Second-Order Constraints in Dynamic Invariant Inference

Kaituo Li (University of Massachusetts Amherst)
Christoph Reichenbach (Goethe University Frankfurt)
Yannis Smaragdakis (University of Athens)
Michal Young (University of Oregon)

21 August 2013

Invariants on Program Behaviour

- Invariants are formal documentation:
  - Pre-/postconditions, class invariants
  - Effect specifications
  - Dependency specifications

    . . .
- Formal semantics: amenable to formal methods, testing

Invariants on Program Behaviour

- Invariants are formal documentation:
  - Pre-/postconditions, class invariants
  - Effect specifications
  - Dependency specifications

  ...
- Formal semantics: amenable to formal methods, testing

Example:

ArrayStack.peek() **requires** preconditions:

```
 storageArray != null
 topOfStackIndex >= 0
 topOfStackIndex < storageArray.length
...
```

Limitations of current invariant-based approaches

- Invariants sometimes hard to write
  Partly addressed by invariant inference
  (e.g., Daikon, DIDUCE, DySy, Heureka, ...)
- Can bury important information
  E.g., some Daikon-inferred specifications for simple methods have dozens of axioms
- Can involve redundancy
- Can be inconsistent with higher-level knowledge

ArrayStack.peek() **requires**:

```
array != null
topOfStack >= 0
topOfStack < array.length
...
```

ArrayStack.pop() **requires**:

```
array != null
topOfStack >= 0
topOfStack < array.length-1
...
```

Limitations of current invariant-based approaches

- Invariants sometimes hard to write
  Partly addressed by invariant inference
  (e.g., Daikon, DIDUCE, DySy, Heureka, ...)
- Can bury important information
  E.g., some Daikon-inferred specifications for simple methods have dozens of axioms
- Can involve redundancy
- Can be inconsistent with higher-level knowledge

ArrayStack.peek() **requires**:

```
array != null
topOfStack >= 0
topOfStack < array.length
...
```

ArrayStack.pop() **requires**:

```
array != null
topOfStack >= 0
topOfStack < array.length-1
...
```

Second-Order Constraints

Our approach: *constraints between sets of invariants*

Second-Order Constraints

Our approach: *constraints between sets of invariants*

Examples:

- if we meet all preconditions of ‘peek’ we also meet all preconditions of ‘pop’:
  SUBDOMAIN(peek, pop)

Second-Order Constraints

Our approach: *constraints between sets of invariants*

Examples:

- if we meet all preconditions of ‘peek’ we also meet all preconditions of ‘pop’:
  SUBDOMAIN(peek, pop)

ArrayStack.peek() **requires**:

```
array != null
topOfStack >= 0
topOfStack < array.length
...
```

SUBDOMAIN(peek, pop)

Second-Order Constraints

Our approach: *constraints between sets of invariants*

Examples:

- if we meet all preconditions of 'peek' we also meet all preconditions of 'pop':
  SUBDOMAIN(peek, pop)
- 'pop' is safe to call after 'push':
  CANFOLLOW(push, pop)

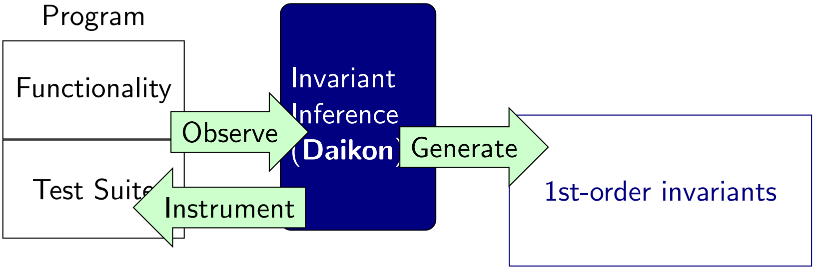
Second-Order Constraints

Our approach: *constraints between sets of invariants*

Examples:

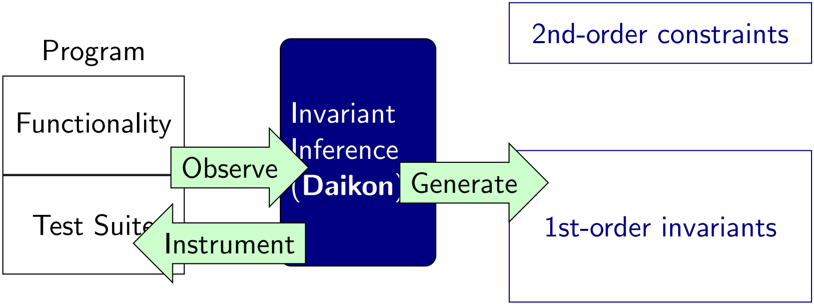
- if we meet all preconditions of ‘peek’ we also meet all preconditions of ‘pop’:
  SUBDOMAIN(peek, pop)
- ‘pop’ is safe to call after ‘push’:
  CANFOLLOW(push, pop)
- ‘invert3x3Matrix’ works like ‘invertMatrix’, but may have stronger preconditions:
  CONCORD(invertMatrix, invert3x3Matrix)

Applying second-order constraints to Daikon



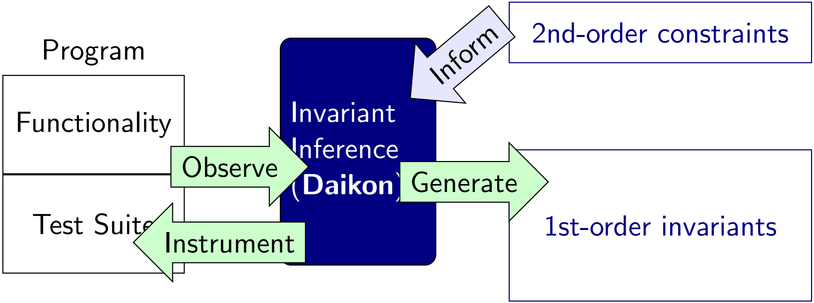
Applying second-order constraints to Daikon

- Hand-written documentation



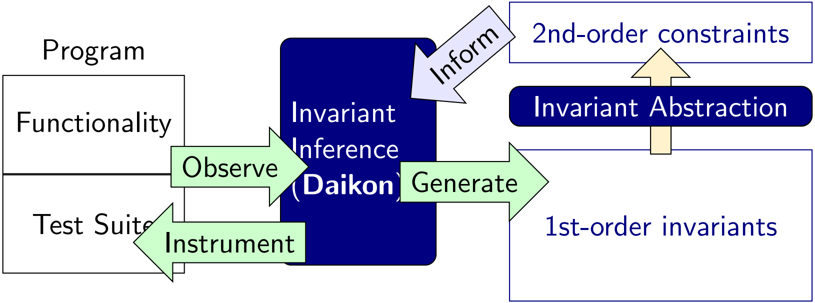
Applying second-order constraints to Daikon

- Hand-written documentation
- Two software tools:
  - A Hints for first-order invariant inference



Applying second-order constraints to Daikon

- Hand-written documentation
- Two software tools:
  - A Hints for first-order invariant inference
  - B *Automatically inferred* higher-level documentation



A catalogue of second-order constraints

- SUBDOMAIN(peek, pop)
  'pop' is applicable whenever 'peek' is

A catalogue of second-order constraints

- SUBDOMAIN(peek, pop)
  'pop' is applicable whenever 'peek' is
- SUBRANGE(intersect, clear)
  'clear' ensures at least as many invariants as 'intersect'

A catalogue of second-order constraints

- SUBDOMAIN(peek, pop)
  'pop' is applicable whenever 'peek' is
- SUBRANGE(intersect, clear)
  'clear' ensures at least as many invariants as 'intersect'
- CANFOLLOW(connect, send)
  'connect' provides the requirements for 'send' is

A catalogue of second-order constraints

- SUBDOMAIN(peek, pop)
  'pop' is applicable whenever 'peek' is
- SUBRANGE(intersect, clear)
  'clear' ensures at least as many invariants as 'intersect'
- CANFOLLOW(connect, send)
  'connect' provides the requirements for 'send' is
- FOLLOWS(push, pop)
  Anyone calling 'pop' may call 'push' first

A catalogue of second-order constraints

- SUBDOMAIN(peek, pop)
  'pop' is applicable whenever 'peek' is
- SUBRANGE(intersect, clear)
  'clear' ensures at least as many invariants as 'intersect'
- CANFOLLOW(connect, send)
  'connect' provides the requirements for 'send' is
- FOLLOWS(push, pop)
  Anyone calling 'pop' may call 'push' first
- CONCORD(computeGeneralFFT, computeCoprimeFFT)
  'computeCoprimeFFT' has behavioural subtype of 'computeGeneralFFT'

A catalogue of second-order constraints

- SUBDOMAIN(peek, pop)
  ‘pop’ is applicable whenever ‘peek’ is
- SUBRANGE(intersect, clear)
  ‘clear’ ensures at least as many invariants as ‘intersect’
- CANFOLLOW(connect, send)
  ‘connect’ provides the requirements for ‘send’ is
- FOLLOWS(push, pop)
  Anyone calling ‘pop’ may call ‘push’ first
- CONCORD(computeGeneralFFT, computeCoprimeFFT)
  ‘computeCoprimeFFT’ has behavioural subtype of ‘computeGeneralFFT’

Other useful properties are conceivable

Logical structure of our constraint catalogue

requires	**ensures**
pre (A)	post (A)
pre (B)	post (B)

Logical structure of our constraint catalogue

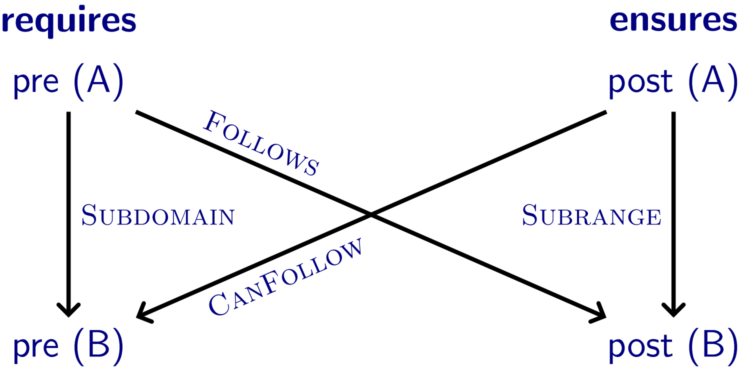
requires	**ensures**
pre (A)	post (A)
$\downarrow$ Subdomain	
pre (B)	post (B)

Logical structure of our constraint catalogue

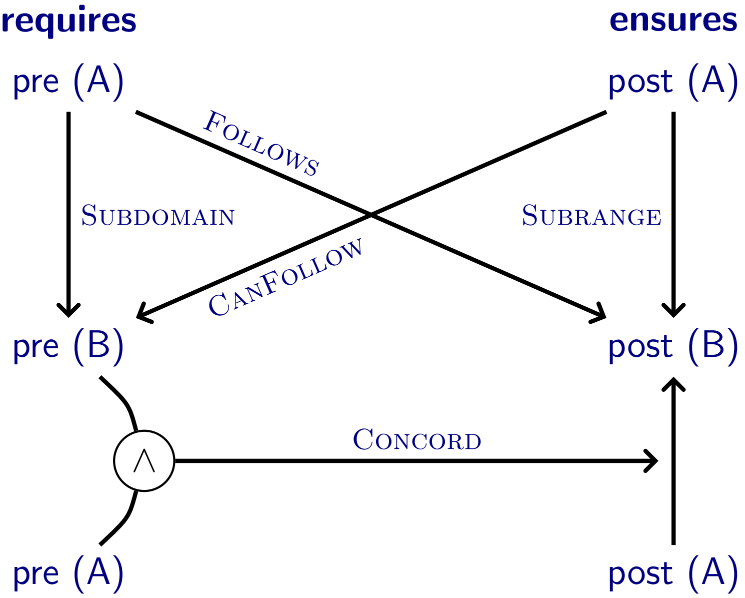
requires | **ensures**

$$\begin{tikzcd} \text{pre }(A) \arrow[d, "\textsc{Subdomain}"] & & \text{post }(A) \arrow[d, "\textsc{Subrange}"'] \\ \text{pre }(B) & & \text{post }(B) \end{tikzcd}$$

Logical structure of our constraint catalogue



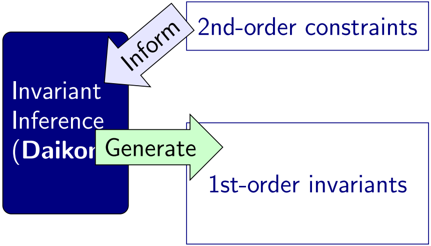
Logical structure of our constraint catalogue



Inference with Second-Order Constraints

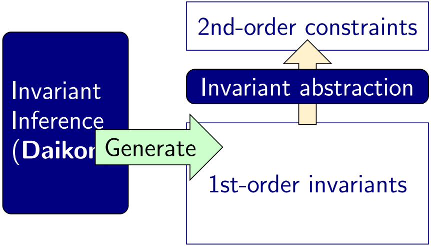
Ⓐ Refined First-Order Inference

- User provides hand-written second-order constraints



Ⓑ Second-Order Inference

- System infers second-order constraints



Refined First-Order Inference

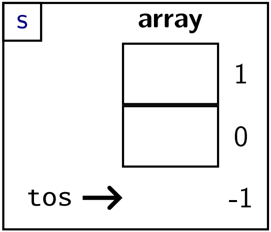
```
ArrayStack s = new ArrayStack();
s.push("bar");
s.push("foo");
assert s.peek() == "foo";
assert s.pop() == "foo";
assert s.peek() == "bar";
```

Daikon

`ArrayStack.peek()` **requires**

`ArrayStack.pop()` **requires**

Refined First-Order Inference

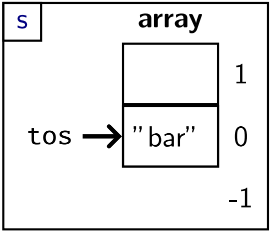



```
⇒ ArrayStack s = new ArrayStack();
   s.push("bar");
   s.push("foo");
   assert s.peek() == "foo";
   assert s.pop() == "foo";
   assert s.peek() == "bar";
```

Daikon

`ArrayStack.peek()` **requires**   `ArrayStack.pop()` **requires**

Refined First-Order Inference

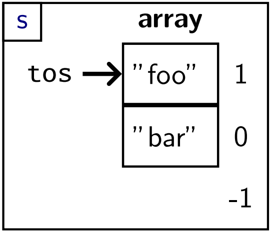



```
ArrayStack s = new ArrayStack();
⇒ s.push("bar");
s.push("foo");
assert s.peek() == "foo";
assert s.pop() == "foo";
assert s.peek() == "bar";
```

Daikon

`ArrayStack.peek()` **requires** `ArrayStack.pop()` **requires**

Refined First-Order Inference

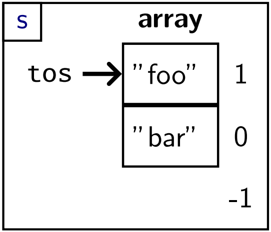



```
ArrayStack s = new ArrayStack();
s.push("bar");
⇒s.push("foo");
assert s.peek() == "foo";
assert s.pop() == "foo";
assert s.peek() == "bar";
```

Daikon

`ArrayStack.peek()` **requires** `ArrayStack.pop()` **requires**

Refined First-Order Inference



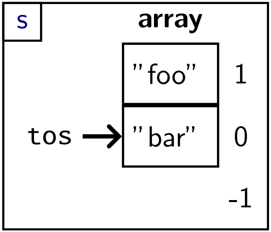

```
ArrayStack s = new ArrayStack();
s.push("bar");
s.push("foo");
⇒ assert s.peek() == "foo";
assert s.pop() == "foo";
assert s.peek() == "bar";
```

Daikon

`ArrayStack.peek()` **requires**
`array != null`
`tos = 1`

`ArrayStack.pop()` **requires**

Refined First-Order Inference



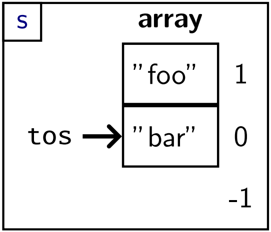

```
ArrayStack s = new ArrayStack();
s.push("bar");
s.push("foo");
assert s.peek() == "foo";
⇒ assert s.pop() == "foo";
assert s.peek() == "bar";
```

Daikon

```
ArrayStack.peek() requires
 array != null
 tos = 1
```

```
ArrayStack.pop() requires
 array != null
 tos = 1
```

Refined First-Order Inference



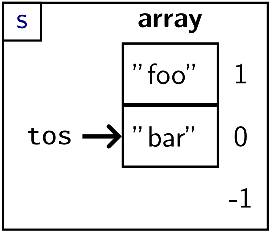

```
ArrayStack s = new ArrayStack();
s.push("bar");
s.push("foo");
assert s.peek() == "foo";
assert s.pop() == "foo";
⇒ assert s.peek() == "bar";
```

Daikon

```
ArrayStack.peek() requires
 array != null
 tos = 1...?
```

```
ArrayStack.pop() requires
 array != null
 tos = 1
```

Refined First-Order Inference




```
ArrayStack s = new ArrayStack();
s.push("bar");
s.push("foo");
assert s.peek() == "foo";
assert s.pop() == "foo";
⇒ assert s.peek() == "bar";
```

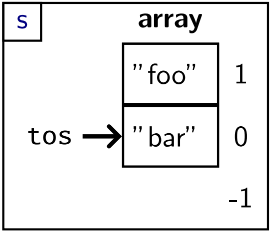
Daikon

```
ArrayStack.peek() requires
 array != null
 tos >= 0
```

~~tos = 1~~

```
ArrayStack.pop() requires
 array != null
 tos = 1
```

Refined First-Order Inference



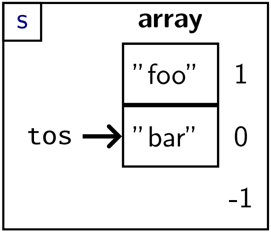

```
ArrayStack s = new ArrayStack();
s.push("bar");
s.push("foo");
assert s.peek() == "foo";
assert s.pop() == "foo";
⇒ assert s.peek() == "bar";
```

SUBDOMAIN(peek, pop)

```
ArrayStack.peek() requires
  array != null
```
~~tos = 1~~
```
  tos >= 0
```

```
ArrayStack.pop() requires
  array != null
  tos = 1
```

Refined First-Order Inference



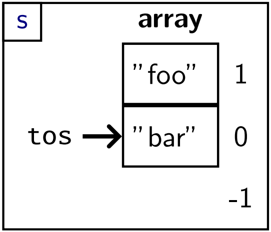

```
ArrayStack s = new ArrayStack();
s.push("bar");
s.push("foo");
assert s.peek() == "foo";
assert s.pop() == "foo";
assert s.peek() == "bar";
```

SUBDOMAIN(peek, pop)

ArrayStack.peek() **requires**
array != null
~~tos = 1~~
tos >= 0

ArrayStack.pop() **requires**
array != null
~~tos = 1~~
tos >= 0

Refined First-Order Inference




```
ArrayStack s = new ArrayStack();
s.push("bar");
s.push("foo");
assert s.peek() == "foo";
assert s.pop() == "foo";
assert s.peek() == "bar";
```

Daikon

```
ArrayStack.peek() requires        ArrayStack.pop() requires
  ar
tos
tos
```

Each second-order constraint propagates information across Daikon

Second-Order Inference

- Hypothesise all possible 2nd-order constraints per class
- Compute fraction of overlap (using automated theorem prover): determine a *confidence* metric on 2nd-order constraint

Example hypothesis: SUBRANGE(ArrayStack.pop, ArrayStack.peek)

peek() **ensures**
  return.class == String
  this has only one value

pop() **ensures**
  return.class == String
  this has only one value
  return has only one value

Second-Order Inference

- Hypothesise all possible 2nd-order constraints per class
- Compute fraction of overlap (using automated theorem prover): determine a *confidence* metric on 2nd-order constraint

Example hypothesis: SUBRANGE(ArrayStack.pop, ArrayStack.peek)

```
peek() ensures
  return.class == String
  this has only one value
```

```
pop() ensures
  return.class == String
  this has only one value
  return has only one value
```

$post(peek) \Rightarrow$ `return.class == String`?

$$confidence = \frac{1}{1}$$

Second-Order Inference

- Hypothesise all possible 2nd-order constraints per class
- Compute fraction of overlap (using automated theorem prover): determine a *confidence* metric on 2nd-order constraint

Example hypothesis: SUBRANGE(ArrayStack.pop, ArrayStack.peek)

```
peek() ensures
 return.class == String
 this has only one value
```

```
pop() ensures
 return.class == String
 this has only one value
 return has only one value
```

$post(peek) \Rightarrow$ `this has only one value?`

$$confidence = \frac{1+1}{1+1}$$

Second-Order Inference

- Hypothesise all possible 2nd-order constraints per class
- Compute fraction of overlap (using automated theorem prover): determine a *confidence* metric on 2nd-order constraint

Example hypothesis: SUBRANGE(ArrayStack.pop, ArrayStack.peek)

`peek()` **ensures**
`return.class == String`
this has only one value

`pop()` **ensures**
`return.class == String`
this has only one value
return has only one value

post(*peek*) ⇒ *return* has only one value?

$$confidence = \frac{1+1+0}{1+1+1}$$

Second-Order Inference

- Hypothesise all possible 2nd-order constraints per class
- Compute fraction of overlap (using automated theorem prover): determine a *confidence* metric on 2nd-order constraint
- Incorporate per-invariant **Daikon Confidence** metrics

Example hypothesis: SUBRANGE(ArrayStack.pop, ArrayStack.peek)

DC `peek()` **ensures**

.9	`return.class == String`
.8	*this* has only one value

DC `pop()` **ensures**

.7	`return.class == String`
.6	*this* has only one value
.5	*return* has only one value

$$\mathit{confidence} =$$

Second-Order Inference

- Hypothesise all possible 2nd-order constraints per class
- Compute fraction of overlap (using automated theorem prover): determine a *confidence* metric on 2nd-order constraint
- Incorporate per-invariant **Daikon Confidence** metrics

Example hypothesis: SUBRANGE(ArrayStack.pop, ArrayStack.peek)

DC `peek()` **ensures**

.9	`return.class == String`
.8	*`this`* `has only one value`

DC `pop()` **ensures**

.7	`return.class == String`
.6	*`this`* `has only one value`
.5	*`return`* `has only one value`

$$\mathit{confidence} = \frac{.9\times.8(\qquad\qquad)}{}$$

Second-Order Inference

- Hypothesise all possible 2nd-order constraints per class
- Compute fraction of overlap (using automated theorem prover): determine a *confidence* metric on 2nd-order constraint
- Incorporate per-invariant **Daikon Confidence** metrics

Example hypothesis: SUBRANGE(ArrayStack.pop, ArrayStack.peek)

DC `peek()` **ensures**

.9	`return.class == String`
.8	*this* has only one value

DC `pop()` **ensures**

.7	`return.class == String`
.6	*this* has only one value
.5	*return* has only one value

$post(peek) \Rightarrow$ `return.class == String`?

$$confidence = \frac{.9 \times .8 (.7 \qquad\qquad)}{1}$$

Second-Order Inference

- Hypothesise all possible 2nd-order constraints per class
- Compute fraction of overlap (using automated theorem prover): determine a *confidence* metric on 2nd-order constraint
- Incorporate per-invariant **Daikon Confidence** metrics

Example hypothesis: SUBRANGE(ArrayStack.pop, ArrayStack.peek)

DC `peek()` **ensures**

.9	`return.class == String`
.8	*this* has only one value

DC `pop()` **ensures**

.7	`return.class == String`
.6	*this* has only one value
.5	*return* has only one value

$post(peek) \Rightarrow$ *this* has only one value?

$$confidence = \frac{.9 \times .8 (.7 + .6 \quad)}{1+1}$$

Second-Order Inference

- Hypothesise all possible 2nd-order constraints per class
- Compute fraction of overlap (using automated theorem prover): determine a *confidence* metric on 2nd-order constraint
- Incorporate per-invariant **Daikon Confidence** metrics

Example hypothesis: SUBRANGE(ArrayStack.pop, ArrayStack.peek)

DC `peek()` **ensures**

.9 `return.class == String`

.8 *this* has only one value

DC `pop()` **ensures**

.7 `return.class == String`

.6 *this* has only one value

.5 *return* has only one value

post(*peek*) $\Rightarrow$ *return* has only one value?

$$\textit{confidence} = \frac{.9 \times .8(.7 + .6 + 0)}{1 + 1 + 1}$$

Evaluation (overview)

A Refined First-Order Inference

- Evaluated changes to inferred first-order invariants
- Second-order constraints written by hand for:
  - Daikon's StackAr
  - 18 Apache Commons Collections classes
  - 7 AspectJ classes

B Second-Order Inference

- Evaluated generated second-order constraints
- For all hand-written second-order constraints
- For random classes:
  - 2 Apache Commons Collections
  - 2 AspectJ
- Confidence threshold: 0.75

Evaluation (overview)

Ⓐ Refined First-Order Inference

- Evaluated changes to inferred first-order invariants
- Second-order constraints written by hand for:
  - Daikon's StackAr
  - 18 Apache Commons Collections classes
  - 7 AspectJ classes

Evaluation: Refined Invariant Inference (1)

StackAr: Daikon's stack-based array implementation

- int topOfStack: top-of-stack location
- int[] theArray: stack representation
- pop(): remove top element
- top(): peek at top element value
- topAndPop(): remove top element and return it

Evaluation: Refined Invariant Inference (1)

StackAr: Daikon's stack-based array implementation

- int topOfStack: top-of-stack location
- int[] theArray: stack representation
- pop(): remove top element
- top(): peek at top element value
- topAndPop(): remove top element and return it

$$\boxed{\begin{array}{l}\textsc{Subdomain}(\textsc{pop},\ \textsc{top})\\ \textsc{Subdomain}(\textsc{top},\ \textsc{topAndPop})\\ \textsc{Subdomain}(\textsc{topAndPop},\ \textsc{pop})\end{array}}$$

Evaluation: Refined Invariant Inference (1)

StackAr: Daikon's stack-based array implementation

- int topOfStack: top-of-stack location
- int[] theArray: stack representation
- pop(): remove top element
- top(): peek at top element value
- topAndPop(): remove top element and return it

SUBDOMAIN(POP, TOP)
SUBDOMAIN(TOP, TOPANDPOP)
SUBDOMAIN(TOPANDPOP, POP)

- Removed 5 false invariants:
  - `this has only one value`
  - `theArray has only one value`
  - `theArray.length == 100`
  - `theArray[topOfStack] != null`
  - `topOfStack < theArray.length-1`

Evaluation: Refined Invariant Inference (1)

StackAr: Daikon's stack-based array implementation

- int topOfStack: top-of-stack location
- int[] theArray: stack representation
- pop(): remove top element
- top(): peek at top element value
- topAndPop(): remove top element and return it

$$\textsc{Subdomain}(\textsc{pop}, \textsc{top})$$
$$\textsc{Subdomain}(\textsc{top}, \textsc{topAndPop})$$
$$\textsc{Subdomain}(\textsc{topAndPop}, \textsc{pop})$$

- Removed 5 false invariants:
- Added 2 new invariants:
  - `this.topOfStack >= -1`
  - `DEFAULT_CAPACITY != theArray.length-1`

Evaluation: Refined Invariant Inference (1)

StackAr: Daikon's stack-based array implementation

- int topOfStack: top-of-stack location
- int[] theArray: stack representation
- pop(): remove top element
- top(): peek at top element value
- topAndPop(): remove top element and return it

$$\boxed{\begin{array}{l}\textsc{Subdomain}(\textsc{pop},\ \textsc{top})\\ \textsc{Subdomain}(\textsc{top},\ \textsc{topAndPop})\\ \textsc{Subdomain}(\textsc{topAndPop},\ \textsc{pop})\end{array}}$$

- Removed 5 false invariants:
- Added 2 new invariants:
- Refined 1 overly-specific invariants:

```
this.topOfStack < size(this.theArray[])-1
```

$$\Downarrow$$

```
this.topOfStack <= size(this.theArray[])-1
```

Evaluation: Refined Invariant Inference (2)

Experiment	2nd-order Constraints Written	1st-order Invariants Added	Removed	Refined
StackAr #1	3	2	5	1
StackAr #2	3	3	4	0
Apache Commons Collections	26	25 + 1	35	5
AspectJ	26	12 + 1	12	3

- Most changes positive
- Only two incorrect invariants introduced
  - Both due to nonmonotonicity in the underlying first-order invariant inference mechanism

Evaluation: Refined Invariant Inference (2)

Experiment	2nd-order Constraints Written	1st-order Invariants Added	Removed	Refined
StackAr #1	3	2	5	1
StackAr #2	3	3	4	0
Apache Commons Collections	26	25 + 1	35	5
AspectJ	26	12 + 1	12	3

- Most changes positive
- Only two incorrect invariants introduced
  - Both due to nonmonotonicity in the underlying first-order invariant inference mechanism

Overwhelmingly positive effects

Evaluation (overview)

Ⓑ Second-Order Inference

- Evaluated generated second-order constraints
- For all hand-written second-order constraints
- For random classes:
  - 2 Apache Commons Collections
  - 2 AspectJ
- Confidence threshold: 0.75

Evaluation: Second-Order Inference

1. Confirming our manual annotations:
   Of our 64 original manual annotations:
   - 37 we inferred
   - 27 we did *not* infer, of which:
     - 12 we had wrongly annotated
     - 7 lacked any supporting data samples
     - 6 we rejected due to 'noise' first-order invariants
     - 2 were **Concord**

Evaluation: Second-Order Inference

1. Confirming our manual annotations:
   Of our 64 original manual annotations:
   - 37 we inferred
   - 27 we did *not* infer, of which:
     - 12 we had wrongly annotated
     - 7 lacked any supporting data samples
     - 6 we rejected due to 'noise' first-order invariants
     - 2 were **Concord**
2. Finding new second-order constraints:

	inferred	**incorrect**
Apache Commons: AbstractMapBag	2	0
Apache Commons: SingletonMap	806	0
AspectJ Reflection:	30	0
AspectJ LstBuildConfigManager:	112	5

Evaluation: Second-Order Inference

1. Confirming our manual annotations:
   Of our 64 original manual annotations:
   - 37 we inferred
   - 27 we did *not* infer, of which:
     - 12 we had wrongly annotated
     - 7 lacked any supporting data samples
     - 6 we rejected due to ‘noise’ first-order invariants
     - 2 were **Concord**
2. Finding new second-order constraints:

IMMUTABLE class: suggests even higher-order invariants!

Apache Commons: AbstractMapBag	2	0
Apache Commons: SingletonMap	806	0
AspectJ Reflection:	30	0
AspectJ LstBuildConfigManager:	112	5

Evaluation: Second-Order Inference

1. Confirming our manual annotations:
   Of our 64 original manual annotations:
   - 37 we inferred
   - 27 we did *not* infer, of which:
     - 12 we had wrongly annotated
     - 7 lacked any supporting data samples
     - 6 we rejected due to ‘noise’ first-order invariants
     - 2 were **Concord**
2. Finding new second-order constraints:

	inferred	**incorrect**
Apache Commons: AbstractMapBag	2	0
Aspects Reflection:	[illegible]	0
AspectJ LstBuildConfigManager:	112	5

$\boxed{\text{Incomplete unit tests} \Rightarrow \text{Poor Daikon invariants}}$

Conclusions

Second-order constraints:

- Permit high level of discourse about program properties
- Refine the quality of detected first-order invariants
- Can be detected automatically
- Are easy to use and powerful