# 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

1. Introduction
2. The Specification Language
3. Intuition behind Our Induction Rules
4. Some Examples
5. Future Work

# Problem Setting

- 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

# The State-of-the-Art: Unfold-and-Match

- 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])

# Example: Unfold-and-Match

Consider $\widehat{\text{ls}}(x,y) \stackrel{def}{=} x{=}y \land \textbf{emp} \ \mid x{\neq}y \land (\texttt{x}{\mapsto}\texttt{t}) * \widehat{\text{ls}}(t,y)$

```
Pre:  ls^(x,y)
   assume(x != y)
   z = x.next
Post:  ls^(z,y)
```

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

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

# Shortcomings

1. Recursion Divergence: when the “recursion” in the recursive rules is structurally *dissimilar* to the program code

2. 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

$$\texttt{Pre:}\quad \widehat{\mathsf{ls}}(\mathtt{x},\mathtt{y}) * (\mathtt{y}\mapsto \_)$$

```
z = y.next
```

$$\texttt{Post:}\quad \widehat{\mathsf{ls}}(\mathtt{x},\mathtt{z})$$

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

$$\widehat{\mathsf{ls}}(\mathtt{x},\mathtt{y}) \stackrel{def}{=} \mathtt{x}=\mathtt{y} \wedge \mathbf{emp} \;\mid\; \mathtt{x}\neq\mathtt{y} \wedge (\mathtt{x}\mapsto\mathtt{t}) * \widehat{\mathsf{ls}}(\mathtt{t},\mathtt{y})$$

$$\mathsf{ls}(\mathtt{x},\mathtt{y}) \stackrel{def}{=} \mathtt{x}=\mathtt{y} \wedge \mathbf{emp} \;\mid\; \mathtt{x}\neq\mathtt{y} \wedge (\mathtt{t}\mapsto\mathtt{y}) * \mathsf{ls}(\mathtt{x},\mathtt{t})$$

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

# Generalization of Predicate:

- 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)

# What is Needed: INDUCTION

- 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

1. Introduction
2. The Specification Language
3. Intuition behind Our Induction Rules
4. Some Examples
5. Future Work

# The Specification Language

- 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)

$$\begin{array}{lll} \mathrm{list}(x, L) & \text{:-} & x = null,\ L \doteq \emptyset. \\ \mathrm{list}(x, L) & \text{:-} & x \neq null,\ L \doteq (x \mapsto t) * L_1,\ \mathrm{list}(t, L_1). \end{array}$$

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

1 Introduction

2 The Specification Language

3 Intuition behind Our Induction Rules

4 Some Examples

5 Future Work

# General Cut-Rule

$$(\textsc{cut})\ \frac{\mathcal{L}_1 \models \mathcal{A} \qquad \mathcal{L}_2 \wedge \mathcal{A} \models \mathcal{R}}{\mathcal{L}_1 \wedge \mathcal{L}_2 \models \mathcal{R}}$$

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

# Induction Rule 1

$$(\textsc{i-1})\ \frac{\boxed{\mathcal{L}_1 \models \mathcal{A}} \qquad \mathcal{L}_2 \wedge \mathcal{A} \models \mathcal{R}}{\mathcal{L}_1 \wedge \mathcal{L}_2 \models \mathcal{R}}$$

$$\vdots$$

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

# Induction Rule 2

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

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

# Summary

- 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)

$$
\begin{aligned}
&\texttt{even}(x) \; \texttt{:-} \quad x = 0. \\
&\texttt{even}(x) \; \texttt{:-} \quad y = x - 2,\ \texttt{even(y)}. \\
\\
&\texttt{m4}(x) \; \texttt{:-} \quad x = 0. \\
&\texttt{m4}(x) \; \texttt{:-} \quad z = x - 4,\ \texttt{m4(z)}.
\end{aligned}
$$

$$\texttt{m4(x)} \models \texttt{even(x)}$$

- 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

$$
\dfrac{
  \dfrac{
    \dfrac{\texttt{True}}{\texttt{x=0} \models \texttt{x=0}}\;(\textsc{smt})
  }{\texttt{x=0} \models \texttt{even(x)}}\;(\textsc{ru})
  \qquad
  \dfrac{
    \boxed{\texttt{m4(z)} \models \texttt{even(z)}}
    \quad
    \dfrac{
      \dfrac{
        \dfrac{\texttt{True}}{\texttt{z=x}-\texttt{4}, \texttt{even(z)} \models \texttt{y=x}-\texttt{2}, \texttt{t=y}-\texttt{2}, \texttt{even(t)}}\;(\textsc{smt})
      }{\texttt{z} = \texttt{x} - \texttt{4}, \texttt{even(z)} \models \texttt{y} = \texttt{x} - \texttt{2},\ \texttt{even(y)}}\;(\textsc{ru})
    }{\texttt{z} = \texttt{x} - \texttt{4}, \texttt{even(z)} \models \texttt{even(x)}}\;(\textsc{ru})
  }{\texttt{z} = \texttt{x} - \texttt{4}, \texttt{m4(z)} \models \texttt{even(x)}}\;(\textsc{i-1})
}{\texttt{m4(x)} \models \texttt{even(x)}}\;(\textsc{lu})
$$

# Example: Induction Works

$$
\dfrac{\text{(LU)}\ \dfrac{\text{(RU)}\ \dfrac{\text{(SMT)}\ \dfrac{\textsc{True}}{\texttt{x=0} \models \texttt{x=0}}}{\texttt{x=0} \models \texttt{even(x)}} \qquad \vdots}{\texttt{m4(x)} \models \texttt{even(x)}}}{}
$$

# Example: Induction Works

$$
\text{(LU)}\ \dfrac{\text{(I-1)}\ \dfrac{\boxed{\mathtt{m4(z)} \models \mathtt{even(z)}} \quad \text{(RU)}\ \dfrac{\text{(RU)}\ \dfrac{\text{(SMT)}\ \dfrac{\mathtt{True}}{\mathtt{z}{=}\mathtt{x}{-}4, \mathtt{even(z)} \models \mathtt{y}{=}\mathtt{x}{-}2, \mathtt{t}{=}\mathtt{y}{-}2, \mathtt{even(t)}}}{\mathtt{z} = \mathtt{x} - 4, \mathtt{even(z)} \models \mathtt{y} = \mathtt{x} - 2,\ \mathtt{even(y)}}}{\mathtt{z} = \mathtt{x} - 4, \mathtt{even(z)} \models \mathtt{even(x)}}}{\mathtt{z} = \mathtt{x} - 4, \mathtt{m4(z)} \models \mathtt{even(x)}}}{\mathtt{m4(x)} \models \mathtt{even(x)}}
$$

- Applying induction rule 1, we discover `even(z)` as a candidate for $\mathcal{A}$.

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

1 Introduction

2 The Specification Language

3 Intuition behind Our Induction Rules

4 Some Examples

5 Future Work

# Some Examples

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

$$
\begin{array}{l}
\mathrm{sorted\_list}(x, min) \models \mathrm{list}(x) \\
\mathrm{sorted\_list}_1(x, len, min) \models \mathrm{list}_1(x, len) \\
\mathrm{sorted\_list}_1(x, len, min) \models \mathrm{sorted\_list}(x, min) \\
\mathrm{sorted\_ls}(x, y, min, max) * \mathrm{sorted\_list}(y, min_2) \wedge max \leq min_2 \models \mathrm{sorted\_list}(x, min) \\
\widehat{\mathrm{ls}}_1(x, y, len_1) * \widehat{\mathrm{ls}}_1(y, z, len_2) \models \widehat{\mathrm{ls}}_1(x, z, len_1 + len_2) \\
\mathrm{ls}_1(x, y, len_1) * \mathrm{list}_1(y, len_2) \models \mathrm{list}_1(x, len_1 + len_2) \\
\widehat{\mathrm{ls}}_1(x, last, len) * (last \mapsto new) \models \widehat{\mathrm{ls}}_1(x, new, len + 1) \\
\mathrm{avl}(x, hgt, min, max, balance) \models \mathrm{bstree}(x, hgt, min, max) \\
\mathrm{bstree}(x, height, min, max) \models \mathrm{bintree}(x, height) \\
\qquad \ldots
\end{array}
$$

(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` | `find, position, index, nth,last,length,append, insert_at_pos,merge_sort, remove,insert_sorted_list` | $<1s$ |
| **glib/glist.c** `Doubly Linked-List` | `nth, position, find, index, last, length` | $<1s$ |
| **OpenBSD/ queue.h** `Queue` | `simpleq_remove_after, simpleq_insert_tail, simpleq_insert_after` | $<1s$ |
| **ExpressOS/ cachePage.c** | `lookup_prev, add_cachepage` | $<1s$ |
| **linux/mmap.c** | `insert_vm_struct` | $<1s$ |

1. Introduction
2. The Specification Language
3. Intuition behind Our Induction Rules
4. Some Examples
5. Future Work

# What Next?

- 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.

# Related Work

- 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.