

Data Abstraction and Modularity

John Mitchell

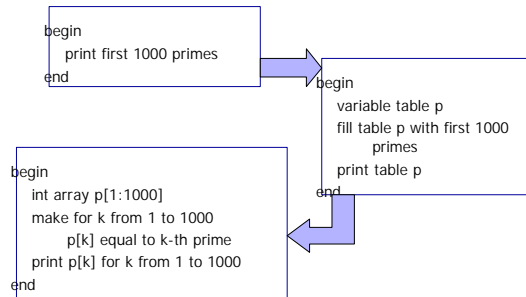
Topics

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

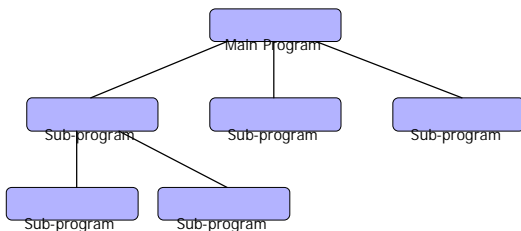
Stepwise Refinement

- ◆ Wirth, 1971
 - "... program ... gradually developed in a sequence of refinement steps"
 - In each step, instructions ... are decomposed into more detailed instructions.

Dijkstra's Example (1969)



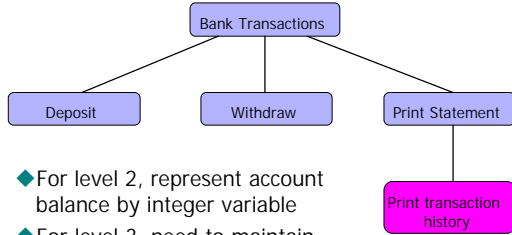
Program Structure



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



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

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

Modularity

- ◆ 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 sqrt(float x)`
- ◆ Specification
 - If $x > 1$, then $\text{sqrt}(x) * \text{sqrt}(x) \approx x$.
- ◆ Implementation

```
float sqrt(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

Heap sort using library data structure

- ◆ Priority queue: structure with three operations

```
empty : pq
insert : elt * pq → pq
deletemax : pq → elt * pq
```

- ◆ Algorithm using priority queue (heap sort)

```
begin
    empty pq s
    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 (see HW)

Language support for info hiding

- ◆ 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

In procedural languages, refine a procedure or data type by rewriting it. Incremental reuse later with objects.

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

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`

Clu Clusters

```
complex = cluster is
  make_complex, real_part, imaginary_part, plus, times
rep = struct [ re, im : real]
make_complex = proc (x,y : real) returns (cvt)
  return (rep${re:x, im:y})
real_part = proc (z:cvt) returns real
  return (z.re)
imaginary_part = proc (z:cvt) returns real
  return (z.im)
plus = proc (z, w: cvt) returns (cvt)
  return (rep${ re: z.re+w.re, im: z.im+w.im })
mult = proc ...
end complex
```

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

Abstype for Complex Numbers

◆ Input

```
abstype cmplx = C of real * real with
  fun cmplx(x,y: real) = C(x,y)
  fun x_coord(C(x,y)) = x
  fun y_coord(C(x,y)) = y
  fun add(C(x1,y1), C(x2,y2)) = C(x1+x2, y1+y2)
end
```

◆ Types (compiler output)

```
type cmplx
val cmplx = fn : real * real -> cmplx
val x_coord = fn : cmplx -> real
val y_coord = fn : cmplx -> real
val add = fn : cmplx * cmplx -> cmplx
```

Abstype for finite sets

◆ Declaration

```
abstype 'a set = SET of 'a list with
  val empty = SET(nil)
  fun insert(x, SET(elts)) = ...
  fun union(SET(elts1), SET(elts2)) = ...
  fun isMember(x, SET(elts)) = ...
end
```

◆ Types (compiler output)

```
type 'a set
val empty = - : 'a set
val insert = fn : 'a * ('a set) -> ('a set)
val union = fn : ('a set) * ('a set) -> ('a set)
val isMember = fn : 'a * ('a set) -> bool
```

Encapsulation Principles

◆ Representation Independence

- Elements of abstract type can be implemented in various ways
- Restricted interface -> client program cannot distinguish one *good* implementation from another

◆ Datatype Induction

- Method for reasoning about abstract data types
- Relies on separation between interface and implementation

Representation Independence

◆ Integers

- Can represent 0,1,2, ..., -1,-2, ... any way you want
- As long as operations work properly
 - + , - , * , / , print , ...
- Example
 - 1's complement vs. 2's complement

◆ Finite Sets

- can represent finite set {x, y, z, ... } any way you want
- As long as operations work properly
 - empty, ismember?, insert, union
- Example
 - linked list vs binary tree vs bit vector

Reality or Ideal?

◆ In Clu, ML, ... rep independence is a theorem

- Can be proved because language restricts access to implementation: access through interface only

◆ In C, C++, this is an ideal

- "Good programming style" will support representation independence
- The language does not enforce it
 - Example: print bit representation of -1
 - This distinguishes 1's complement from 2's complement

Induction

(Toward Datatype Induction)

◆ Main idea

- 0 is a natural number
- if x is a natural number, then x+1 is a natural number
- these are all the natural numbers

◆ Prove p(n) for all n

- prove p(0)
- prove that if p(x) then p(x+1)
- that's all you need to do

Induction for integer lists

- ◆ Principle
 - nil is a list
 - if y is a list and x is an int, then cons(x,y) is a list
 - these are all of the lists
- ◆ Prove p(y) for all lists y
 - prove p(nil)
 - prove that if p(y) then p(cons(x,y))
 - that's all you need to do
- ◆ Example: next slide
 - Note: we do not need to consider car, cdr
 - Why? No new lists. (No subtraction in integer induction.)

Example of list induction

- ◆ Function to sort lists
 - fun sort(nil) = nil
 - | sort(x::xs) = insert(x, sort(xs))
- ◆ Insertion into sorted list
 - fun insert(x, nil) = [x]
 - | insert(x, y::ys) = if x < y then x::(y::ys)
 - | insert(x, y::ys) = y::insert(x,ys)
- ◆ Prove correctness of these functions
 - Use induction on lists (easy because that's how ML let's us write them)

Interfaces for Datatype Induction

- ◆ Partition operations into groups
 - constructors: build elements of the data type
 - operators: combine elements, but no "new" ones
 - observers: produce values of other types
- ◆ Example:
 - sets with

```
empty : set
insert : elt * set -> set
union : set * set -> set
isMember : elt * set -> bool
```
 - partition

```
constructors: empty, insert
operator: union
observer: isMember
```

Induction on constructors

- ◆ Operator: produces no new elements
 - Example: union for finite sets

```
Every set defined using union can be defined without union:
union(empty, s) = s
union(insert(x,y), s) = insert(x, union(y,s))
```
- ◆ Prove property by induction
 - Show for all elements produced by constructors
 - Set example: Prove P(empty) and P(y) => P(insert(x,y))
 - This covers all elements of the type

Example in course reader: equivalence of implementations

Example of set induction

- ◆ Assume map function
 - map(f,empty) = empty
 - map(f, insert(y,s)) = union(f(y), map(f,s))
- ◆ Function to find minimum element of list
 - fun intersect(s,s') = if empty(s') then s'
 - else let f(x) = if member(x,s) then {x} else empty
 - in map(f, s') end;
- ◆ Prove that this work:
 - Use induction on s':
 - Correct if s' = empty
 - Correct if s' = insert(y, s'')

What's the point of all this induction?

- ◆ Data abstraction hides details
- ◆ We can reason about programs that use abstract data types in an abstract way
 - Use basic properties of data type
 - Ignore way that data type is implemented
- ◆ This is not a course about induction
 - We may ask some simple questions
 - You will not have to derive any principle of induction

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

```
module Set
interface
  type set
  val empty : elt
  fun insert : elt * set -> set
  fun union : set * set -> set
  fun isMember : elt * set -> bool
implementation
  type set = elt list
  val empty = nil
  fun insert(x, elts) = ...
  fun union(...) = ...
  ...
end Set
```

- ◆ Can define ADT
 - Private type
 - Public operations
- ◆ More general
 - Several related types and operations
- ◆ Some languages
 - Separate interface and implementation
 - One interface can have multiple implementations

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

C++ Templates

- ◆ Type parameterization mechanism
 - `template<class T> ...` indicates type parameter T
 - C++ has class templates and function templates
 - Look at function case now
- ◆ Instantiation at link time
 - Separate copy of template generated for each type
 - Why code duplication?
 - Size of local variables in activation record
 - Link to operations on parameter type

Example

- ◆ Monomorphic swap function

```
void swap(int& x, int& y){
  int tmp = x; x = y; y = tmp;
}
```
- ◆ Polymorphic function template

```
template<class T>
void swap(T& x, T& y){
  T tmp = x; x = y; y = tmp;
}
```
- ◆ Call like ordinary function

```
float a, b; ... swap(a, b);
```

Generic sort function

- ◆ Function requires < on parameter type

```
template <class T>
void sort( int count, T * A[count] ) {
  for (int i=0; i<count-1; i++)
    for (int j=i+1; j<count-1; j++)
      if (A[j] < A[i]) swap(A[i],A[j]);
}
```
- ◆ How is function < found?
 - Link sort function to calling program
 - Determine actual T at link time
 - If < is defined on T, then OK else error
 - May require overloading resolution, etc.

Compare to ML polymorphism

```
fun insert(less, x, nil) = [x]
| insert(less, x, y::ys) = if less(x,y) then x::y::ys
                           else y::insert(less,x,ys)

fun sort(less, nil) = nil
| sort(less, x::xs) = insert(less, x, sort(less,xs))
```

- ◆ Polymorphic sort function
 - Pass operation as function
 - No instantiation since all lists are represented in the same way (using cons cells like Lisp).
- ◆ Uniform data representation
 - Smaller code, can be less efficient, no complicated linking

Standard Template Library for C++

- ◆ Many generic abstractions
 - Polymorphic abstract types and operations
- ◆ Useful for many purposes
 - Excellent example of *generic programming*
- ◆ Efficient running time (but not always space)
- ◆ Written in C++
 - Uses template mechanism and overloading
 - Does *not* rely on objects

Architect: Alex Stepanov

Main entities in STL

- ◆ Container: Collection of typed objects
 - Examples: array, list, associative dictionary, ...
- ◆ Iterator: Generalization of pointer or address
- ◆ Algorithm
- ◆ Adapter: Convert from one form to another
 - Example: produce iterator from updatable container
- ◆ Function object: Form of closure ("by hand")
- ◆ Allocator: encapsulation of a memory pool
 - Example: GC memory, ref count memory, ...

Example of STL approach

- ◆ Function to merge two sorted lists
 - $\text{merge} : \text{range}(s) \times \text{range}(t) \times \text{comparison}(u) \rightarrow \text{range}(u)$
This is conceptually right, but not STL syntax.
- ◆ Basic concepts used
 - $\text{range}(s)$ - ordered "list" of elements of type s , given by pointers to first and last elements
 - $\text{comparison}(u)$ - boolean-valued function on type u
 - subtyping - s and t must be subtypes of u

How merge appears in STL

- ◆ Ranges represented by iterators
 - iterator is generalization of pointer
 - supports ++ (move to next element)
- ◆ Comparison operator is object of class Compare
- ◆ Polymorphism expressed using template

```
template < class InputIterator1, class InputIterator2,
           class OutputIterator, class Compare >
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, InputIterator1 last2,
                    OutputIterator result, Compare comp)
```

Comparing STL with other libraries

- ◆ C:

```
qsort( (void*)v, N, sizeof(v[0]), compare_int );
```
- ◆ C++, using raw C arrays:

```
int v[N];
sort( v, v+N );
```
- ◆ C++, using a vector class:

```
vector v(N);
sort( v.begin(), v.end() );
```

Efficiency of STL

◆ Running time for sort

	N = 50000	N = 500000
C	1.4215	18.166
C++ (raw arrays)	0.2895	3.844
C++ (vector class)	0.2735	3.802

◆ Main point

- Generic abstractions can be convenient and efficient !
- But watch out for code size if using C++ templates...