Simplifying Reasoning about Objects with Tako

Gregory Kulczycki and Jyotindra Vasudeo
Virginia Tech

Immediate objective

Introduce students to "heavyweight" formal reasoning without scaring them (too much)

- familiar syntax (Java-like)
- value semantics
- variable-based state space

The Tako Language

- Similar to Java, but...
  - Avoids common sources of aliasing
    - Automatically initializes variables
    - Uses swapping (:=:) instead of assignment
    - Methods have in-out parameter passing

Tako

```
public class BddStack {
    private final Integer MAX;
    private Object[ ] contents;
    private Integer top;
    public BddStack(Integer n) {
        MAX := n;
        contents := new Object[MAX];
    }
    public void push(Object x) {
        assert depth( ) < MAX;
        contents[top] :=: x;
        top++;
    }
    public void pop(Object x) {
        assert depth( ) > 0;
        top--;
        x :=: contents[top];
    }
    public Integer depth( ) {
        result := top;
    }
}
```

Java

```
public class BddStack {
    private final int MAX;
    private Object[ ] contents;
    private int top;
    public BddStack(int n) {
        MAX = n;
        contents = new Object[MAX];
    }
    public void push(Object x) {
        assert depth( ) < MAX;
        contents[top] = x;
        top++;
    }
    public Object pop( ) {
        assert depth( ) > 0;
        top--;
        return contents[top];
    }
    public int depth( ) {
        return top;
    }
}
```

Tako	Java

Tako

```
public class BddStack {
    private final Integer MAX;
    private Object[ ] contents;
    private Integer top;
    public BddStack(Integer n) {
        MAX := n;
        contents := new Object[MAX];
    }
    public void push(Object x) {
        assert depth( ) < MAX;
        contents[top] :=: x;
        top++;
    }
    public void pop(Object x) {
        assert depth( ) > 0;
        top--;
        x :=: contents[top];
    }
    public Integer depth( ) {
        result := top;
    }
}
```

Java

```
public class BddStack {
    private final int MAX;
    private Object[ ] contents;
    private int top;
    public BddStack(int n) {
        MAX = n;
        contents = new Object[MAX];
    }
    public void push(Object x) {
        assert depth( ) < MAX;
        contents[top] = x;
        top++;
    }
    public Object pop( ) {
        assert depth( ) > 0;
        top--;
        return contents[top];
    }
    public int depth( ) {
        return top;
    }
}
```

Tako Initialization

- By convention, Tako programmers should always write a default constructor, which is applied to all variables as soon as they are declared.
- Integer top; ≡ Integer top := **new** Integer( );
- IStack s; ≡ IStack s := **null**;   // interface
- But use BddStack s := **new** BddStack(20);

Tako

```
public class BddStack {
    private final Integer MAX;
    private Object[ ] contents;
    private Integer top;
    public BddStack(Integer n) {
        MAX := n;
        contents := new Object[MAX];
    }
    public void push(Object x) {
        assert depth( ) < MAX;
        contents[top] :=: x;
        top++;
    }
    public void pop(Object x) {
        assert depth( ) > 0;
        top--;
        x :=: contents[top];
    }
    public Integer depth( ) {
        result := top;
    }
}
```

Java

```
public class BddStack {
    private final int MAX;
    private Object[ ] contents;
    private int top;
    public BddStack(int n) {
        MAX = n;
        contents = new Object[MAX];
    }
    public void push(Object x) {
        assert depth( ) < MAX;
        contents[top] = x;
        top++;
    }
    public Object pop( ) {
        assert depth( ) > 0;
        top--;
        return contents[top];
    }
    public int depth( ) {
        return top;
    }
}
```

Tako

```
public class BddStack {
    private final Integer MAX;
    private Object[ ] contents;
    private Integer top;
    public BddStack(Integer n) {
        MAX := n;
        contents := new Object[MAX];
    }
    public void push(Object x) {
        assert depth( ) < MAX;
        contents[top] :=: x;
        top++;
    }
    public void pop(Object x) {
        assert depth( ) > 0;
        top--;
        x :=: contents[top];
    }
    public Integer depth( ) {
        result := top;
    }
}
```

Java

```
public class BddStack {
    private final int MAX;
    private Object[ ] contents;
    private int top;
    public BddStack(int n) {
        MAX = n;
        contents = new Object[MAX];
    }
    public void push(Object x) {
        assert depth( ) < MAX;
        contents[top] = x;
        top++;
    }
    public Object pop( ) {
        assert depth( ) > 0;
        top--;
        return contents[top];
    }
    public int depth( ) {
        return top;
    }
}
```

Function Assignment

- Has the form variable := my.expression( );
- If the compiler finds:
  MAX := n;
  it will translate it as:
  MAX := n.replica( );
  where replica( ) is a programmer defined method expected to perform a deep copy

Tako

```
public class BddStack {
    private final Integer MAX;
    private Object[ ] contents;
    private Integer top;
    public BddStack(Integer n) {
        MAX := n;
        contents := new Object[MAX];
    }
    public void push(Object x) {
        assert depth( ) < MAX;
        contents[top] :=: x;
        top++;
    }
    public void pop(Object x) {
        assert depth( ) > 0;
        top--;
        x :=: contents[top];
    }
    public Integer depth( ) {
        result := top;
    }
}
```

Java

```
public class BddStack {
    private final int MAX;
    private Object[ ] contents;
    private int top;
    public BddStack(int n) {
        MAX = n;
        contents = new Object[MAX];
    }
    public void push(Object x) {
        assert depth( ) < MAX;
        contents[top] = x;
        top++;
    }
    public Object pop( ) {
        assert depth( ) > 0;
        top--;
        return contents[top];
    }
    public int depth( ) {
        return top;
    }
}
```

Tako

```
public class BddStack {
    private final Integer MAX;
    private Object[ ] contents;
    private Integer top;
    public BddStack(Integer n) {
        MAX := n;
        contents := new Object[MAX];
    }
    public void push(Object x) {
        assert depth( ) < MAX;
        contents[top] :=: x;
        top++;
    }
    public void pop(Object x) {
        assert depth( ) > 0;
        top--;
        x :=: contents[top];
    }
    public Integer depth( ) {
        result := top;
    }
}
```

Java

```
public class BddStack {
    private final int MAX;
    private Object[ ] contents;
    private int top;
    public BddStack(int n) {
        MAX = n;
        contents = new Object[MAX];
    }
    public void push(Object x) {
        assert depth( ) < MAX;
        contents[top] = x;
        top++;
    }
    public Object pop( ) {
        assert depth( ) > 0;
        top--;
        return contents[top];
    }
    public int depth( ) {
        return top;
    }
}
```

Swapping

- Compiler can implement it by swapping references
- Preserves unique references, so the programmer can reason about it as if objects are swapped

Tako

```
public class BddStack {
    private final Integer MAX;
    private Object[ ] contents;
    private Integer top;
    public BddStack(Integer n) {
        MAX := n;
        contents := new Object[MAX];
    }
    public void push(Object x) {
        assert depth( ) < MAX;
        contents[top] :=: x;
        top++;
    }
    public void pop(Object x) {
        assert depth( ) > 0;
        top--;
        x :=: contents[top];
    }
    public Integer depth( ) {
        result := top;
    }
}
```

Java

```
public class BddStack {
    private final int MAX;
    private Object[ ] contents;
    private int top;
    public BddStack(int n) {
        MAX = n;
        contents = new Object[MAX];
    }
    public void push(Object x) {
        assert depth( ) < MAX;
        contents[top] = x;
        top++;
    }
    public Object pop( ) {
        assert depth( ) > 0;
        top--;
        return contents[top];
    }
    public int depth( ) {
        return top;
    }
}
```

Tako

```
public class BddStack {
    private final Integer MAX;
    private Object[ ] contents;
    private Integer top;
    public BddStack(Integer n) {
        MAX := n;
        contents := new Object[MAX];
    }
    public void push(Object x) {
        assert depth( ) < MAX;
        contents[top] :=: x;
        top++;
    }
    public void pop(Object x) {
        assert depth( ) > 0;
        top--;
        x :=: contents[top];
    }
    public Integer depth( ) {
        result := top;
    }
}
```

Java

```
public class BddStack {
    private final int MAX;
    private Object[ ] contents;
    private int top;
    public BddStack(int n) {
        MAX = n;
        contents = new Object[MAX];
    }
    public void push(Object x) {
        assert depth( ) < MAX;
        contents[top] = x;
        top++;
    }
    public Object pop( ) {
        assert depth( ) > 0;
        top--;
        return contents[top];
    }
    public int depth( ) {
        return top;
    }
}
```

Tako Parameter Passing

- Unlike Java, variable values in Tako can always be updated through parameters
- Similar to:
  - C++ pass-by-reference
  - C# ref parameters
  - Ada in-out parameter passing
  - Pascal value-result parameter passing

Functions vs. Procedures

- By convention, Tako functions (non-void methods) do not change the program state
  (for all variables v, v = #v)
- If you want to change a variable's value, use a procedure (void method)
- For example, use **public void** pop(Item x); instead of **public** Item pop( );

Tako

```
public class BddStack {
    private final Integer MAX;
    private Object[ ] contents;
    private Integer top;
    public BddStack(Integer n) {
        MAX := n;
        contents := new Object[MAX];
    }
    public void push(Object x) {
        assert depth( ) < MAX;
        contents[top] :=: x;
        top++;
    }
    public void pop(Object x) {
        assert depth( ) > 0;
        top--;
        x :=: contents[top];
    }
    public Integer depth( ) {
        result := top;
    }
}
```

Java

```
public class BddStack {
    private final int MAX;
    private Object[ ] contents;
    private int top;
    public BddStack(int n) {
        MAX = n;
        contents = new Object[MAX];
    }
    public void push(Object x) {
        assert depth( ) < MAX;
        contents[top] = x;
        top++;
    }
    public Object pop( ) {
        assert depth( ) > 0;
        top--;
        return contents[top];
    }
    public int depth( ) {
        return top;
    }
}
```

Tako

```
public class BddStack {
    private final Integer MAX;
    private Object[ ] contents;
    private Integer top;
    public BddStack(Integer n) {
        MAX := n;
        contents := new Object[MAX];
    }
    public void push(Object x) {
        assert depth( ) < MAX;
        contents[top] :=: x;
        top++;
    }
    public void pop(Object x) {
        assert depth( ) > 0;
        top--;
        x :=: contents[top];
    }
    public Integer depth( ) {
        result := top;
    }
}
```

Java

```
public class BddStack {
    private final int MAX;
    private Object[ ] contents;
    private int top;
    public BddStack(int n) {
        MAX = n;
        contents = new Object[MAX];
    }
    public void push(Object x) {
        assert depth( ) < MAX;
        contents[top] = x;
        top++;
    }
    public Object pop( ) {
        assert depth( ) > 0;
        top--;
        return contents[top];
    }
    public int depth( ) {
        return top;
    }
}
```

```
public Integer depth( ) {
    result := top;
}
```

translates to

```
public Integer depth( ) {
    Integer result = new Integer( );
    result = top.replica( );
    return result;
}
```

Clean semantics

Variable-based property

The state space is simply the abstract object values of the defined variables

Frame property

During a method call, the only variable values that may change are:

1. arguments to the call, and
2. globals defined in the *affects* clause.

```
import spec.MathString;

public interface Stack {
    model MathString;

    initialization ensures
        this = EMPTY_STRING;

    public void push(Object x);
        ensures this = <#x> o #this;

    public void pop(Object x);
        requires |this| > 0;
        ensures #this = <x> o this;

    public Integer depth( );
        ensures this = #this and result = |this|;
}
```

```
import spec.MathString;

public interface Stack {
    model MathString;

    initialization ensures
        this = EMPTY_STRING;

    public void push(Object x);
        ensures this = <#x> o #this;

    public void pop(Object x);
        requires |this| > 0;
        ensures #this = <x> o this;

    public Integer depth( );
        ensures this = #this and result = |this|;
}
```

```
import spec.MathString;

public interface Stack {
    model MathString;

    initialization ensures
        this = EMPTY_STRING;

    public void push(Object x);
        ensures this = <#x> o #this;

    public void pop(Object x);
        requires |this| > 0;
        ensures #this = <x> o this;

    public Integer depth( );
        ensures this = #this and result = |this|;
}
```

public void push(Object x) guarantees that . . .

The current stack value is equal to a string containing the old x value concatenated with the old stack value

```
ensures this = <#x> o #this;
```

public void push(Object x) guarantees that . . .

The current stack value is equal to a string containing the old x value concatenated with the old stack value

```
ensures this = <#x> o #this;
```

public void push(Object x) guarantees that . . .

The current stack value is equal to a string containing the old x value concatenated with the old stack value

ensures this = <#x> o #**this**;

public void push(Object x) guarantees that . . .

The current stack value is equal to a string containing the old x value concatenated with the old stack value

ensures this = <#x> o #**this**;

public void push(Object x) guarantees that . . .

The current stack value is equal to a string containing the old x value concatenated with the old stack value

ensures this = <#x> o #**this**;

public void push(Object x) guarantees that . . .

The current stack value is equal to a string containing the old x value concatenated with the old stack value

ensures this = <#x> o **#this**;

public void push(Object x) guarantees that . . .

The current stack value is equal to a string containing the old x value concatenated with the old stack value

```
ensures this = <#x> o #this;
```

public void push(Object x) guarantees that . . .

The current stack value is equal to a string containing the old x value concatenated with the old stack value and the current x value is a valid but unspecified value

ensures this = <#x> o #**this**;

and x = ??

public void push(Object x) guarantees that . . .

The current stack value is equal to a string containing the old x value concatenated with the old stack value and the current x value is a valid but unspecified value and no other variables in the state space change

ensures this = <#x> o #**this**;

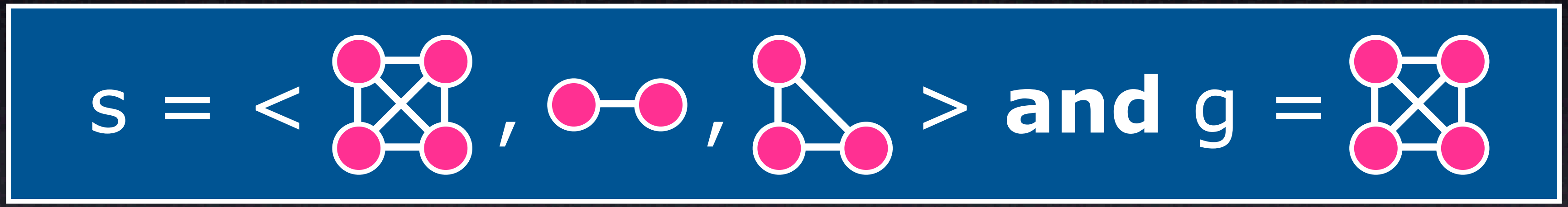
and x = ??

and v1 = #v1 **and** v2 = #v2 **and** . . .

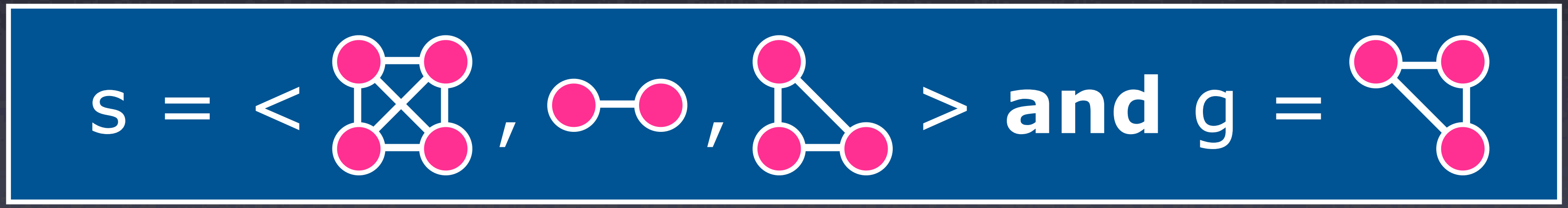
Three most important facts for Java programmers to remember about Tako

1. Everything is *an object*
2. *Everything* is an object
3. See facts 1 & 2

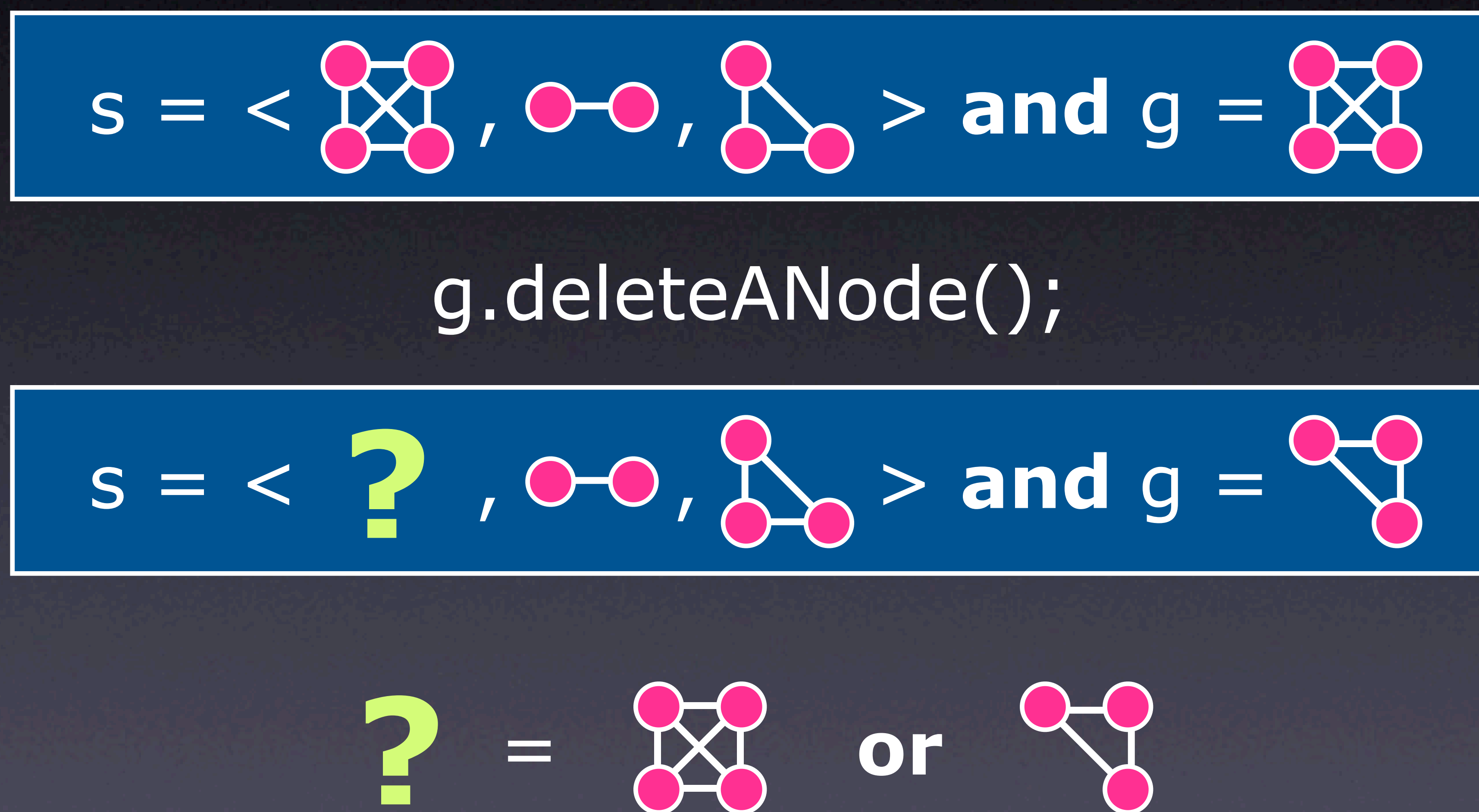
Reasoning about a Tako Stack




```
g.deleteANode();
```

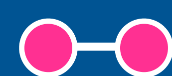
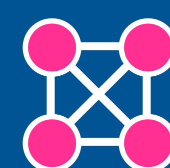
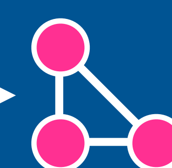


Reasoning about a Java Stack



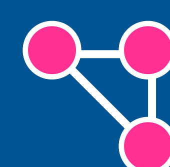
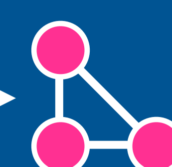
Reasoning Unambiguously about a Java Stack

s = <@26, @18, @47> **and** g = @26 **and**

GMAP = { @18 ↦  , @26 ↦  , @47 ↦  }

g.deleteANode();

s = <@26, @18, @47> **and** g = @26 **and**

GMAP = { @18 ↦  , @26 ↦  , @47 ↦  }

Pointer Component

- A pointer component is provided for times when programmers need the behavior and performance of pointer (as in implementing linked data structures)
- It is similar to the RESOLVE pointer component (see Kulczycki in SAVCBS2005)
- Model uses *conceptual variables*

Tracing Table for a Method

- Given: an example program state that conforms to the method's precondition
- After each statement: update the state based on the statement's specification
- Final state: should be consistent with the specification of the original method

```
public void reverse( )
    ensures this = REV(#this);
{
    Stack temp;
    Object x;
    while (this.depth( ) != 0)
        decreasing . . .
        maintaining . . .
    {
        this.pop(x);
        temp.push(x);
    }
    this :=: temp;
}
```

pre-state: **this** = < 3, 4 >

0	
1	
2	
3	
4	
5	

```
while (this.depth() != 0) {

    this.pop(x);

    temp.push(x);

}

this :=: temp;
```

expected post-state: **this** = < 3, 4 >

pre-state: **this** = < 3, 4 >

0	**this** = < 3, 4 > **and** temp = < > **and** x = 0
1	
2	
3	
4	
5	

```
while (this.depth() != 0) {
    this.pop(x);
    temp.push(x);
}
this :=: temp;
```

expected post-state: **this** = < 3, 4 >

pre-state: **this** = < 3, 4 >

0	**this** = < 3, 4 > **and** temp = < > **and** x = 0	
		while (**this**.depth() != 0) {
1	**this** = < 3, 4 > **and** temp = < > **and** x = 0	
		this.pop(x);
2		
		temp.push(x);
3		
		}
4		
		this :=: temp;
5		

expected post-state: **this** = < 3, 4 >

pre-state: **this** = < 3, 4 >

	State
0	**this** = < 3, 4 > **and** temp = < > **and** x = 0
1	**this** = < 3, 4 > **and** temp = < > **and** x = 0
2	**this** = < 4 > **and** temp = < > **and** x = 3
3	
4	
5	

```
while (this.depth() != 0) {
    this.pop(x);
    temp.push(x);
}
this :=: temp;
```

expected post-state: **this** = < 3, 4 >

pre-state: **this** = < 3, 4 >

0	**this** = < 3, 4 > **and** temp = < > **and** x = 0	
		while (this.depth() != 0) {
1	**this** = < 3, 4 > **and** temp = < > **and** x = 0	
		this.pop(x);
2	**this** = < 4 > **and** temp = < > **and** x = 3	
		temp.push(x);
3	**this** = < 4 > **and** temp = < 3 > **and** x = ??	
		}
4		
		this :=: temp;
5		

expected post-state: **this** = < 3, 4 >

pre-state: **this** = < 3, 4 >

0	**this** = < 3, 4 > **and** temp = < > **and** x = 0	
1	**this** = < 3, 4 > **and** temp = < > **and** x = 0	
2	**this** = < 4 > **and** temp = < > **and** x = 3	
3	**this** = < 4 > **and** temp = < 3 > **and** x = ??	
4		
5		

```
while (this.depth() != 0) {

    this.pop(x);

    temp.push(x);

}   // this.depth() != 0

this :=: temp;
```

expected post-state: **this** = < 3, 4 >

pre-state: **this** = < 3, 4 >

0	**this** = < 3, 4 > **and** temp = < > **and** x = 0	
1	**this** = < 3, 4 > **and** temp = < > **and** x = 0	**this** = < 4 > **and** temp = < 3 > **and** x = ??
2	**this** = < 4 > **and** temp = < > **and** x = 3	**this** = < > **and** temp = < 3 > **and** x = 4
3	**this** = < 4 > **and** temp = < 3 > **and** x = ??	**this** = < > **and** temp = < 4, 3 > **and** x = ??
4		
5		

```
while (this.depth() != 0) {

    this.pop(x);

    temp.push(x);

}

this :=: temp;
```

expected post-state: **this** = < 3, 4 >

pre-state: **this** = < 3, 4 >

0	**this** = < 3, 4 > **and** temp = < > **and** x = 0	
1	**this** = < 3, 4 > **and** temp = < > **and** x = 0	**this** = < 4 > **and** temp = < 3 > **and** x = ??
2	**this** = < 4 > **and** temp = < > **and** x = 3	**this** = < > **and** temp = < 3 > **and** x = 4
3	**this** = < 4 > **and** temp = < 3 > **and** x = ??	**this** = < > **and** temp = < 4, 3 > **and** x = ??
4		
5		

```
while (this.depth() != 0) {

    this.pop(x);

    temp.push(x);

}   // this.depth() = 0

this :=: temp;
```

expected post-state: **this** = < 3, 4 >

pre-state: **this** = < 3, 4 >

0	**this** = < 3, 4 > **and** temp = < > **and** x = 0	
1	**this** = < 3, 4 > **and** temp = < > **and** x = 0	**this** = < 4 > **and** temp = < 3 > **and** x = ??
2	**this** = < 4 > **and** temp = < > **and** x = 3	**this** = < > **and** temp = < 3 > **and** x = 4
3	**this** = < 4 > **and** temp = < 3 > **and** x = ??	**this** = < > **and** temp = < 4, 3 > **and** x = ??
4	**this** = < > **and** temp = < 4, 3 > **and** x = ??	
5		

```
while (this.depth() != 0) {

    this.pop(x);

    temp.push(x);

}

this :=: temp;
```

expected post-state: **this** = < 3, 4 >

pre-state: **this** = < 3, 4 >

0	**this** = < 3, 4 > **and** temp = < > **and** x = 0	
1	**this** = < 3, 4 > **and** temp = < > **and** x = 0	**this** = < 4 > **and** temp = < 3 > **and** x = ??
2	**this** = < 4 > **and** temp = < > **and** x = 3	**this** = < > **and** temp = < 3 > **and** x = 4
3	**this** = < 4 > **and** temp = < 3 > **and** x = ??	**this** = < > **and** temp = < 4, 3 > **and** x = ??
4	**this** = < > **and** temp = < 4, 3 > **and** x = ??	
5	**this** = < 4, 3 > **and** temp = < > **and** x = ??	

```
while (this.depth() != 0) {

    this.pop(x);

    temp.push(x);

}

this :=: temp;
```

expected post-state: **this** = < 3, 4 >

Symbolic Reasoning Table

- Used to prove an implementation is correct with respect to its specification
- Assume precondition; show postcondition
- For each state you have:
  - Path condition – What has to be true in order for us to get to this state?
  - Facts – What do we know in this state?
  - Obligations – What do we need to prove before we can move to the next state?

Proof of Correctness

- To prove an implementation correct with respect to its specification:
  - Construct a symbolic reasoning table
  - Prove that all the obligations are true

```
public void reverse( )
    ensures this = REV(#this);
{
    Stack temp;
    Object x;
    while (this.depth( ) != 0)
        decreasing |this|;
        maintaining REV(temp) o this = #this;
    {
        this.pop(x);
        temp.push(x);
    }
    this :=: temp;
}
```

	PC	Facts	Obligations
0			
1			
2			
3			
4			
5			

```
while (this.depth() != 0) {
    this.pop(x);
    temp.push(x);
}
this :=: temp;
```

	PC	Facts	Obligations
0			
1			
2			
3			
4			
5			

maintaining
REV(temp) o **this**
= #**this**

```
while (this.depth() != 0) {

    this.pop(x);

    temp.push(x);

}

this :=: temp;
```

	PC	Facts	Obligations
0			
1		REV($temp_1$) o **this**$_1$ = #**this**	
2			
3			REV($temp_3$) o **this**$_3$ = #**this**
4			
5			

maintaining
REV(temp) o **this**
= #**this**

```
while (this.depth() != 0) {

    this.pop(x);

    temp.push(x);

}

this :=: temp;
```

	PC	Facts	Obligations
0			
1		REV($temp_1$) o **$this_1$** = #**this**	
2			
3			REV($temp_3$) o **$this_3$** = #**this**
4			
5			

decreasing |this|

```
while (this.depth() != 0) {
    this.pop(x);
    temp.push(x);
}
this :=: temp;
```

PC		Facts	Obligations
0			
1		REV($temp_1$) o **this**$_1$ = #**this and** x_1 = ??	
2			
3			REV($temp_3$) o **this**$_3$ = #**this and** |**this**$_3$| < |**this**$_1$|
4			
5			

decreasing |this|

```
while (this.depth() != 0) {
    this.pop(x);
    temp.push(x);
}
this :=: temp;
```

$|\mathbf{this}_{pre\_1}| = 0$ **and** $\mathbf{this}_1 = \mathbf{this}_{pre\_1}$

	PC	Facts	Obligations
0			
1		$REV(temp_1) \circ \mathbf{this}_1 = \#\mathbf{this}$ **and** $x_1 = ??$	
2			
3			$REV(temp_3) \circ \mathbf{this}_3 = \#\mathbf{this}$ **and** $\lvert\mathbf{this}_3\rvert < \lvert\mathbf{this}_1\rvert$
4			
5			

```
while (this.depth() != 0) {

    this.pop(x);

    temp.push(x);

}

this :=: temp;
```

	PC	Facts	Obligations
0			
1	$\lvert \mathbf{this}_1 \rvert \neq 0$	REV($temp_1$) o $\mathbf{this}_1$ = #**this and** x_1 = ??	
2	$\lvert \mathbf{this}_1 \rvert \neq 0$		
3	$\lvert \mathbf{this}_1 \rvert \neq 0$		REV($temp_3$) o $\mathbf{this}_3$ = #**this and** $\lvert \mathbf{this}_3 \rvert < \lvert \mathbf{this}_1 \rvert$
4			
5			

$\lvert \mathbf{this}_{pre\_1} \rvert = 0$ **and** $\mathbf{this}_1 = \mathbf{this}_{pre\_1}$

```
while (this.depth() != 0) {

    this.pop(x);

    temp.push(x);

}

this :=: temp;
```

requires
|**this**| > 0

	PC	Facts	Obligations
0			
1	$\|\mathbf{this}_1\| \neq 0$	$REV(temp_1)$ o $\mathbf{this}_1$ = #**this and** x_1 = ??	
2	$\|\mathbf{this}_1\| \neq 0$		
3	$\|\mathbf{this}_1\| \neq 0$		$REV(temp_3)$ o $\mathbf{this}_3$ = #**this and** $\|\mathbf{this}_3\| < \|\mathbf{this}_1\|$
4			
5			

```
while (this.depth() != 0) {
    this.pop(x);
    temp.push(x);
}
this :=: temp;
```

	PC	Facts	Obligations
0			
1	$\lvert \textbf{this}_1 \rvert \neq 0$	REV($temp_1$) o $\textbf{this}_1$ = #**this and** x_1 = ??	$\lvert \textbf{this}_1 \rvert > 0$
2	$\lvert \textbf{this}_1 \rvert \neq 0$		
3	$\lvert \textbf{this}_1 \rvert \neq 0$		REV($temp_3$) o $\textbf{this}_3$ = #**this and** $\lvert \textbf{this}_3 \rvert < \lvert \textbf{this}_1 \rvert$
4			
5			

requires
|**this**| > 0

```
while (this.depth() != 0) {

    this.pop(x);

    temp.push(x);

}

this :=: temp;
```

ensures #this = <x> o this

	PC	Facts	Obligations
0			
1	$\lvert \mathbf{this}_1 \rvert \neq 0$	$REV(temp_1) \circ \mathbf{this}_1 = \#\mathbf{this}$ **and** $x_1 = ??$	$\lvert \mathbf{this}_1 \rvert > 0$
2	$\lvert \mathbf{this}_1 \rvert \neq 0$		
3	$\lvert \mathbf{this}_1 \rvert \neq 0$		$REV(temp_3) \circ \mathbf{this}_3 = \#\mathbf{this}$ **and** $\lvert \mathbf{this}_3 \rvert < \lvert \mathbf{this}_1 \rvert$
4			
5			

```
while (this.depth() != 0) {
    this.pop(x);
    temp.push(x);
}
this :=: temp;
```

```
ensures #this =
  <x> o this
```

	PC	Facts	Obligations
0			
1	$\|this_1\| \neq 0$	$REV(temp_1)\ o\ this_1 = \#this$ and $x_1 = ??$	$\|this_1\| > 0$
2	$\|this_1\| \neq 0$	$this_1 = <x_2>\ o\ this_2$ and $temp_2 = temp_1$	
3	$\|this_1\| \neq 0$		$REV(temp_3)\ o\ this_3 = \#this$ and $\|this_3\| < \|this_1\|$
4			
5			

```
while (this.depth() != 0) {

    this.pop(x);

    temp.push(x);

}

this :=: temp;
```

	PC	Facts	Obligations
0			
1	$\lvert \mathbf{this}_1 \rvert \neq 0$	REV($temp_1$) o $\mathbf{this}_1$ = #**this and** x_1 = ??	$\lvert \mathbf{this}_1 \rvert > 0$
2	$\lvert \mathbf{this}_1 \rvert \neq 0$	$\mathbf{this}_1$ = <x_2> o $\mathbf{this}_2$ **and** $temp_2 = temp_1$	
3	$\lvert \mathbf{this}_1 \rvert \neq 0$		REV($temp_3$) o $\mathbf{this}_3$ = #**this and** $\lvert \mathbf{this}_3 \rvert < \lvert \mathbf{this}_1 \rvert$
4			
5			

-- no requires clause for push

```
while (this.depth() != 0) {
    this.pop(x);
    temp.push(x);
}
this :=: temp;
```

ensures this =
<#x> o #this

	PC	Facts	Obligations
0			
1	$\lvert\textbf{this}_1\rvert \neq 0$	REV($temp_1$) o $\textbf{this}_1$ = #**this and** x_1 = ??	$\lvert\textbf{this}_1\rvert > 0$
2	$\lvert\textbf{this}_1\rvert \neq 0$	$\textbf{this}_1$ = <x_2> o $\textbf{this}_2$ **and** $temp_2 = temp_1$	
3	$\lvert\textbf{this}_1\rvert \neq 0$		REV($temp_3$) o $\textbf{this}_3$ = #**this and** $\lvert\textbf{this}_3\rvert < \lvert\textbf{this}_1\rvert$
4			
5			

```
while (this.depth() != 0) {

    this.pop(x);

    temp.push(x);

}

this :=: temp;
```

ensures this = <#x> o #this

	PC	Facts	Obligations
0			
1	$\lvert \textbf{this}_1 \rvert \neq 0$	$\text{REV}(\text{temp}_1)\ o\ \textbf{this}_1 = \#\textbf{this}$ **and** $x_1 = ??$	$\lvert \textbf{this}_1 \rvert > 0$
2	$\lvert \textbf{this}_1 \rvert \neq 0$	$\textbf{this}_1 = \langle x_2 \rangle\ o\ \textbf{this}_2$ **and** $\text{temp}_2 = \text{temp}_1$	
3	$\lvert \textbf{this}_1 \rvert \neq 0$	$\textbf{this}_3 = \langle x_2 \rangle\ o\ \textbf{this}_2$ **and** $x_3 = ??$ **and** $\text{temp}_3 = \text{temp}_2$	$\text{REV}(\text{temp}_3)\ o\ \textbf{this}_3 = \#\textbf{this}$ **and** $\lvert \textbf{this}_3 \rvert < \lvert \textbf{this}_1 \rvert$
4			
5			

```
while (this.depth() != 0) {

    this.pop(x);

    temp.push(x);

}

this :=: temp;
```

	PC	Facts	Obligations
0			
1	$\|$**this**$_1\| \neq 0$	REV($temp_1$) o **this**$_1$ = #**this and** x_1 = ??	$\|$**this**$_1\| > 0$
2	$\|$**this**$_1\| \neq 0$	**this**$_1$ = <x_2> o **this**$_2$ **and** $temp_2 = temp_1$	
3	$\|$**this**$_1\| \neq 0$	**this**$_3$ = <x_2> o **this**$_2$ **and** x_3 = ?? **and** $temp_3 = temp_2$	REV($temp_3$) o **this**$_3$ = #**this and** $\|$**this**$_3\| < \|$**this**$_1\|$
4			
5			

-- other parts deal with initialization and termination of loop invariant

```
while (this.depth() != 0) {
    this.pop(x);
    temp.push(x);
}
this :=: temp;
```

	PC	Facts	Obligations
0		Object.**is_init**(x_0) **and** $temp_0 = \langle\ \rangle$	$\lvert\mathbf{this}_0\rvert \neq 0$ **implies** $REV(temp_0) \circ \mathbf{this}_0 = \#\mathbf{this}$
1	$\lvert\mathbf{this}_1\rvert \neq 0$	$REV(temp_1) \circ \mathbf{this}_1 = \#\mathbf{this}$ **and** $x_1 = ??$	$\lvert\mathbf{this}_1\rvert > 0$
2	$\lvert\mathbf{this}_1\rvert \neq 0$	$\mathbf{this}_1 = \langle x_2\rangle \circ \mathbf{this}_2$ **and** $temp_2 = temp_1$	
3	$\lvert\mathbf{this}_1\rvert \neq 0$	$\mathbf{this}_3 = \langle x_2\rangle \circ \mathbf{this}_2$ **and** $x_3 = ??$ **and** $temp_3 = temp_2$	$REV(temp_3) \circ \mathbf{this}_3 = \#\mathbf{this}$ **and** $\lvert\mathbf{this}_3\rvert < \lvert\mathbf{this}_1\rvert$
4	$\lvert\mathbf{this}_4\rvert = 0$	$REV(temp_4) \circ \mathbf{this}_4 = \#\mathbf{this}$ **and** $x_4 = ??$	
5	$\lvert\mathbf{this}_4\rvert = 0$	$\mathbf{this}_5 = temp_4$ **and** $temp_5 = \mathbf{this}_4$ **and** $x_5 = x_4$	$\mathbf{this}_5 = REV(\#\mathbf{this})$

```
while (this.depth() != 0) {

    this.pop(x);

    temp.push(x);

}

this :=: temp;
```

	PC	Facts	Obligations
0		Object.**is_init**(x_0) **and** $temp_0$ = < >	|**this**$_0$| ≠ 0 **implies** REV($temp_0$) o **this**$_0$ = #**this**
1	|**this**$_1$| ≠ 0	REV($temp_1$) o **this**$_1$ = #**this and** x_1 = ??	|**this**$_1$| > 0
2	|**this**$_1$| ≠ 0	**this**$_1$ = <x_2> o **this**$_2$ **and** $temp_2$ = $temp_1$	
3	|**this**$_1$| ≠ 0	**this**$_3$ = <x_2> o **this**$_2$ **and** x_3 = ?? **and** $temp_3$ = $temp_2$	REV($temp_3$) o **this**$_3$ = #**this and** |**this**$_3$| < |**this**$_1$|
4	|**this**$_4$| = 0	REV($temp_4$) o **this**$_4$ = #**this and** x_4 = ??	
5	|**this**$_4$| = 0	**this**$_5$ = $temp_4$ **and** $temp_5$ = **this**$_4$ **and** x_5 = x_4	**this**$_5$ = REV(#**this**)

```
while (this.depth() != 0) {

    this.pop(x);

    temp.push(x);

}

this :=: temp;
```

Show: $\mathrm{REV}(temp_3) \circ \mathbf{this}_3 = \#\mathbf{this}$

Fact 1: $\mathrm{REV}(temp_1) \circ \mathbf{this}_1 = \#\mathbf{this}$ **and** $x_1 = ??$

Fact 2: $\mathbf{this}_1 = \langle x_2 \rangle \circ \mathbf{this}_2$ **and** $temp_2 = temp_1$

Fact 3: $\mathbf{this}_3 = \langle x_2 \rangle \circ \mathbf{this}_2$ **and** $x_3 = ??$ **and** $temp_3 = temp_2$

Show: $\mathrm{REV}(temp_3) \text{ o } \mathbf{this}_3 = \#\mathbf{this}$

or

$\mathrm{REV}(temp_2) \text{ o } \langle x_2 \rangle \text{ o } \mathbf{this}_2 = \#\mathbf{this}$ (by Fact 3)

Fact 1: $\mathrm{REV}(temp_1) \text{ o } \mathbf{this}_1 = \#\mathbf{this}$ **and** $x_1 = ??$

Fact 2: $\mathbf{this}_1 = \langle x_2 \rangle \text{ o } \mathbf{this}_2$ **and** $temp_2 = temp_1$

Fact 3: $\mathbf{this}_3 = \langle x_2 \rangle \text{ o } \mathbf{this}_2$ **and** $x_3 = ??$ **and** $temp_3 = temp_2$

Show: REV($temp_3$) o **this**$_3$ = #**this**

or

REV($temp_2$) o <x_2> o **this**$_2$ = #**this** (by Fact 3)

or

REV($temp_1$) o **this**$_1$ = #**this** (by Fact 2)

Fact 1: REV($temp_1$) o **this**$_1$ = #**this and** x_1 = ??

Fact 2: **this**$_1$ = <x_2> o **this**$_2$ **and** $temp_2$ = $temp_1$

Fact 3: **this**$_3$ = <x_2> o **this**$_2$ **and** x_3 = ?? **and** $temp_3$ = $temp_2$

Show: $\mathrm{REV}(temp_3)\ o\ \mathbf{this}_3 = \#\mathbf{this}$

or

$\mathrm{REV}(temp_2)\ o\ \langle x_2 \rangle\ o\ \mathbf{this}_2 = \#\mathbf{this}$ (by Fact 3)

or

$\mathrm{REV}(temp_1)\ o\ \mathbf{this}_1 = \#\mathbf{this}$ (by Fact 2)

or

$\#\mathbf{this} = \#\mathbf{this}$ (by Fact 1)

Fact 1: $\mathrm{REV}(temp_1)\ o\ \mathbf{this}_1 = \#\mathbf{this}$ **and** $x_1 = ??$

Fact 2: $\mathbf{this}_1 = \langle x_2 \rangle\ o\ \mathbf{this}_2$ **and** $temp_2 = temp_1$

Fact 3: $\mathbf{this}_3 = \langle x_2 \rangle\ o\ \mathbf{this}_2$ **and** $x_3 = ??$ **and** $temp_3 = temp_2$

Show: $\text{REV}(\text{temp}_3) \text{ o } \mathbf{this}_3 = \#\mathbf{this}$

or

$\text{REV}(\text{temp}_2) \text{ o } <x_2> \text{ o } \mathbf{this}_2 = \#\mathbf{this}$ (by Fact 3)

or

$\text{REV}(\text{temp}_1) \text{ o } \mathbf{this}_1 = \#\mathbf{this}$ (by Fact 2)

or

✓ $\#\mathbf{this} = \#\mathbf{this}$ (by Fact 1)

Fact 1: $\text{REV}(\text{temp}_1) \text{ o } \mathbf{this}_1 = \#\mathbf{this}$ **and** $x_1 = ??$

Fact 2: $\mathbf{this}_1 = <x_2> \text{ o } \mathbf{this}_2$ **and** $\text{temp}_2 = \text{temp}_1$

Fact 3: $\mathbf{this}_3 = <x_2> \text{ o } \mathbf{this}_2$ **and** $x_3 = ??$ **and** $\text{temp}_3 = \text{temp}_2$

Tako compiler

- SourceForge project: takocompiler
- Text-based adventure program:
  - ~40 classes
  - ~4000 lines of code

Questions

Inheritance and in-out parameter passing

- void swapMethod(Object x, Object y) {
  x :=: y;
  }

- Cat c; Dog d;
  swapMethod(c, d);