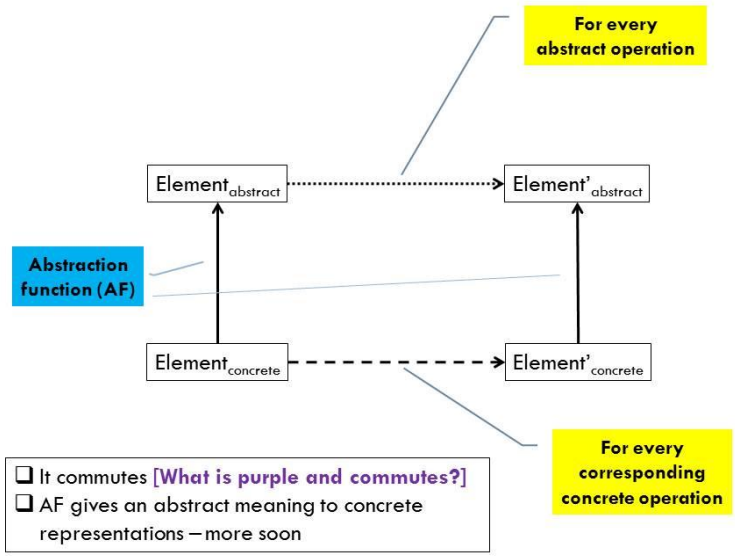
CSE 503 Winter 2013 • David Notkin

Lecture #4 Notes: proving ADTs, electrifying verification

1. Proving a procedure/method – earlier lecture. Define semantics of programming language constructs. Write down the pre- and post-condition – the specification – for the procedure. Show that the states defined by the pre-condition are transformed into (a logical formula that implies the) post-condition using the semantic definitions. Roughly.
2. Types
   a. Without getting precise, types are used to interpret and manipulate the bit patterns – that is, they give them (some level of) meaning
   b. "Concrete" types manipulate the information in memory directly
   c. Abstract types define a protocol for manipulating instances of those types, but they do not define an implementation
   d. Abstract data type = objects + operations
   e. The only operations on objects of the type are those provided by the abstraction
   f. The implementation is hidden
   g. We need to show that the abstraction and the implementation are each "correct" … and properly related … to be continued

2. Big picture



An Abelian grape (sorry)

3. Specifying ADTs

a. A common way is to define the abstract effect of each operation (including constructors) using formal/informal pre- and post-conditions
b. Might see this using an extended JavaDoc
c. Example

```
// Overview: An IntSet is a mutable, unbounded set of integers.

class IntSet {

 // effects: makes a new IntSet = {}
public IntSet()

// returns: true if x ∈ this
//          else returns false public boolean contains(int x)
// effects:  this_post = this_pre ∪ {x}

public void add(int x)
// effects:  this_post = this_pre - {x}

public void remove(int x)

…
```

4. Algebraic specifications -- From Stotts (http://www.cs.unc.edu/~stotts/723/adt.html)
   a. Define a *sort* – give signatures of operations (you've seen this kind of thing before in typed OO and functional languages)

```
sort IntSet imports Int, Bool
signatures
    new : -> IntSet
    insert : IntSet × Int -> IntSet
    member : IntSet × Int -> Bool
    remove : IntSet × Int -> IntSet
```

   b. Define axioms -- "Just" like high school algebra

```
variables i, j : Int; s : IntSet
axioms
    member(new(), i) = false
    member(insert(s, j), i) =
        if i = j then true else member(s, i)
    remove(new(), i) = new()
    remove(insert(s, j), i) =
        if i = j then remove(s, i)
                 else insert(remove(s, i), j)
```

c. Are these really sets?
    i. Posit stuff like…

```
insert(insert(s, i), j) =
insert(insert(s, j), i)
```

    ii. `insert(insert(s, i), i) = insert(s, i)`

d. Prove from axioms
e. Tons of issues about completeness, consistency, equality (initial vs. final algebras), etc.
f. But again, "just" like high school algebra

5. Proving specification properties
    a. Regardless of the style of specification, proofs are usually done inductively
    b. No information about the concrete representation and implementation – rather, showing the correctness of the protocol over the ADT's operations
    c. **LetterSet** case-insensitive character set [from Ernst]

```
// effects: creates an empty LetterSet
public LetterSet ( );

// effects: this_post =
//          if (∃ c_1 ∈ this_pre | toLowerCase(c_1) = toLowerCase(c)
//                      then this_pre else this_pre ∪ {c}
public void insert (char c);

// effects: this_post = this_pre − {c}
public void delete (char c);

// returns:  (c ∈ this)
public boolean member (char c);
```

In small groups, sketch a proof that a large enough **`LetterSet`** contains two distinct characters

6. Abstraction function and representation invariant
    a. Abstraction function **(AF): $E_c \rightarrow E_a$**
        i. Maps a concrete object to an abstract value
        ii. Defines how the data structure is to be interpreted
    b. Representation invariant **(RI)**: a boolean predicate characterizing legal concrete representations
        i. States data structure well-formedness -- in essence, defines the domain of **AF**
        ii. Captures information that must be shared across implementations of multiple operations

CharSet Abstraction

A finite mutable set of Characters [From Ernst]

```
// Overview: A CharSet is a finite mutable set of Character

// effects: creates a fresh, empty CharSet
public CharSet ( )

// effects: thispost = thispre ∪ {c}
public void insert (Character c);

// effects: thispost = thispre - {c}

public void delete (Character c);

// returns: (c ∈ this)
public boolean member (Character c);

// returns: cardinality of this
public int size ( );
```

A **CharSet** implementation

```
class CharSet {
    private List<Character> elts = new ArrayList<Character>();

public void insert(Character c) {
    elts.add(c);

 }

public void delete(Character c) {
    elts.remove(c);

}

public boolean member(Character c) {
    return elts.contains(c);
}
public int size() {
    return elts.size();
}
…
```

The RI can help identify an error

i. Perhaps **delete** is wrong -- It should remove all occurrences

ii. Perhaps **insert** is wrong -- it should not insert a character that is already there

```
class CharSet {
  // Rep invariant: elts has no nulls and no duplicates
  private List<Character> elts;
  …
```

Or…

a. ∀ **indices i of elts . elts.elementAt(i) ≠ null**
b. ∀ **indices i, j of elts . i ≠ j** ⇒
   ¬**elts.elementAt(i).equals(elts.elementAt(j))**

iii. Where's the error?

```
// Rep invariant: elts has no nulls and no duplicates
public void insert(Character c) {
    elts.add(c);
}

public void delete(Character c) {
    elts.remove(c);
}
```

The RI constrains structure, not meaning – but if the RI fails, it means that the ADT has no well-defined meaning

Another implementation of insert that *preserves the RI*

```
public void insert(Character c) {
    Character cc = new Character(encrypt(c));
    if (!elts.contains(cc))
        elts.addElement(cc);
}

public boolean member(Character c) {
    return elts.contains(c);
}
```

The program is wrong … call on the AF, abstraction function mapping concrete to abstract values

AF(CharSet this) = { c | c is contained in this.elts }

i. set of Characters represented by elements contained in **this.elts**
ii. Typically not executable, but useful to reason about client behavior

iii. Helps reason about the semantics of insert

```
// effects: this_post = this_pre ∪ {c}
public void insert (Character c);
```

iv. Helps identify a problem

a. Applying the AF to the result of the call to insert yields

AF(elts) ∪ {encrypt('a')}

v. Consider the following reasonable AF

a. **AF(this) = { c | encrypt(c) is contained in this.elts }**
b. **AF(this) = { decrypt(c) | c is contained in this.elts**

vi. "Placing blame" using AF

a. **AF(CharSet this) = { c | c is contained in this.elts }**
b. Consider a call to insert:

i. On entry, the meaning is **AF(this$_{pre}$) ≈ elts$_{pre}$**

ii. On exit, the meaning is **AF(this$_{post}$) = AF(this$_{pre}$) ∪ {encrypt('a')}**

vii. Does this AF fix things?

```
AF(this) = { c | encrypt(c) is contained in this.elts }
     = { decrypt(c) | c is contained in this.elts }
```

7. Looking at these examples using the commutative diagram may help clarify any confusions

a. AF's can be maintained across fairly complicated implementations that (for example) reorganize dynamically for performance -- Multiple concrete values still map to the same abstract value

b. Why map concrete to abstract?

i. It's not a function in the other direction

ii. Ex: lists [a,b] and [b,a] each represent the set {a, b}

iii. It's not as useful in the other direction

8. Electrifying formalisms

a. One thing is clear so far: programs *execute* – they are kinetic – while formalisms just sit there – they represent potential

b. The goal is the execution behaviors; the formalisms/programs are the conduit

c. This property makes many formalisms less attractive to many people, as the benefits are harder to see

d. **"**It is easier to change the specification to fit the program than vice versa**." –Perlis**

e. Daniel Jackson (and others) have worked on addressing this concern by “electrifying” formalisms – that is, making them “executable” in some sense, or at least providing useful feedback to a developer quickly

f. Alloy is Jackson’s core approach to this, but it’s not the only one out there – the objective, in some sense, is to put formalisms “in motion”, making them more alive and thus more interesting and perhaps of some value. Automated verification is real now, even with lots of problems, so electrification Is making progress.

g. One way to electrify a formalism is to execute it – many formalisms represent high-level programs

    i. Google Scholar found ~95K entries to “executable specifications”

    ii. Many such executable specifications look a lot like (various kinds of) logic programs or functional programs; much of this work is related to *automatic programming*

    iii. The execution gives insight into what the specification means

    iv. Performance of these “programs” is usually poor

    v. And automated refinement techniques to evolve from an executable specification to an efficient program seem to be limited

    vi. This work goes back to at least 1976 with Darlington, Burstall, Manna and others

h. Type checking is another form of electrification – quick feedback on common potential errors. Types are becoming increasing rich, too, adding power to this.

9. Model checking is one of the bases for electrifying comparisons of program views

    a. Basic inputs: finite state machine and temporal logic formula

    b. Basic outputs: “Yes” (satisfies), “No” (doesn’t satisfy, and here’s a counterexample that shows a path through the state machine that contradicts the temporal logic formula.

    c. Questions include

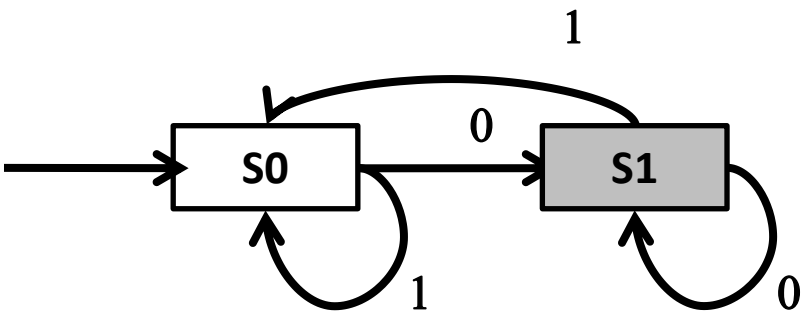
        i. What can the finite state machines and temporal logic formulae represent?

        ii. What does “satisfy” mean? How does “satisfy” work? Why should we care?

        iii. What is a counterexample?

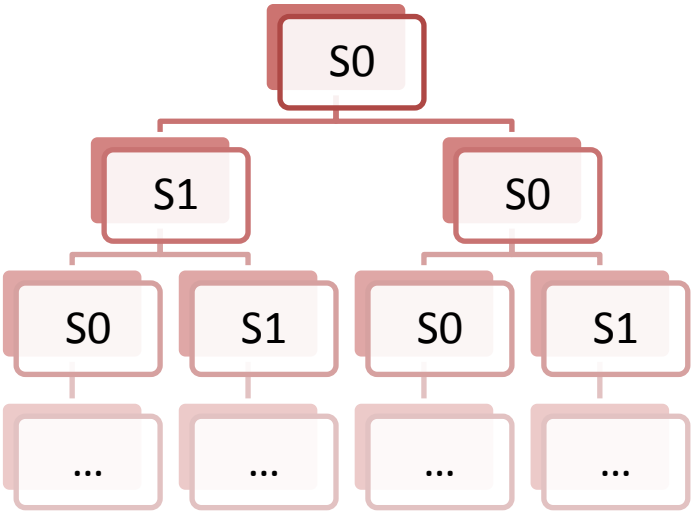
        iv. What does “Yes” actually mean?

10. State machine



a. Generate computation tree from state machine to represent all possible FSM paths.

b. Even though it's an FSM, there may be infinite paths; but they will have structure because they are generated from the FSM.



c. Then apply the temporal logic formula to the computation tree

d. Model checking answers questions about this tree structure – kinds of queries include

   i. Does every accepting input include a **0**? A **1**?

   ii. Does any accepting input include a **0**? A **1**?

   iii. Does every accepting input that has a **1** have a **1** in the remaining input?

   iv. More generally, safety and liveness properties of many kinds

11. FAQ
    a. What can the finite state machines and temporal logic formulae represent?
        i. Lot's!
    b. What does "satisfy" mean? How does "satisfy" work?
        i. Satisfy means that the temporal logic formula is guaranteed to hold over the computation tree defined by the FSM
    c. Why should we care?
        i. Guarantees can be a good thing
    d. What is a counterexample?
        i. A path through the computation tree that contradicts the temporal logic formula
        ii. There is a mismatch between the two descriptions – but one cannot tell which is "wrong" without further work
    e. What does "Yes" actually mean?
        i. It is a guarantee that the property holds, but it provides no guarantee that the property or the FSM are what the developers thinks they are
12. Two kinds of model checkers
    a. Explicit – represent all states
        i. Use conventional state-space search
        ii. Reduce state space by folding equivalent states together
    b. Symbolic – represent sets of states using boolean formulae
        i. Reduce huge state spaces by considering large sets of states simultaneously – to the first order, this is the meeting of BDDs (binary decision diagrams) and model checking (more later)
        ii. Convert state machines, logic formulae, etc. to boolean representations
        iii. Perform state space exploration using boolean operators to perform set operations
        iv. SAT solvers are often at the base of symbolic model checking