CSE 3302

Programming Languages

Abstract Data Types and Modules

Chengkai Li, Weimin He

Spring 2008

Data Types

- Predefined
- Type constructors: build new data types
- How to provide "queue"?
  - What should be the data values?
  - What should be the operations?
  - How to implement (data representation, operations)?

What are inadequate here?

- **The operations are not associated with the data type**
  - You can use the operation on an invalid value.
- **Users see all the details:**

  direct access to date elements, implementations
  - Implementation dependent
  - Users can even mess up with things

What do we want?

- For basic types:
  - 4 bytes or 2 bytes, users don't need to know.
  - Can only use predefined operations.
- Similarly, for the "Queue" data type:

  ?

Abstract Data Type

- Encapsulation:

  all definitions of allowed operations for a data type in one place.
- Information Hiding:

  separation of implementation details from definitions. Hide the details .

Algebraic Specification of ADT

- Syntactic specification (signature, interface):

  the name of the type, the prototype of the operations
- Semantic specification (axioms, implementation):

  guide for required properties in implementation

  mathematical properties of the operations

They don't specify:

- data representation
- implementation details

Syntactic Specification

```
type queue(element) imports boolean
operations:
   createq:    queue
   enqueue:    queue × element → queue
   dequeue:    queue → queue
   frontq:     queue → element
   emptyq:     queue → boolean
```

- `imports:` the definition queue needs `boolean`
- Parameterized data type (`element`)
- `createq:` not a function, or viewed as a function with no parameter

Algebraic Specification

```
variables: q: queue; x: element
axioms:
   emptyq(createq)          = true
   emptyq(enqueue(q,x))     = false
   frontq(createq)          = error
   frontq(enqueue(q,x))     = if emptyq(q) then x
                              else frontq(q)
   dequeue(createq)         = error
   dequeue(enqueue(q,x))    = if emptyq(q) then q
                              else enqueue(dequeue(q),x)
```

- error axiom (exceptions)

Stack

```
type stack(element) imports boolean
operations:
   createstk : stack
   push      : stack × element → stack
   pop       : stack → stack
   top       : stack → element
   emptystk  : stack → boolean

variables: s: stack; x: element
axioms:
   emptystk(createstk)     = true
   emptystk(push(s,x))     = false
   top(createstk)          = error
   top(push(s,x))          = x
   pop(createstk)          = error
   pop(push(s,x))          = s
```

Axioms

- How many axioms are sufficient for proving all necessary properties?

Some Heuristics

```
type stack(element) imports boolean
operations:
   createstk : stack
   push      : stack × element → stack
   pop       : stack → stack
   top       : stack → element
   emptystk  : stack → boolean

variables: s: stack; x: element
axioms:
   emptystk(createstk)     = true
   emptystk(push(s,x))     = false
   top(createstk)          = error
   top(push(s,x))          = x
   pop(createstk)          = error
   pop(push(s,x))          = s
```

Constructor:
createstk
push

Inspector:
pop
top
emptystk

2 * 3 = 6 rules

Binary Search Tree

```
type BST(element) imports boolean, int
operations:
   createbst   : BST
   emptybst    : BST → boolean
   insert      : BST × element → BST
   delete      : BST × element → BST
   getRoot     : BST → element
   getHeight   : BST → int
   max         : BST → element
   search      : BST × element → boolean

variables: t: bst; x: element
axioms:
   emptystk(createbst)       = true
   …
```

Other Examples of ADT

- Stack
- Queue
- Tree
- Set
- Map
- Vector
- List
- Priority Queue
- Graph
- ...

ADT Mechanisms

- Specific ADT mechanisms
  - ML abstype
- General module mechanism : not just about a single data type and its operations
  - Separate compilation and name control:
    C, C++, Java
  - Ada, ML
- Class in OO languages

ML Abstype

```
abstype 'element queue = Q of 'element list
with
    val creatq              = Q [],
    fun enqueue(Q lis, elem) = Q(lis @ [elem]),
    fun dequeue(Q lis)       = Q(tl lis),
    fun frontq(Q lis)        = hd lis,
    fun emptyq(Q [])         = true |
        emptyq(Q(h::t))      = false,
end;

type 'a queue
val creatq = - : 'a queue
val enqueue = fn : 'a queue * 'a -> 'a queue
val dequeue = fn : 'a queue -> 'a queue
val frontq = fn : 'a queue -> 'a
val emptyq = fn : 'a queue -> bool

- val q = enqueue(creatq,1);
val q = - : int queue
```


Modules

- **Module**: A program unit with a public interface and a private implementation; all services that are available from a module are described in its public interface and are exported to other modules, and all services that are needed by a module must be imported from other modules.
- In addition to ADT, module supports structuring of large programs:
  Separate compilation and name control

C: Separate Compilation

- queue.h: header file

```
#ifdef QUEUE_H
#define QUEUE_H

struct Queuerep;
typedef struct Queuerep * Queue;
Queue creatq(void);
Queue enqueue(Queue q, void* elem);
void* frontq(Queue q);
Queue dequeue(Queue q);
int emptyq(Queue q);

#endif
```

Incomplete type: Separate implementation

Simulate: Parametric polymorphism

C: Separate Compilation

- queue.c: queue implementation

```
#include "queue.h"

struct Queuerep
{ void* data;
  Queue next;
};

Queue creatq(void)
{ return 0;
}

void* frontq(Queue q)
{ return q->next->data;
}
...
```