in a graph. They imply a contradiction if and only if the graph contains a negative cycle.
- We implement:
  - Apply transitivity to pairs $a \leq b + m$ and $b \leq c + n$.
  - Contradiction from $a + n \leq a$ with constant $n > 0$.
  - Use $a \leq b + n$ to justify $a \leq b + m$ for any $m \geq n$.
  - Use $a + n \leq b$ to justify $a \leq b + m$ for any m, and $a + m \leq b$ for any $m \leq n$.

Table of Contents

1 Introduction

2 Verification of functional programs

3 Separation logic

4 Case studies

Set up for separation logic

- Forward reasoning with Hoare triples.
- Matching of inductively-defined assertions.

Reasoning using Hoare triples

- Reason about programs in the forward direction.
- General form of a Hoare triple:

  $<p_1 \ * \cdots * \ p_m \ * \uparrow(a_1) \ * \cdots \ * \uparrow(a_k)>$
  $\quad cmd$
  $<\lambda r.\ \exists_A \vec{x}.\ q_1 \ * \cdots \ * \ q_n \ * \uparrow(b_1) \ * \cdots \ * \uparrow(b_l)>$

- Steps for applying a single Hoare triple:
  - Match *cmd* with current command.
  - Frame inference with the spatial part $p_1 * \cdots * p_m$ of the precondition.
  - Create case-analysis to check the pure part $a_1, \ldots, a_k$ of the precondition.
  - Apply the Hoare triple to produce assertion on the next heap. The pure parts $b_1, \ldots, b_l$ are added as facts.

Matching of inductively-defined assertions

- Example for binary trees:

```
btree Tip p = ↑(p = None)
btree (tree.Node lt k v rt) (Some p) =
  (∃_A lp rp. p ↦_r Node lp k v rp * btree lt lp * btree rt rp)
btree (tree.Node lt k v rt) None = false
```

- For assertion on the current heap, always expand as much as possible.
- Improve matching function so that

$$p \mapsto_r \texttt{Node lp k v rp} * \texttt{btree lt lp} * \texttt{btree rt rp}$$

will match the pattern `btree ?t p`.

Table of Contents

1. Introduction
2. Verification of functional programs
3. Separation logic
4. Case studies

Case studies

- Union-find[1].
- Red-black tree.
- Interval tree.
- Rectangle intersection.
- Indexed priority queue[2].
- Dijkstra's algorithm[3].

[1]Statement of definitions and lemmas from corresponding example in the AFP entry "A Separation Logic Framework for Imperative HOL" by Peter Lammich and Rene Meis.

[2]Also formalized in the AFP entry "The Imperative Refinement Framework" by Peter Lammich.

[3]Also formalized in the AFP entry "Dijkstra's Shortest Path Algorithm" by Benedikt Nordhoff and Peter Lammich.

Case studies: statistics

	#Imp	#Def	#Thm	#Step	Ratio	#LOC
Union-find	49	7	26	42	0.86	244
Red-black tree	270	27	83	173	0.64	998
Interval tree	84	17	50	83	0.99	520
Rectangle intersection	33	18	31	111	3.36	417
Indexed priority queue	83	10	53	84	1.01	477
Dijkstra's algorithm	44	19	62	150	3.41	549

#Imp: number of lines of imperative code.
#Def: number of definitions.
#Thm: number of lemmas and theorems.
#Step: number of "steps" in the proof.
Ratio: ratio between #Step and #Imp.
#LOC: total number of lines of code in the theories.

Interval tree: search

```
fun search :: "interval_tree ⇒ nat interval ⇒ bool" where
  "search Tip x = False"
| "search (Node l y m r) x =
   (if is_overlap (int y) x then True
    else if l ≠ Tip ∧ tmax l ≥ low x then search l x
    else search r x)"
setup {* fold add_rewrite_rule @{thms search.simps} *}

lemma search_correct [rewrite]:
  "is_interval_tree t ⟹ is_interval x ⟹ search t x ⟷ has_overlap (tree_set t) x"
@proof
  @induct t @with
    @subgoal "t = Node l y m r"
      @let "t = Node l y m r"
      @case "is_overlap (int y) x"
      @case "l ≠ Tip ∧ tmax l ≥ low x" @with
        @obtain "p∈tree_set l" where "high (int p) = tmax l"
        @case "is_overlap (int p) x"
      @end
      @case "l = Tip"
    @endgoal
  @end
@qed
```

Interval tree: imperative search

```
partial_function (heap) search_impl :: "nat interval ⇒ int_tree ⇒ bool Heap" where
  "search_impl x b = (case b of
     None ⇒ return False
   | Some p ⇒ do {
      t ← !p;
      (if is_overlap (int (val t)) x then return True
       else case lsub t of
           None ⇒ do { b ← search_impl x (rsub t); return b }
         | Some lp ⇒ do {
            lt ← !lp;
            if tmax lt ≥ low x then
              do { b ← search_impl x (lsub t); return b }
            else
              do { b ← search_impl x (rsub t); return b }})})"

lemma search_impl_correct [hoare_triple]:
  "<int_tree t b>
    search_impl x b
   <λr. int_tree t b * ↑(r ⟷ search t x)>"
@proof @induct t arbitrary b @with
  @subgoal "t = interval_tree.Node l v m r"
    @case "is_overlap (int v) x"
    @case "l ≠ Tip ∧ interval_tree.tmax l ≥ low x"
  @endgoal @end
@qed
```

Sweep-line algorithm for rectangle intersection

```
lemma has_overlap_at_k_equiv2 [resolve]:
  "is_rect_list rects ⟹ ops = all_ops rects ⟹ has_rect_overlap rects ⟹
   ∃k<length ops. has_overlap_at_k rects k"
@proof
  @obtain i j where "i < length rects" "j < length rects" "i ≠ j"
                    "is_rect_overlap (rects ! i) (rects ! j)"
  @have "is_rect_overlap (rects ! j) (rects ! i)"
  @obtain i1 where "i1 < length ops" "ops ! i1 = ins_op rects i"
  @obtain j1 where "j1 < length ops" "ops ! j1 = ins_op rects j"
  @obtain i2 where "i2 < length ops" "ops ! i2 = del_op rects i"
  @obtain j2 where "j2 < length ops" "ops ! j2 = del_op rects j"
  @case "ins_op rects i < ins_op rects j" @with
    @have "i1 < j1"
    @have "j1 < i2" @with @have "ops ! j1 < ops ! i2" @end
    @have "is_overlap (int (IdxInterval (xint (rects ! i)) i)) (xint (rects ! j))"
    @have "has_overlap_at_k rects j1"
  @end
  @case "ins_op rects j < ins_op rects i" @with
    @have "j1 < i1"
    @have "i1 < j2" @with @have "ops ! i1 < ops ! j2" @end
    @have "is_overlap (int (IdxInterval (xint (rects ! j)) j)) (xint (rects ! i))"
    @have "has_overlap_at_k rects i1"
  @end
@qed
```

Dijkstra's algorithm

```
lemma derive_dist [backward2]:
  "known_dists G V ⟹
   m ∈ verts G - V ⟹
   ∀i∈verts G - V. dist_on G 0 i V ≥ dist_on G 0 m V ⟹
   has_dist G 0 m ∧ dist G 0 m = dist_on G 0 m V"
@proof
  @obtain p where "is_shortest_path_on G 0 m p V"
  @have "is_shortest_path G 0 m p" @with
    @have "p ∈ path_set G 0 m"
    @have "∀p'∈path_set G 0 m. path_weight G p' ≥ path_weight G p" @with
      @obtain p1 p2 where "joinable G p1 p2" "p' = path_join G p1 p2"
                          "int_pts p1 ⊆ V" "hd p2 ∉ V"
      @let "x = last p1"
      @have "dist_on G 0 x V ≥ dist_on G 0 m V"
      @have "p1 ∈ path_set_on G 0 x V"
      @have "path_weight G p1 ≥ dist_on G 0 x V"
      @have "path_weight G p' ≥ dist_on G 0 m V + path_weight G p2"
    @end
  @end
@qed
```

Future work

- Support for `while` and `for` loops.
- Decision procedures for linear arithmetic and for arrays.
- Extend framework to verify running time (CADE 2018).
- https://github.com/bzhan/auto2