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 createq                = 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 createq = - : '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(createq,3);
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 createq(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 Parameteric polymorphism

C: Separate Compilation

- `queue.c`: queue implementation

```
#include "queue.h"

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

Queue createq(void)
{ return 0;
}

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

C: Separate Compilation

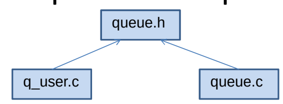
- `q_user.C`: client code

```
#include "queue.h"

int *x = malloc(sizeof(int));
int *y = malloc(sizeof(int));
int *z;
*x =2;
*y =3;

Queue q = createq();
q = enqueue(q,x);
q = enqueue(q,y);
q = dequeue(q);
z = (int*) frontq(q);
```


C: Separate Compilation



- Not real ADT
  - casting, allocation: for parametric polymorphism
  - header file directly incorporated into q_user.c: definition / usage consistent
  - data not protected: user may manipulate the type value in arbitrary ways
  - The language itself doesn't help in tracking changes and managing compilation/linking: thus tools like `make`

C++: Namespaces

- queue.h:

```
#ifdef QUEUE_H
#define QUEUE_H
namespace MyQueue
{
  struct Queuerep;
  typedef struct Queuerep * Queue;
  Queue createq(void);
  ...
}
#endif
```

- queue.c:

```
#include "queue.h"

struct MyQueue::Queuerep
{ void* data;
  Queue next;
};
...
```


C++: Namespaces

- q_user.cpp:

```
#include "queue.h"

using std::endl;
using namespace MyQueue;
main(){
  int *x = malloc(sizeof(int));
  int *y = malloc(sizeof(int));
  int *z;
  *x =2;
  *y =3;
  Queue q = MyQueue::createq();
  q = enqueue(q,x);
  q = enqueue(q,y);
  q = dequeue(q);
  z = (int*) frontq(q);
  …
}
```


Java: Packages

Queue.java:

```
package queues.myqueue;
…
```

PQueue.java:

```
package queues.myqueue;
…
```

directory: queues/myqueue

class files: Queue.class, PQueue.class

Q_user.java:

```
import queues.myqueue.Queue;
import queues.myqueue.*;
queues.myqueue.Queue;
```

queues/myqueue in CLASSPATH

Example

- Package java.util
  http://java.sun.com/j2se/1.5.0/docs/api/java/util/package-summary.html
- Interface Collection
  http://java.sun.com/j2se/1.5.0/docs/api/java/util/Collection.html
- Class PriorityQueue
  http://java.sun.com/j2se/1.5.0/docs/api/java/util/PriorityQueue.html

Ada: Packages

- **Package Specification**

```
generic
  type T is private;
package Queues is
  type Queue is private;
  function createq return Queue;
  function enqueue(q:Queue;elem:T) return Queue;
  function frontq(q:Queue) return T;
  function dequeue(q:Queue) return Queue;
  function emptyq(q:Queue) return Boolean;
private
  type Queuerep;
  type Queue is access Queuerep;
end Queues;
```

parameterized package: Parametric polymorphism

Prevents direct access

Pointers: Hide implementation details. Just making Queue incomplete won't work.

Ada: Packages

- **Package Body**

```
package body Queues is
  type Queuerep is
  record
      data: T;
      next: Queue;
  end record;

  function createq return Queue is
  begin
      return null;
  end createq;
  ...
end Queues;
```


Ada: Packages

- **User Code**

```
with Queues;

procedure Quser is
  package IntQueues is new Queues(Integer);
  use IntQueues;
  package FloatQueues is new Queues(Float);
  use FloatQueues;

  iq: IntQueues.Queue := createq;
  fq: FloatQueues.Queue := createq;
begin
  iq := enqueue(iq,3);
  fq := enqueue(fq,3.5);
end Quser;
```

Import packages: Specify dependency

Parametric polymorphism

Overloading

ML: Modules

- Signature (interface)

```
singature QUEUE =
  sig
  type 'a Queue
  val createq: 'a Queue
  val enqueue: 'a Queue * 'a -> 'a Queue
  val dequeue: 'a Queue -> 'a Queue
  val frontq: 'a Queue -> 'a
  val emptyq: 'a Queue -> bool
  end;
```

Parametric polymorphism

ML: Modules

- Structure (implementation)

```
structure Queue1: QUEUE =
        struct
        datatype 'a Queue = Q of 'a list
        val createq                = 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;
```


ML: Modules

- Use the queue

```
- val q = Queue1.enqueue(Queue1.createq,3);
val q = Q [3] : int Queue1.Queue
Queue1.frontq q;
val it = 3 : int
- val q1 = Queue1.dequeue q;
val q1 = Q [] : int Queue1.Queue
- Queue1.emptyq q1;
val it = true : bool
```

Must refer to implementation

ML: Modules

-open Queue1;

```
- val q = enqueue(createq,3);
val q = Q [3] : int Queue
- frontq q;
val it = 3 : int
- val q1 = dequeue q;
val q1 = Q [] : int Queue
- emptyq q1;
val it = true : bool
```

Without qualification