On the Simplicity of Synthesizing Linked Data Structure Operations

Darya Kurilova, Carnegie Mellon University
Derek Rayside, University of Waterloo
Ideal Code Generation Scenario

I want the program to...
...
...

*.java
*.exe
*.c
*.js
...

Magic Box

© Darya Kurilova 2013
Proposed Code Generation Scenario

@SpecField(...)  @Modifies(...)  ...
...
...

*.java  *.class  *.exe

SAT Solver
Our Approach

• **Given:**
  – Prior work on code synthesis
  – SAT-solver limits

• **Problem:**
  – Generating code for data structures that require *heap manipulations* is believed to be hard

• **Our solution:**
  – Use *“divide-and-conquer”* and a *common pattern* for code generation
  – *Proof of concept*: code generation via systematic manual procedure
Common Pattern

```java
public return-type method-name(arguments) {
    if (condition\textsubscript{1} \&\& condition\textsubscript{2} \&\& ... \&\& condition\textsubscript{N1}) {
        statement\textsubscript{1};
        statement\textsubscript{2};
        ...
        statement\textsubscript{M1};
        return object-of-the-return-type-or-nothing;
    }

    ...

    if (condition\textsubscript{1} \&\& condition\textsubscript{2} \&\& ... \&\& condition\textsubscript{Nk}) {
        statement\textsubscript{1};
        statement\textsubscript{2};
        ...
        statement\textsubscript{Mk};
        return object-of-the-return-type-or-nothing;
    }
}
```
AVL Trees

• A self-balancing binary search tree
• Difference between height of the left subtree and of the right subtree is at most 1
Code Generation Algorithm

1. Part I: Programmer’s Actions
2. Part II: Automatic Actions
Code Generation Algorithm: Part I
(Programmer’s Actions)

1. Define basic data structures.
2. Write invariants.
3. Declare and specify methods.
Code Generation Algorithm: Part I
(Programmer’s Actions)

1. Define basic data structures.
2. Write invariants.
3. Declare and specify methods.
Basic Data Structures

1 class Node {
2   int value;
3   int height;
4   Node left;
5   Node right;
6   Node parent;
7 }

8 class AVLTree {
9   Node root;
10 }

© Darya Kurilova 2013
Code Generation Algorithm: Part I (Programmer’s Actions)

1. Define basic data structures.
2. Write invariants.
3. Declare and specify methods.
Invariants for an AVL Tree

@Invariant({
  // root has no parents
  "no (left + right).(this.root)",

  // acyclic
  "no this in this.^((left + right))",

  // ordered
  "all x : this.left.^((left + right)) + this.left - null | x.value < this.value",
  "all x : this.right.^((left + right)) + this.right - null | x.value > this.value",

  // height of a leaf node is 1
  "((this.left = null and this.right = null) => this.height = 1",

  // height for semi-leaf nodes
  "((this.left = null and this.right != null) => this.height = this.right.height + 1",
  "((this.left != null and this.right = null) => this.height = this.left.height + 1",

  // height for interior nodes
  "((this.left != null and this.right = null) => this.height = this.left.height + 1",
  "this.left != null => this.height > this.left.height",
  "this.right != null => this.height > this.right.height",
  "no null in this.(left + right) => this.left.height - this.right.height <= 1",
  "no null in this.(left + right) => this.right.height - this.left.height <= 1",
  "no null in this.(left + right) => (this.height = this.left.height + 1)
      or (this.height = this.right.height + 1)"
})
Code Generation Algorithm: Part I
(Programmer’s Actions)

1. Define basic data structures.
2. Write invariants.
3. Declare and specify methods.
Methods and Their Specifications

@Pure
@Returns("x in this.values")
public boolean contains(int x) {
    // generated code goes here
}

@Modifies("Node.value, Node.height, Node.left, Node.right, Node.parent, this.root")
@Ensures("x in this.values")
public void insert(int x) {
    // generated code goes here
}

@Modifies("Node.value, Node.height, Node.left, Node.right, Node.parent, this.root")
@Ensures("no x in this.values")
public void delete(int x) {
    // generated code goes here
}
Code Generation Algorithm: Part II
(Automatic Actions)

1. Generate helper methods and their specifications *based on basic data structures definitions*.

2. Generate helper methods and their specifications *based on invariants*.

3. Determine the helper methods’ call order.
Code Generation Algorithm: Part II (Automatic Actions)

1. Generate helper methods and their specifications based on basic data structures definitions.

2. Generate helper methods and their specifications based on invariants.

3. Determine the helper methods’ call order.
swapChildren() Helper Method

1  @Ensures("this.right = @old(this.left)
2       and this.left = @old(this.right)")
3
4  private void swapChildren(Node one) {
5      Node temp = one.right;
6      one.right = one.left;
7      one.left = temp;
8  }

© Darya Kurilova 2013
Code Generation Algorithm: Part II
(Automatic Actions)

1. Generate helper methods and their specifications based on basic data structures definitions.

2. Generate helper methods and their specifications based on invariants.
   1) Analyze invariants.
   2) Generate helper methods’ bodies.
   3) Clean up.

3. Determine the helper methods’ call order.
Invariant Analysis

@Invariant({
    // root has no parents
    "no (left + right).(this.root)",

    // acyclic
    "no this in this.(left + right)",

    // ordered
    "all x : this.left.(left + right) + this.left - null | x.value < this.value",
    "all x : this.right.(left + right) + this.right - null | x.value > this.value",

    // height of a leaf node is 1
    "(this.left = null and this.right = null) => this.height = 1",

    // height for semi-leaf nodes
    "(this.left = null and this.right != null) => this.height = this.right.height + 1",
    "(this.left != null and this.right = null) => this.height = this.left.height + 1",

    // height for interior nodes
    "(this.left != null and this.right = null) => this.height = this.left.height + 1",
    "this.left != null => this.height > this.left.height",
    "this.right != null => this.height > this.right.height",
    "no null in this.(left + right) => this.left.height - this.right.height <= 1",
    "no null in this.(left + right) => this.right.height - this.left.height <= 1",
    "no null in this.(left + right) => (this.height = this.left.height + 1) or (this.height = this.right.height + 1)"
})
Invariant Analysis Conclusion

• Invariants that mention the \textit{value} field are related to \textit{ordering}.
• Invariants that mention the \textit{height} field are related to \textit{balancing}.
• Therefore:
  – \textit{two} groups of invariants,
  – \textit{two} main properties, and
  – \textit{two} helper methods.
@Pure
@Returns("x in this.values")
public boolean contains(int x) {
    @Pure
    @Returns("x in this.values")
    containsHelper(...);
}
@Modifies("Node.value, Node.height, Node.left, Node.right, Node.parent, this.root")
@Ensures("x in this.values")
public void insert(int x) {
    @Modifies("Node.value, Node.left, Node.right, Node.parent, this.root")
    @Ensures("x in this.values")
    orderingInsertHelper(...);
    @Modifies("Node.height, Node.left, Node.right, Node.parent, this.root")
    balancingInsertHelper(...);
}
@Modifies("Node.value, Node.height, Node.left, Node.right, Node.parent, this.root")
@Ensures("no x in this.values")
public void delete(int x) {
    @Modifies("Node.value, Node.left, Node.right, Node.parent, this.root")
    @Ensures("no x in this.values")
    orderingDeleteHelper(...);
    @Modifies("Node.height, Node.left, Node.right, Node.parent, this.root")
    balancingDeleteHelper(...);
}
Code Generation Algorithm: Part II (Automatic Actions)

1. Generate helper methods and their specifications based on basic data structures definitions.

2. Generate helper methods and their specifications based on invariants.
   1) Analyze invariants.
   2) Generate helper methods’ bodies.
   3) Clean up.

3. Determine the helper methods’ call order.
Common Pattern

1. public return-type method-name(arguments) {
2.   if (condition\_1 && condition\_2 && ... && condition\_N\_1) {
3.     statement\_1;
4.     statement\_2;
5.     ...
6.     statement\_M\_1;
7.     return object-of-the-return-type-or-nothing;
8.   }
9.   ...
10. if (condition\_1 && condition\_2 && ... && condition\_N\_k) {
11.    statement\_1;
12.    statement\_2;
13.    ...
14.    statement\_M\_k;
15.    return object-of-the-return-type-or-nothing;
16.  }
17.}
Pattern Fillers

• Conditions:
  – Mathematical comparisons using ==, !=, <, <=, >, >=.
  – **Node** fields, **null**, and numeric values from specifications.

• Executable statements:
  – a recursive call to the method itself,
  – a call to another method,
  – an assignment to an existing variable,
  – a variable declaration,
  – an assignment to the introduced variable.
public boolean containsHelper(int x, Node node) {
    if (node != null && node.left == null && node.right == null && node.value > x) {
        return false;
    }
    if (node != null && node.left == null && node.right == null && node.value < x) {
        return false;
    }
    if (node != null && node.left == null && node.right == null && node.value == x) {
        return true;
    }
    if (node != null && node.left == null && node.right != null && node.value > x) {
        return false;
    }
    if (node != null && node.left == null && node.right != null && node.value < x) {
        return containsHelper(x, node.right);
    }
    ... // 42 MORE IF-BLOCKS
    if (node == null && node.left != null && node.right != null && node.value == x) {
        return false;
    }
}
Code Generation Algorithm: Part II (Automatic Actions)

1. Generate helper methods and their specifications based on basic data structures definitions.

2. Generate helper methods and their specifications based on invariants.
   1) Analyze invariants.
   2) Generate helper methods’ bodies.
   3) Clean up.

3. Determine the helper methods’ call order.
containsHelper() After Clean Up

```java
public boolean containsHelper(int x, Node node) {
    if (node == null) {
        return false;
    }
    if (node.value == x) {
        return true;
    }
    if (node.value < x) {
        if (node.right != null) {
            return containsHelper(x, node.right);
        } else {
            return false;
        }
    }
    if (node.value > x) {
        if (node.left != null) {
            return containsHelper(x, node.left);
        } else {
            return false;
        }
    }
}
```
Code Generation Algorithm: Part II (Automatic Actions)

1. Generate helper methods and their specifications based on basic data structures definitions.

2. Generate helper methods and their specifications based on invariants.
   1) Analyze invariants.
   2) Generate helper methods’ bodies.
   3) Clean up.

3. Determine the helper methods’ call order.
Helper Methods’ Call Order

```java
@Pure
@Returns("x in this.values")
public boolean contains(int x) {
    @Pure
    @Returns("x in this.values")
    containsHelper(...);
}

@Modifies("Node.value, Node.height, Node.left, Node.right, Node.parent, this.root")
@Ensures("x in this.values")
public void insert(int x) {
    @Modifies("Node.value, Node.left, Node.right, Node.parent, this.root")
    @Ensures("x in this.values")
    orderingInsertHelper(...);

    @Modifies("Node.height, Node.left, Node.right, Node.parent, this.root")
    balancingInsertHelper(...);
}

@Modifies("Node.value, Node.height, Node.left, Node.right, Node.parent, this.root")
@Ensures("no x in this.values")
public void delete(int x) {
    @Modifies("Node.value, Node.left, Node.right, Node.parent, this.root")
    @Ensures("no x in this.values")
    orderingDeleteHelper(...);

    @Modifies("Node.height, Node.left, Node.right, Node.parent, this.root")
    balancingDeleteHelper(...);
}
```

© Darya Kurilova 2013
Code Generation Algorithm Summary

Part I: Programmer’s Actions

1. Define basic data structures.
2. Write invariants.
3. Declare and specify methods.

Part II: Automatic Actions

1. Generate helper methods and their specifications based on basic data structures definitions.
2. Generate helper methods and their specifications based on invariants.
   1) Analyze the invariants.
   2) Generate helper methods’ bodies.
   3) Clean up.
3. Determine the helper methods’ call order.
Code Generation Algorithm Summary

Part I: Programmer’s Actions

1. Define basic data structures.
2. Write invariants.
3. Declare and specify methods.

Part II: Automatic Actions

1. Generate helper methods and their specifications based on basic data structures definitions.

2. Generate helper methods and their specifications based on invariants.
   1) Analyze the invariants.
   2) Generate helper methods’ bodies.
   3) Clean up.

3. Determine the helper methods’ call order.
Search Space Analysis

• Usual case:
  – 6 statements + 2 levels of dereferencing = $2^{30}$
  – Can be handled by modern SAT solvers

• Exception: tree re-balance helper method
  – 12 statements + 3 levels of dereferencing = $2^{84}$
  – Mitigated by “smart” order in which types of statements are introduced
Conclusion

• Use “divide-and-conquer” and the common pattern
• Search spaces manageable by current SAT-solvers
• Problem might be not as difficult as previously imagined
• Approach is applicable to recursive linked data structures; possibly to others