Automatic Induction Proofs of Data-Structures in Imperative Programs

Duc-Hiep Chu, Joxan Jaffar, and Minh-Thai Trinh

National University of Singapore (NUS)

June 17, 2015



- 2 The Specification Language
- 3 Intuition behind Our Induction Rules
- 4 Some Examples



- Verifying functional correctness of dynamic data structures
- Specifications are formalized using a logic of heaps and separation
 A core feature is the use of user-defined recursive predicates
- The Problem: entailment checking, where both LHS and RHS involve such predicates

- Performs systematic *folding* and *unfolding* steps of the recursive rules, and succeeds when we produce a formula which is *obviously provable*:
 - no recursive predicate in RHS of the proof obligation, and a direct proof can be achieved by consulting some generic SMT solver;
 - no special consideration is needed on any occurrence of a predicate appearing in the formula, i.e., *formula abstraction* can be applied.
- E.g., HIP/SLEEK (Chin et al. [2012]), DRYAD (Qiu et al. [2013])

$$\text{Consider } \widehat{\mathsf{ls}}(x,y) \stackrel{\textit{def}}{=} x {=} y \land \text{ emp } | x {\neq} y \land (x {\mapsto} t) * \widehat{\mathsf{ls}}(t,y)$$

Pre:
$$\widehat{ls}(x,y)$$

assume(x != y)
z = x.next
Post: $\widehat{ls}(z,y)$

Unfold the precondition $\hat{ls}(x,y)$

- Case 1: holds because (x = y) and assume(x != y) implies *false*
- Case 2: holds by matching z with t

- Recursion Divergence: when the "recursion" in the recursive rules is structurally *dissimilar* to the program code
- Generalization of Predicate: when the predicate describing a loop invariant or a function is used later to prove a weaker property

(occurs often in practice, especially in iterative programs)

Recursion Divergence

• When the "recursion" in the recursive rules is structurally *dissimilar* to the program code

Pre:
$$\widehat{ls}(x,y) * (y \mapsto_{-})$$

z = y.next
Post: $\widehat{ls}(x,z)$

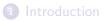
Fundamentally, it is about relating two definitions of a list segment: (recurse rightwards, and recurse leftwards)

$$\widehat{ls}(x,y) \stackrel{def}{=} x=y \land emp | x\neq y \land (x\mapsto t) * \widehat{ls}(t,y) ls(x,y) \stackrel{def}{=} x=y \land emp | x\neq y \land (t\mapsto y) * ls(x,t)$$

(sometimes inevitable, e.g., queue implementation using list segment)

- When the predicate describing a loop invariant or a function is used later to prove a weaker property
 - sorted_list(x, len, min) \models list(x, len)
 - $ls(x, y) * list(y) \models list(x)$

- Traditional works on automated induction generally require variables of inductive type (so that the notions of base case and induction step are well-defined)
- Our predicates are (user-)defined over pointer variables, which are not inductive



Intuition behind Our Induction Rules

4 Some Examples

5 Future Work

- We use the language proposed by Duck *et al.* [2013], a logic with the features of *explicit heaps* and *a separation operator*
 - It facilitates symbolic execution and therefore VC generation
 - Our induction method is not confined to this language

 E.g. the below defines a skeleton list (we inherit the CLP semantics) list(x, L) :- x = null, L ≏ Ø. list(x, L) :- x ≠ null, L ≏ (x→t) * L₁, list(t, L₁).

(note that * applies to terms, and not predicates as in traditional Separation Logic)



Intuition behind Our Induction Rules

4 Some Examples

5 Future Work

$$(\text{CUT}) \ \frac{\mathcal{L}_1 \models \mathcal{A} \qquad \mathcal{L}_2 \land \mathcal{A} \models \mathcal{R}}{\mathcal{L}_1 \land \mathcal{L}_2 \models \mathcal{R}}$$

- Trivial from the deduction point of view (top to bottom)
- For proof derivation (bottom to top), obtaining an appropriate A is tantamount to a *magic step*
 - In manual proofs, we perform this magic step all the time
 - Automating this step is extremely hard

$$(I-1) \frac{ \begin{array}{c} \mathcal{L}_1 \models \mathcal{A} \end{array}}{\mathcal{L}_1 \land \mathcal{L}_2 \models \mathcal{R}} \\ \mathcal{L}_1 \land \mathcal{L}_2 \models \mathcal{R} \\ \vdots \end{array}$$

- $\left\lfloor \mathcal{L}_1 \models \mathcal{A} \right\rfloor$ is "the same" as some obligation previously encountered in the proof path (indicated by \cdots above), which acts as an *induction hypothesis*, thus $\left\lceil \mathcal{L}_1 \models \mathcal{A} \right\rceil$ will be discharged immediately
- In other words, the proof path gives us a systematic way to discover the magic formula A

Induction Rule 2

(I-2)
$$\frac{\mathcal{L}_1 \models \mathcal{A} \qquad \boxed{\mathcal{L}_2 \land \mathcal{A} \models \mathcal{R}}}{\mathcal{L}_1 \land \mathcal{L}_2 \models \mathcal{R}}$$

- $\mathcal{L}_2 \wedge \mathcal{A} \models \mathcal{R}$ is "the same" as some obligation previously encountered in the proof path (indicated by \cdots above), which acts as an induction hypothesis, thus $\mathcal{L}_2 \wedge \mathcal{A} \models \mathcal{R}$ will be discharged immediately
- $\bullet\,$ Again, the proof path gives us a systematic way to discover the magic formula ${\cal A}$

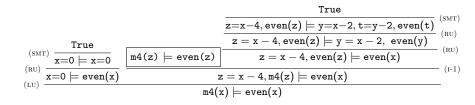
- Our automated induction rules allow for
 - a systematic method to discover \mathcal{A} (in the cut-rule)
 - application of induction to discharge a proof obligation, thus we only need to proceed with the remaining obligation
- A technical challenge is to ensure induction applications do not lead to *circular* (i.e., wrong) reasoning

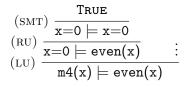
Example (simplified by ignoring heaps)

even(x) :-
$$x = 0$$
.
even(x) :- $y = x - 2$, even(y).
m4(x) :- $x = 0$.
m4(x) :- $z = x - 4$, m4(z).

 $m4(x) \models even(x)$

• Unfold-and-Match will not work: there always remains obligation with predicate m4 in the LHS and predicate even in the RHS



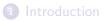


Example: Induction Works

$$(I-1) \begin{array}{c} (SMT) \\ (RU) \\ (RU) \\ (RU) \\ (RU) \\ (RU) \end{array} \underbrace{ \begin{array}{c} (SMT) \\ (RU) \\ (RU) \end{array}}_{(RU)} \underbrace{ \begin{array}{c} (SMT) \\ (Z=x-4, even(z) \models y=x-2, even(t) \\ (Z=x-4, even(z) \models y=x-2, even(y) \\ (Z=x-4, even(z) \models even(x) \\ (U) \\ (LU) \end{array} \underbrace{ \begin{array}{c} (SMT) \\ (RU) \\ (RU) \\ (Z=x-4, even(z) \models even(x) \\ (Z=x-4, even(z) \models even(x) \\ (RU) \\ (RU$$

Applying induction rule 1, we discover even(z) as a candidate for A.

This step allows us to "flip" the predicate even(z) into the LHS so that subsequently Unfold-and-Match can work.



Intuition behind Our Induction Rules





Some Examples

• Proving commonly-used "lemmas" (or "axioms"); many existing systems simply accept them as facts from the users

$$\begin{array}{l} \mathsf{sorted_list}(x,\mathit{min}) \models \mathsf{list}(x) \\ \mathsf{sorted_list}_1(x,\mathit{len},\mathit{min}) \models \mathsf{list}_1(x,\mathit{len}) \\ \mathsf{sorted_list}_1(x,\mathit{len},\mathit{min}) \models \mathsf{sorted_list}(x,\mathit{min}) \\ \mathsf{sorted_lis}_1(x,\mathit{len},\mathit{min}) \models \mathsf{sorted_list}(y,\mathit{min}_2) \land \mathit{max} \leq \mathit{min}_2 \models \mathsf{sorted_list}(x,\mathit{min}) \\ \widehat{\mathsf{ls}}_1(x,y,\mathit{len}_1) * \widehat{\mathsf{ls}}_1(y,z,\mathit{len}_2) \models \widehat{\mathsf{ls}}_1(x,z,\mathit{len}_1+\mathit{len}_2) \\ \mathsf{ls}_1(x,y,\mathit{len}_1) * \mathsf{list}_1(y,\mathit{len}_2) \models \mathsf{list}_1(x,\mathit{len}_1+\mathit{len}_2) \\ \widehat{\mathsf{ls}}_1(x,\mathit{last},\mathit{len}) * (\mathit{last} \mapsto \mathit{new}) \models \widehat{\mathsf{ls}}_1(x,\mathit{new},\mathit{len}+1) \\ \mathsf{avl}(x,\mathit{hgt},\mathit{min},\mathit{max},\mathit{balance}) \models \mathsf{bstree}(x,\mathit{height}) \\ \cdots \end{array}$$

(running time ranges from 0.2 - 1 second per benchmark)

Some Examples

- Eliminate the usage of lemmas: it indeed runs faster
 - at a node, we only look at the available induction hypotheses (0 3)
 - other systems look at all the "lemmas" (or "axioms")

Table: Verification of Open-Source Libraries

Program	Function	Time per Function
glib/gslist.c Singly Linked-List	<pre>find, position, index, nth,last,length,append, insert_at_pos,merge_sort, remove,insert_sorted_list</pre>	<15
glib/glist.c Doubly Linked-List	nth, position, find, index, last, length	<1s
OpenBSD/ queue.h Queue	<pre>simpleq_remove_after, simpleq_insert_tail, simpleq_insert_after</pre>	<15
ExpressOS/ cachePage.c	lookup_prev, add_cachepage	<1 <i>s</i>
linux/mmap.c	insert_vm_struct	<1 <i>s</i>



Intuition behind Our Induction Rules

4 Some Examples



- Improve the robustness
 - e.g. works for $A \models B$, but might fail if we strengthen A (or weaken B)
 - having too strong antecedent (or too weak consequent) is an obstacle to the usage of induction

Questions?

- J. Brotherston, D. Distefano, and R. L. Petersen. Automated cyclic entailment proofs in separation logic. In *CADE*, 2011.
- W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. In SCP, pages 1006–1036, 2012.
- G. Duck, J. Jaffar, and N. Koh. A constraint solver for heaps with separation. In *CP, LNCS 8124*, 2013.
- X. Qiu, P. Garg, A. Stefanescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In *PLDI*, pages 231–242, 2013.

- HIP/SLEEK (Chin *et al.* [2012]): a very comprehensive system supporting specifications written in SL
- DRYAD (Qiu *et al.* [2013]): more deterministic algorithm where unfolding is guided by program footprint
- "Cyclic Proof" (Brotherston *et al.* [2011]): a theory to ensure sound termination of cyclic proof paths. This is similar to deciding if an induction application is valid.