Formal Specification of Abstract Datatypes

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.uni-linz.ac.at

Datatypes

What is a datatype?

- Traditional view: collection of data with same structure.
  - Mathematics:

    set $S := int \times char = \{(a,b) \mid a \in int \wedge b \in char\}$.
  - Programming:

    struct S $\{int\ a; char\ b\}$
- Modern view: collection of data with same services.
  - Mathematics

    $$\begin{aligned} \text{algebra } T &= (S,\ getA : S \to int,\ getB : S \to char) \\ &= (int \times char, \lambda(a,b).a, \lambda(a,b).b). \end{aligned}$$
  - Programming:

    class T $\{$ S x;
    $int\ getA()$ $\{$**return** $x.a\}$; $char\ getB()$ $\{$**return** $x.b\}$ $\}$.

In this course, we will take the modern view of datatypes.

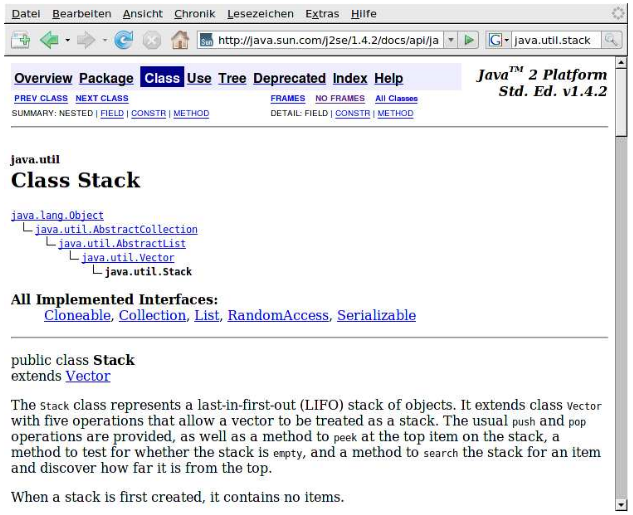
Abstract Datatypes

What is an abstract datatype (ADT)?

- The set of services to be provided by an implementing datatype.
  - The description of the services is the specification of the ADT.
    - The specification does not enforce a particular data representation.
  - A datatype providing such services is an implementation of the ADT.
    - Provides concrete data representations for the values of the ADT.
    - Provides concrete program methods for the services of the ADT.
- There may be zero, one, many implementations of an ADT possible.
  - The specification of the ADT should be as general as possible in order not to constrain the implementation more than necessary.
- The specification is the contract between user and implementer.
  - "Design by contract" (Bertrand Meyer).

Thus we need specification languages to describe ADTs.

Java API Documentation


Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
http://java.sun.com/j2se/1.4.2/docs/api/ja
java.util.stack
Overview Package Class Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS
FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD

Java™ 2 Platform
Std. Ed. v1.4.2

java.util

Class Stack

```
java.lang.Object
  └ java.util.AbstractCollection
      └ java.util.AbstractList
          └ java.util.Vector
              └ java.util.Stack
```

All Implemented Interfaces:
Cloneable, Collection, List, RandomAccess, Serializable

public class **Stack**
extends Vector

The `Stack` class represents a last-in-first-out (LIFO) stack of objects. It extends class `Vector` with five operations that allow a vector to be treated as a stack. The usual `push` and `pop` operations are provided, as well as a method to `peek` at the top item on the stack, a method to test for whether the stack is `empty`, and a method to `search` the stack for an item and discover how far it is from the top.

When a stack is first created, it contains no items.

Java API Documentation

```
public Object push(Object item)
    Pushes an item onto the top of this stack.

    Parameters:
        item - the item to be pushed onto this stack.
    Returns:
        the item argument.

public Object pop()
    Removes the object at the top of this stack and returns that object
    as the value of this function.

    Returns:
        The object at the top of this stack.
    Throws:
        EmptyStackException - if this stack is empty.
```

Java Interfaces

```
interface StackADT
{
  // Pushes an item onto the top of this stack.
  // Returns the item pushed on the stack.
  Object push(Object item);

  // Removes the object at the top of this stack and
  // returns that object as the value of this function.
  // Throws EmptyStackException, if this stack is empty.
  Object pop();

  // Returns the object at the top of this stack
  // without removing it from the stack.
  // Throws EmptyStackException, if this stack is empty.
  Object peek();

  // Returns true if and only if this stack contains no items.
  boolean empty();
}
```

Specification Languages

Programming languages only describe the syntax (interface) of an ADT.

- Specification languages also describe the semantics (behavior).
  - Based on concepts from universal algebra and logic.
  - Notions “datatype” and “ADT” have a precise meaning.
    - An algebra T and a (particular) class $\mathcal{A}$ of algebras, respectively.
  - Statement “datatype T implements ADT $\mathcal{A}$” has a precise meaning.
    - $T \in \mathcal{A}$.
    - Formal calculus to prove the statement.
- Constructive specifications may be even executed.
  - Describe not only requirements but also suggest an implementation.
  - Term rewriting engines for executing constructive specifications.
  - Rapid prototyping of specifications in the design phase.

Formal specifications can overcome the ambiguity of natural language when describing program requirements.

Larch

```
Stack (E, C): trait
  introduces
    empty: -> C
    push: E, C -> C
    top: C -> E
    pop: C -> C
    isEmpty: C -> Bool
  asserts
    C generated by empty, push
    forall e: E, stk: C
      top(push(e, stk)) == e;
      pop(push(e, stk)) == stk;
      isEmpty(stk) == stk = empty
```

Formal description of ADT “Stack” in the Larch Shared Language (LSL).

Larch/C++

```
template <class Elem                           virtual Stack<Elem>& pop() throw();
/*@ expects contained_objects(Elem) @*/ >      //@ behavior {
class Stack {                                  //@ requires ~isEmpty(self^);
public:                                        //@ modifies self;
                                               //@ ensures self' =
//@ uses Stack(Elem for E,                     //@   pop(self^) /\ result = self; }
               Stack<Elem> for C);
                                               virtual Elem top() const throw();
Stack() throw();                               //@ behavior {
//@ behavior {                                 //@ requires ~isEmpty(self\any);
//@ modifies self;                             //@ ensures result = top(self\any); }
//@ ensures liberally self' = empty; }
                                               virtual bool isEmpty() const throw();
virtual Stack<Elem>& push(Elem e) throw();     //@ behavior {
//@ behavior {                                 //@ ensures result =
//@ modifies self;                             //@   (isEmpty(self\any)); }
//@ ensures liberally self' =
//@   push(self^,e) /\ result = self; }        };
```

Formal specification of a C++ "Stack" in Larch/C++.

CafeOBJ

```
dragonfly!1> /zvol/formal/bin/cafeobj
-- loading standard prelude
; Loading /usr3/cafeobj-1.4/prelude/std.bin

          -- CafeOBJ system Version 1.4.6(PigNose0.99,p3) --
                   built: 2004 Nov 17 Wed 6:37:33 GMT
                         prelude file: std.bin
                                  ***
                     2005 Sep 10 Sat 12:39:32 GMT
                            Type ? for help
                                  ***
                   -- Containing PigNose Extensions --
                                  ---
         built on International Allegro CL Enterprise Edition
                6.2 [Linux (x86)] (Nov 17, 2004 15:37)
CafeOBJ>
```

System for executing constructive specifications.

CafeOBJ

```
CafeOBJ> module! STACK
{
  protecting (NAT)
  signature
  {
    [ Stack ]
    op empty : -> Stack
    op push : Nat Stack -> Stack
    op top : Stack -> Nat
    op pop : Stack -> Stack
  }
  axioms
  {
    var N : Nat
    var S : Stack

    eq top(push(N, S)) = N .
    eq pop(push(N, S)) = S .
  }
}
```

CafeOBJ

```
CafeOBJ> open STACK
-- opening module STACK.. done.
%STACK> parse top(push(1, empty)) .
top(push(1,empty)) : Nat
%STACK> reduce top(push(1, empty)) .
-- reduce in %STACK : top(push(1,empty))
1 : NzNat
(0.000 sec for parse, 1 rewrites(0.000 sec), 1 matches)
%STACK> parse top(pop(push(2, push(1, empty)))) .
top(pop(push(2,push(1,empty)))) : Nat
%STACK> reduce top(pop(push(2, push(1, empty)))) .
-- reduce in %STACK : top(pop(push(2,push(1,empty))))
1 : NzNat
(0.000 sec for parse, 2 rewrites(0.000 sec), 2 matches)
%STACK> parse top(pop(push(1, empty))) .
top(pop(push(1,empty))) : Nat
%STACK> reduce  top(pop(push(1, empty))) .
-- reduce in %STACK : top(pop(push(1,empty)))
top(empty) : Nat
(0.000 sec for parse, 1 rewrites(0.000 sec), 2 matches)
%STACK> close
```

Algebraic/Axiomatic Specifications

- Approach rooted in universal algebra.
  - Logical axioms relate different operations of ADT to each other.
  - Similar as in the description of algebras in mathematics.
- Original focus (1970s/1980s): initial semantics.
  - Specifications in (conditional) equational logic.
    - Main interest in executable design specifications.
    - Strong connections to term rewriting.
  - Languages: Clear, ACT ONE/TWO, OBJ family, . . .
- Alternative focus (1990s): loose semantics.
  - Specifications in full first-order predicate logic.
    - Main interest in precise requirement specifications.
    - Strong connections to object-oriented program specification.
  - Languages: Larch/C++, Java Modeling Language (JML), . . .
- Common Algebraic Specification Language (CASL)
  - Result of Common Framework Initiative (CoFI), since 1995.
  - Unifying framework for algebraic specifications in different logics.

Course Outline

- Abstract Datatypes.
- *CafeOBJ*.
- Logic.
- Loose Specifications.
- *Larch/C++, JML*.
- Term Algebras.
- Initial Specifications.
- Specifications in the Large.
- *CASL*.

Interspersed with presentations of various case studies; exercises both theoretical (paper and pencil) and practical (CafeOBJ).