CSE503: Software Engineering

David Notkin
University of Washington
Department of Computer Science & Engineering
Winter 2002

Abstract data types

- Abstract data types (ADTs) are a common foundation for software development
  - They grew out of Parnas' notion of information hiding, which we'll cover during our design lectures
  - Very roughly, an encapsulated type or a class: a set of procedures (methods) that are the only way to access and manipulate encapsulated data
- ADTs are commonly specified by
  - Natural language comments associated with
  - Signatures of the procedures; for example,
  - `void copyIntBuf(int *pin,int *pout,int len)`

Algebraic specifications

- Algebraic specifications provide a mathematical framework for specifying ADTs
- The intent is to provide clear and well-defined semantics for the operations (procedures), rather than depending on natural language associated with precisely defined syntax

Algebras: roughly

- A set of objects
- A set of rules, called axioms, for determining the equality among those objects
- "K-12" algebra
  - Set of objects is the real numbers
  - $x*(y+z) = x*y + x*z$
  - $x+y=y+x$
  - …

Algebraic specification for ADT

1. The name of the *sort* (roughly, the type) being specified
2. The signatures of the primitive operations
3. The axioms

- There are a number of languages that support algebraic specification, including Anna, Clear, Larch, OBJ, …

Sort

- A sort is a set of values
  - roughly a "type" or "class"
  - Ex: integers, stacks of integers, strings, complex numbers, …
- The *sort of interest* is the one that is being defined by a particular specification
- To define this specification may require other sorts (we'll see an example)
- This approach induces a hierarchy of sorts

Signatures

- The name of the operator
- The types of its parameters
- The return type

- Like programming language signatures, but usually represented more abstractly
  - `push: Stack x Elem -> Stack`
  - `+: Integer x Integer -> Integer`
  - `Round: Real -> Integer`
- May look semi-familiar to those who studied ML in 505

Axioms

- Rules that must hold true in any legal implementation of the sort

Example: queue

- Signature
  - `create: -> Queue`
  - `add: Queue x Element -> Queue`
  - `remove: Queue -> Queue`
  - `front: Queue -> Element`
- Axioms

```
– front(add(create(),x)) = x
– front(add(add(q,x),y)) = front(add(q,x))
– remove(add(create(),x)) = create()
– remove(add(add(q,x),y)) =
                  add(remove(add(q,x)),y)
```

Conditional axioms

```
front(add(q,i)) =
   if (IsEmpty(q))then i
   else front(q);
```

- In some cases (not necessarily this one) one can increase the clarity with conditional axioms

Operations

- Usually separated into
  - Constructors (that create an instance of the sort)
  - Accessors (that take an instance of the sort as a parameter and return an element from a supporting sort)
  - Modifiers (that take an instance of the sort as a parameter and return a modified instance of it)

Issues

- Equality: two elements in a sort are equal if and only if all operations applied to them produce equal results
  - Closely related to the rewriting in the lambda-calculus
  - Inequality is defined as the inability to prove equality
- Consistency?
  - Roughly, can we show that the axioms cannot be used to prove “false”?
- Completeness?
  - Roughly, does it represent all the values (e.g., queues) that we intended?

Another example: signatures

```
algebra StringSpec;
  sorts String, Char, Nat, Bool;
  operations
    new: () -> String
    append: String, String -> String
    add: String, Char -> String
    length: String -> Nat
    isEmpty: String -> Bool
    equal: String, String -> Bool
```

```
StringSpec generated by [new, add]
for all [s1, s2, s3: String; c: Char]

isEmpty (new()) = true;
isEmpty (add(s1,c)) = false;
length (new()) = 0;
length (add(s1,c)) = length (s1) + 1
append (s1, new()) = s1
append (s1, add(s2,c)) = add
          (append(s1,s2), c)
equal (new(), new()) = true
equal (new(), add(s1,c)) = false
equal (add(s1,c), new()) = false
equal (add(s1,c), add(s2,c)) = equal(s1,s2)
```

Pros of algebraic specifications

- Language independent
- Implementation independent
- Nicely matched to ADTs
- Strong mathematical foundation
- Suited to automation of the underlying theorem proving
- Can “electrify” the specifications by tracing rewriting

Cons of algebraic specifications

- Difficult to deal with procedures that have side effects, reference parameters, multiple returns, etc.
- Not all interesting behaviors are expressed via equality
- The limits of notation can lead to messy and complicated specifications