Data Abstraction and Modularity

John Mitchell

Reading: Sections 9.1, 9.2 (except 9.2.5), and 9.3.1

Topics

- Modularity
  - Interface, specification, and implementation
- Modular program development
  - Step-wise refinement; Prototyping; ...
- Language support for modularity
  - Procedural abstraction
  - Abstract data types
    - Representation independence
    - Datatype induction
  - Packages and modules
    - Generic abstractions
- Functions and modules with type parameters

Modularity: Basic Concepts

- Component
  - Meaningful program unit
    - Function, data structure, module, ...
- Interface
  - Types and operations defined within a component that are visible outside the component
- Specification
  - Intended behavior of component, expressed as property observable through interface
- Implementation
  - Data structures and functions inside component

Example: Function Component

- Component
  - Function to compute square root
- Interface
  - float sqroot (float x)
- Specification
  - If x>1, then sqrt(x)*sqrt(x)  x.
- Implementation
  
```c
float sqroot (float x){
    float y = x/2; float step=x/4; int i;
    for (i=0; i<20; i++) {
        if ((y*y)<x)
            y=y+step;
        else
            y=y-step;
        step = step/2;
    }
    return y;
}
```

Example: Data Type

- Component
  - Priority queue: data structure that returns elements in order of decreasing priority
- Interface
  - Type pq
  - Operations empty : pq
    - insert : elt * pq → pq
    - deletemax : pq → elt * pq
- Specification
  - Insert add to set of stored elements
  - Deletemax returns max elt and pq of remaining elts

Philosophy

- Build reusable program components
- Construct systems by divide-and-conquer
  - Limit interactions between components
  - Each component is assumed to satisfy spec
    - If another component satisfies the same specification, you can replace the first by the second
    - Internal improvements only improve the overall system, not break it
Example program using component

- Priority queue: structure with three operations
  - empty : pq
  - insert : elt * pq → pq
  - deletemax : pq → elt * pq
- Sorting algorithm using priority queue
  - begin
    - create empty pq
    - insert each element from array into s
    - remove elements in decreasing order and place in array
  - end
  - This gives us an $O(n \log n)$ sorting algorithm (HW 7)

Modular program design

- Top-down design
  - Begin with main tasks, successively refine
- Bottom-up design
  - Implement basic concepts, then combine
- Prototyping
  - Build coarse approximation of entire system
  - Successively add functionality

Stepwise Refinement

- Wirth, 1971
  - “… program … gradually developed in a sequence of refinement steps
  - In each step, instructions … are decomposed into more detailed instructions.
- Historical reading on web (CS242 Reading page)
  - N. Wirth, Program development by stepwise refinement, Communications of the ACM, 1971
  - D. Parnas, On the criteria to be used in decomposing systems into modules, Comm ACM, 1972
  - Both ACM Classics of the Month

Dijkstra’s Example (1969)

```
begin
  print first 1000 primes
end
```

```
begin
  int array p[1:1000]
  make for k from 1 to 1000
    p[k] equal to k-th prime
  print p[k] for k from 1 to 1000
end
```

Program Structure

```
Main Program
  ├── Sub-program
  │    └── Sub-program
  ├── Sub-program
  └── Sub-program
```
Data Refinement

• Wirth, 1971 again:
  – As tasks are refined, so the data may have to be refined, decomposed, or structured, and it is natural to refine program and data specifications in parallel

Example

Bank Transactions
  - Deposit
  - Withdraw
  - Print Statement
  - Print transaction history

• For level 2, represent account balance by integer variable
• For level 3, need to maintain list of past transactions

Language support for modularity

• Interface definition
  – Interface may consist of types, functions, subtype relationships, other language concepts exposed to other modules
• Isolation
  – Restrict dependence to factors visible through explicitly defined interface

Examples

• Procedural abstraction
  – Hide functionality in procedure or function
• Data abstraction
  – Hide decision about representation of data structure and implementation of operations
  – Example: priority queue can be binary search tree or partially-sorted array

Abstract Data Types

• Prominent language development of 1970’s
• Main ideas:
  – Separate interface from implementation
  – Example:
    – Sets have empty, insert, union, is_member?, ...
    – Sets implemented as linked list ...
  – Use type checking to enforce separation
  – Client program only has access to operations in interface
  – Implementation encapsulated inside ADT construct

ML Abstype

• Declare new type with values and operations
  abstype t = <tag> of <type>  
  with  
  val <pattern> = <body>  
  ...  
  fun f(<pattern>) = <body>  
  ...  
  end  
• Representation
  t = <tag> of <type>  
  similar to ML datatype decl
Abtype for Complex Numbers

- **Input**
  - abstype complex = C of real * real with
    - fun complex(x, y: real) = C(x, y)
    - fun x_coord(C(x, y)) = x
    - fun y_coord(C(x, y)) = y
    - fun add(Complex(x1, y1), Complex(x2, y2)) = Complex(x1+x2, y1+y2)
- **Types** (compiler output)
  - type complex
    - val complex = fn real * real -> complex
    - val x_coord = fn complex -> real
    - val y_coord = fn complex -> real
    - val add = fn complex * complex -> complex

Abtype for finite sets

- **Declaration**
  - abstype 'a set = SET of 'a list with
    - val empty = SET([], nil)
    - fun insert(x: 'a * 'a set) = SET(x, 'a set)
    - fun union(x: 'a set * 'a set) = 'a set
    - fun isMember(x: 'a, SET(elements)) = ...
- **Types** (compiler output)
  - type 'a set
    - val empty = nil : 'a set
    - val insert(x: 'a * 'a set) = 'a set
    - val union(x: 'a set * 'a set) = 'a set
    - val isMember(x: 'a * 'a set) = bool

Origin of Abstract Data Types

- **Structured programming, data refinement**
  - Write program assuming some desired operations
  - Later implement those operations
  - Example:
    - Write expression parser assuming a symbol table
    - Later implement symbol table data structure
- **Research on extensible languages**
  - What are essential properties of built-in types?
  - Try to provide equivalent user-defined types
  - Example:
    - ML sufficient to define list type that is same as built-in lists

Comparison with built-in types

- **Example: int**
  - Can declare variables of this type x: int
  - Specific set of built-in operations +, -, *, ...
  - No other operations can be applied to integer values
- **Similar properties desired for abstract types**
  - Can declare variables x: abstract_type
  - Define a set of operations (give interface)
  - Language guarantees that only these operations can be applied to values of abstract_type

Modules

- **General construct for information hiding**
- **Two parts**
  - Interface:
    - A set of names and their types
  - Implementation:
    - Declaration for every entry in the interface
    - Additional declarations that are hidden
- **Examples**
  - Modula modules, Ada packages, ML structures, ...

Modules and Data Abstraction

<table>
<thead>
<tr>
<th>Module</th>
<th>Can define ADT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>Private type</td>
</tr>
<tr>
<td></td>
<td>Public operations</td>
</tr>
<tr>
<td></td>
<td>More general</td>
</tr>
<tr>
<td></td>
<td>Several related types and operations</td>
</tr>
<tr>
<td></td>
<td>Some languages provide separate interface and implementation</td>
</tr>
<tr>
<td></td>
<td>One interface can have multiple implementations</td>
</tr>
</tbody>
</table>
Haskell modules

- Hide and selectively export declarations

```haskell
module Tree (Tree(Leaf,Branch), fringe) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)

fringe :: Tree a → [a]
fringe (Leaf x) = [x]
fringe (Branch left right) = fringe left ++ fringe right
```

Basic description: http://www.haskell.org/tutorial/modules.html

Generic Abstractions

- Parameterize modules by types, other modules
- Create general implementations
  - Can be instantiated in many ways
- Language examples:
  - Ada generic packages, C++ templates, ML functors, ...
  - ML geometry modules in course reader
  - C++ Standard Template Library (STL) provides extensive examples

Summary

- Modularity
  - Interface, specification, and implementation
- Modular program development
  - Step-wise refinement; Prototyping; ...
- Language support for modularity
  - Procedural abstraction
  - Abstract data types
    - Representation independence
  - Datatype induction
  - Packages and modules
  - Generic abstractions
  - Functions and modules with type parameters
- Modularity is supported by object-oriented languages, but did not originate with OOP