High-Level Expressions in Language L

Hesam Samimi

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Memo M-2009-009

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

Hesam Samimi
09/28/2009

High-level Expressions in Language L

I. Introduction

Formal specifications, often used to declare global *invariants* for a class, as well as *pre-* and *post-conditions* for methods, are usually expressed in high level syntax involving first-order and relational logic. Moreover, it is desirable for specifications to be part of and written in the same language as the implemented program itself, so that they can be run and tested like the rest of the code. We suggest syntactic sugars to be supported for these expressions, including quantified expressions, as well as transitive closures of field accesses, useful for recursive data types. But support for high-level expressions in the specifications is not just a matter of providing more readable and mathematically formed code. Unlike normal control structures such as loops, boolean expressions written in first order and relational logic carry the semantics necessary to enable translating them into formulas in the syntax of external constraint solving tools. The specifications are made executable by resorting to constraint solvers to find models for the specified formulas, and then applying the values in the models to the objects.

A lightweight manner to build an extension of a programming language to support these high-level expressions is to only allow them inside specially marked methods, say with keyword *function* in the Java language. The compiler specially handles these high-level function declarations and compiles the desugared version of them instead. For each of the high-level methods, the compiler also instruments an extra method that generates the formula specified by the expression, in the syntax of the external constraint solver used. For specifications to be executed, the logical counterpart of such methods will be called to generate the code to pass onto the external model extracting tool.

The next section lists the high level expressions we propose to be supported in an object oriented language. After that, a few points on the dynamic translation of such expressions into the syntax of the logical constraint solver are discussed. A fully runnable red black tree implementation with insert and delete operations written in an extension of Java supporting these expressions is included.

II. High-level Expressions

The relational logic language *Alloy*, used for object modeling, supports more high-level, flexible expressions than most object oriented languages. The expressions we list below are all adapted from this language. Given that the underlying inference tool we use to execute specifications supports a higher level of flexibility in the expressions, the one-time, light amount of work to support the syntactic sugars in our object oriented language should be easily justifiable.

All of the expressions below are desugared into method calls, since those expressions may have occurred as a sub-expression, and thus cannot be substituted for statements. For instance they may appear as a literal in a conjunction of boolean expressions, or may be used as the receiver of a method call, etc. All but the last of the expressions below have a *Set* type, and the last one, the quantified expression, has a *boolean* type. The examples are taken from our prototype, an extension of Java language.

i. Multi-Field Access

It is possible to get multiple fields of the object, provided the fields have the same type. The expression should return a *Set* containing the values of fields of the receiver object. The syntax may look like the following.

object.(field1+field2+...)

Such expression is simply desugared into a loop going through the mentioned fields and adding their values to a set only if they have *non-null* values. The example below gets the children of a node in a binary tree.

```
function public HashSet<Node> children() { this.(left+right) }
```

ii. Field Access over A Set

It's useful to access values of a particular field over a set of objects of the same type, effectively as a *map* function. The result is a set of the same cardinality. The symbol .< is adapted in the syntax shown below.

set.<field

We have used the syntax below to obtain the integer values of the children of a *Node* object.

```
function public Set<Integer> childrenValues() { children().<value }
```

iii. Transitive Closure of Fields

Closures enable compact specifications for recursive data types, such as linked lists and trees. Alloy syntax supports transitive closure and reflexive transitive closure of multiple fields, denoted with .^ and .* symbols respectively.

object.^(field1+field2+...)
object.*(field1+field2+...)

The desugared version starts with an empty result set, and an unexpanded set only consisting of the original object. It then adds all non-null values for the mentioned fields of this object to the unexpanded list and the result set, then removes the original object from the unexpanded set. The process is repeated on each object in the unexpanded set, until it is empty. Because data structures may be cyclic, there needs to be a *fixed point* test mechanism to stop the process when the resulting set has reached a fixed point. One example of transitive closure shows a method that gets all nodes descending from a node in a binary tree.

```
function public ESJSet<Node> descendants() { this.^(left+right) }
```

iv. Set Comprehension (Filtering)

It is common in specifications to make a statement about a qualified subset of elements in a set. *Set comprehensions* are common in relational expressions, and useful in making specifications more compact and readable. There are times when one qualifies over an explicitly given set (second syntax below), and other cases when the set is omitted and implied to be all instantiated object of the given type. It's therefore necessary for the compiler to automatically add a set field for classes to keep track of the instantiated objects and augment the *constructor* to add the new instance to this set.

<table>
<tr><td>{ type var | boolean expression }</td></tr>
<tr><td>{ type var : set | boolean expression }</td></tr>
</table>

In a red black tree where nodes have colors for example, it may be needed to define a function that returns the set of the black descendants of a node.

```
function public Set<Node> blackDescendants() {
        { Node n : descendants() | n.color == BLACK }
}
```

Note that when a primitive type like integer is being quantified over, one must pre-impose a lower and upper bound on the type to make the expression executable.

v. Quantified Expressions

Universal and *Existential* quantifications and a few of their minor siblings replace the need for loops to express boolean properties. The syntax is the same as in set comprehension, but without the curly braces and starting with one of the quantifiers: *all, some, no, one,* or *lone*. The last one implies *at most one* in Alloy language. The return type of these expressions is boolean. These may be nested.

<table>
<tr><td>all/some/no/one/lone type var : set | boolean expression</td></tr>
<tr><td>all/some/no/one/lone type var | boolean expression</td></tr>
</table>

As with previous syntactic sugars, these expressions are desugared into a call to a method that the compiler automatically instruments. Such method is simply a *for loop* over the set, returning true or false as soon as the value of quantified predicate is determined. With support for quantified expressions and transitive closure, acyclic property on a binary tree is easily specified.

```
function public boolean isAcyclic() { no Node n | n.^(left+right).contains(n) }
```

III. Annotations

The specifications appear as general invariants for a class, or post-conditions for select methods. For those methods that include specifications—*esnured methods*—the compiler automatically adds an assertion check after the body of the method. The asserted expression for each method will be the specified post-condition, in conjunction with any declared class invariants. This section shows the adapted syntax.

i. The e*nsures* Keyword

To support adding invariant specifications for a class and post-conditions for its methods, the syntax of class and method header definition is extended. Aside from regular boolean expressions, the *boolean expression* here may include a call to the high-level *functions* as defined in the previous section.

<class header> ***ensures*** *<boolean expression>{ ... }*
<method header> ***ensures*** *<boolean expression>{ ... }*

ii. The o*ld* Keyword

Post-conditions for methods often include relations between values before and after the body is executed. For instance, the specification of a *sort* method for a *List* class, declares that after being sorted, the list is a *permutation* of itself before the sorting. To support assertion check of such specifications, the compiler adds a reserved field named *old* when compiling classes. Whenever method specifications include reference to the *old* value of an object (meaning its value before the method was run), a *deep copy* of the object is made before the method body is executed.

IV. Translation to Logic

For the invariants and the pre- and post-conditions listed for a class and its methods to be executable, these expressions need to be translated into the syntax of the external tool that is chosen for extracting models. All of the syntactic sugars in the previous section are adapted from the the syntax of the Alloy language and the relational constraint solver it uses, and thus specifications written in our extended syntax, as well as common relations like algebraic comparisons, etc, are easily translated into a constraint problem understood by the external tool at compiled time. The run-time part of the translation, however, is harder. The *objects* and *fields* need to be mapped to *atoms* and *relations* for the relational constraint solver. Before specifications can be run, every object and its fields which are involved in the boolean expression need to be mapped. Two simple dictionaries may easily track the mappings from objects and fields in the object domain into atoms and relations in the logic domain, and vise versa. The latter mapping will be used when a solution is obtained and needs to be applied to the objects. The implementation of such translation is fairly straightforward. A few points are discussed in this section.

i. Class Fields

For each declared field in the class being compiled, a relation is constructed, say called *<class>.<field>*. When comes the time to execute some specification using the logic solver, the relations need to be filled in with the existing objects field values. These relations may be implemented as dictionaries of *[object, value]* pairs.

relation <class>.<field> = { [<object1, value1>], ... }

If object *A* of class *C*'s field *F* has a value *V*, then relation *C.F* includes the pair *[A,V]*. For collection type fields, the relation simply holds more than one pair: *[A,V1], [A,V2], ...* one for each element in the collection. It should be noted that usually not all constructed relations will be referenced by the specifications. One can provide a mechanism to track the relations referenced in the formula in the particular specification being called to narrow down the problem generated for the constraint solver.

ii. Field Access

A field get, *<object>.<field>* is translated to a *join* operation, denoted by . , in relational logic. The left of the binary operation would be the receiver object and the right side is the relation corresponding to the field of the class that object belongs to. The closure operations .* and .^ are translated the same manner.

<object>.(Relation <class>.<field>)

iii. Hard Coded Translations

Depending on the object oriented language and the constraint solvers used, the translations for the common library methods need to be handled at compile time. For instance, the Java *Set* method *contains(Object)* is mapped to *some <object> & <set>* (for non-empty intersection) in the relational logic solver used in our prototype.

iv. Translating Models Back to The Object World

Once the constraint solver finds a model of the formula provided by the specification, the model is applied to the objects by setting the values for fields as mentioned in the model relations. Only those relations that served as unknowns in the specified problem need to be applied.

V. A Red Black Tree Program

A *red black tree* was written with *insert* and *delete* operations in the Java language extended with the high-level function methods. The two operations are fully runnable without any implementation, only relying on the post-condition specifications accompanying them denoted by the *ensures* keyword, and an external relational constraint solver. The program is robust: the specifications guarantee that the RB Tree properties are never violated. This code is about 5X smaller (and more X understandable!) than a common implementation of RB trees.

ESJSet is simply a subclass of *HashSet* that includes functional methods, *plus(Object)* and *minus(Object)*, each returning a new set equal to the original one after adding/removing the given object to/from it.

A fully runnable red black tree program with insert and delete operations only relying on specifications

```
public class Node {
    public Integer value;
    public Color color;
    public Node left, right, parent;

    public Node(Integer value, Color color, Node left, Node right) {
        this.value = value;
        this.color = color;
        this.left = left;
        this.right = right;
```

```
        if (left != null) left.parent = this;
        if (right != null) right.parent = this;
    }

    public void value(Integer v) { this.value = v; }
    public void color(Color c) { this.color = c; }
    public void left(Node l) { this.left = l; }
    public void right(Node r) { this.right = r; }
    public void parent(Node p) { this.parent = p; }

    function public ESJSet<Node> children() { this.(left+right) }

    function public ESJSet<Node> descendants() { this.^(left+right) }

    function public ESJSet<Node> leftDescendants() { this.left.*(left+right) }

    function public ESJSet<Node> rightDescendants() { this.right.*(left+right) }

    function public ESJSet<Node> ancestors() { this.^parent }

    function public ESJSet<Node> thisAndAncestors() { this.*parent }

    function public ESJSet<Node> blackAncestors() {
      { Node n : thisAndAncestors() | n.color == Color.BLACK }
    }
}

public class RBTree ensures isRBTree() {
    public Node root;

    public void root(Node r) { this.root = r; }
    function public ESJSet<Node> nodes() { root.*(left+right) }

    function public ESJSet<Node> leaves() {
        { Node n : nodes() | (n.left == null || n.right == null) }
    }

    function public ESJSet<Integer> nodeValues() { this.nodes().<value }

    function public boolean isRBTree() {
      isBinarySearchTree() && rootBlack() && redsChildren() && eqBlacks()
    }


    function public boolean isBinarySearchTree() {
      isAcyclic() && parentDef() && oneParent() && isValidBinarySearch()
    }

    function public boolean rootBlack() {
        this.root == null || this.root.color == Color.BLACK
    }

    function public boolean redsChildren() {
      all Node n : nodes() | (n.color == Color.BLACK ||
                              all Node c : n.children() | c.color == Color.BLACK)
    }
```

```
    function public boolean eqBlacks() {
      all Node l1 : leaves() |
          all Node l2: leaves() |
              (l1 == l2 || l1.blackAncestors().size() == l2.blackAncestors().size())
    }

    function public boolean isAcyclic() {
      no Node n | (n.descendants().contains(n) || n.ancestors().contains(n))
    }

    function public boolean parentDef() {
      all Node n | all Node p | (!p.children().contains(n) || n.parent == p)
    }

    function public boolean oneParent() {
      (root == null) ?
           true :
           all Node n : this.root.descendants() | one Node p | n.parent == p
    }

    function public boolean isValidBinarySearch() {
       all Node n |
           ((n.left == null  || all Node lc : n.leftDescendants()  | lc.value < n.value) &&
            (n.right == null || all Node rc : n.rightDescendants() | rc.value > n.value))
    }

    public void insert(Integer value) {
       Node insertedNode = new Node(value, null, null);
       insert(insertedNode);
    }

    public void insert(Node insertedNode)
      modifies RBTree.root, Node.color, Node.left, Node.right, Node.parent
      ensures this.nodes().equals(this.old.nodes().plus(insertedNode)) {
      // no implementation...
    }

    public void delete(Integer value)
      modifies RBTree.root, Node.color, Node.left, Node.right, Node.parent
      ensures this.nodeValues().equals(this.old.nodeValues().minus(value)) {
      // no implementation...
    }
}
```