510: Programming Languages

Data Abstraction

David Walker

Fall, 2002

What is Data Abstraction?

Reynolds: "Types are a syntactic mechanism for enforcing levels of abstraction."

What does this mean in practical terms? How can we use a type system to enforce abstraction?

Can we precisely characterize what we mean by an abstract type?

Type Safety and Abstraction

Type safety is a manifestation of Reynolds' principle!

Type safety states that there is no "confusion" between values of distinct types.

1. Cannot apply a number to a boolean.

2. Cannot add two functions.

The representations of values are hidden from the programmer.

Type Safety and Abstraction

MinML types are abstract because we can only use the designated operations for creating and using values of that type.

- Integers are "created" by writing numerals, and are used by the arithmetic and comparison operations.

- Booleans are "created" by `true` and `false` and are used by the conditional expression.

- Functions are "created" by lambda abstractions, and are used by application.

Abstract Types

An abstract type consists of two things:

1. A **representation type** for the values of the ADT. The representation type is hidden from the user of the ADT.

2. A collection of **operations** for creating and using values of the abstract type. The operations are implemented in terms of the representation type, but their code is hidden from the user of the ADT.

Client-Implementor Model

This suggests a separation between the **client** of the abstraction and the **implementor** of the abstraction.

- The **implementor** knows the representation details. *e.g.*, the compiler-writer knows how integers and functions are represented.

- The **client** knows only the type name and the operations on it. *e.g.*, the programmer does not know how integers and functions are represented.

Example: Integer Queues

Representation type: `queue`.

Operations:

1. `empty`, the empty queue.

2. `insert`(n,q), taking an integer n and a queue q, yielding a queue with that element inserted at the back.

3. `remove`(q), taking a queue q and returning the front element and a queue with that element removed.

Example: Integer Queues

That's what the client sees. The implementor must provide

1. a definition for the representation of type `queue`;

2. implementations of the operations consistent with that representation.

Defining ADT's in ML

The interaction between the client and implementor is mediated by a **signature** specifying the representation type and its operations.

```
signature QUEUE =
  sig
    type queue
    val empty : queue
    val insert : int * queue -> queue
    val remove : queue -> int * queue
  end
```

Defining ADT's in ML

The ADT is implemented by a **structure** binding, with an **opaque** signature ascription to hide the representation type.

```
structure IntQueue :> QUEUE =
  struct
    type queue = int list
    val empty = nil
    fun insert (x, xs) = x::xs
    fun remove [x] = (x, [])
      | remove (x::xs) =
        let val (y, xs') = remove xs in
          (y, x::xs')
        end
  end
```

Defining ADT's in ML

Another implementation of queues, as a pair of lists:

```
structure IntQueue :> QUEUE =
  struct
    type queue = int list * int list
    val empty = (nil, nil)
    fun insert (x, (bs, fs)) = (x::bs, fs)
    fun remove (bs, nil) =
          remove (nil, rev bs)
      | remove (bs, f::fs) =
          (f, (bs, fs))
  end
```

Defining ADT's in C

The **header file** serves as the interface. It consists of

1. A type definition for the representation type.

2. The prototypes of the operations.

Unfortunately, it is not possible to hide the representation type from the client (because of casting).

Example: Queues in C

```
typedef struct queue {
  int elt;
  struct queue *next;
} QUEUE;

QUEUE *empty();
QUEUE *insert (int x, QUEUE *q);
QUEUE *front (QUEUE *q);
QUEUE *delete (QUEUE *q);
```

Information Hiding: Pro's

Why is it advantageous to hide the representation type?

- The **representation can be changed** without affecting the correctness of user-level code.

- **Safety is ensured** because values of a type cannot be abused in an implementation-dependent manner.

- The implementor can **enforce invariants**.

Information Hiding: Con's

Doesn't data abstraction limit expressiveness?

- It depends on what you mean by expressiveness! What is important is to have the **choice** to hide or expose the representation type. Any **fixed** policy (always hidden, always exposed) is worse than having a choice.

- Expressiveness depends as much on the **strictures** as the **affordances** of a language! The Cole Porter dictum ("anything goes") is a **bad** principle of language design because it deprives you of choice!

Information Hiding: Con's

Doesn't data abstraction limit efficiency?

- At the lowest level, yes — certain "cool hacks" are not possible without knowing the details of the representations.

  But watch out: Apple spent man-years optimizing their 2d graphics code at the lowest level. But their core algorithm was $O(n^2)$, whereas an $O(n \lg n)$ algorithm existed!

- There are many notions of efficiency, *e.g.*, cost of maintenance, cost of modification, ease of team development, that are not measured in instructions per second.

Enforcing Invariants

Using an ADT definition mechanism the implementor can **enforce** his or her own levels of abstraction, called **representation invariants (RI's)**.

Since the representation type is hidden, the client can **only** use the operations of the ADT to manipulate its values.

Consequently, the **only** possible violations of the representation invariant are those that are introduced by the operations of the ADT.

This **dramatically reduces** the points at which errors can arise! If the language supports data abstraction, you can be certain that any RI violation must be in the implementor's code.

Example: Dictionaries

Dictionaries with string keys:

```
signature DICT =
  sig
    type 'a dict
    val empty : 'a dict
    val lookup : 'a dict * string -> 'a
    val insert :
            'a dict * string * 'a -> 'a dict
  end
```

Example: Red-Black Trees

Dictionaries can be implemented with red-black trees.

```
structure Dict :> DICT =
  struct
    datatype 'a dict =
      Empty |
      Red of 'a dict * string * 'a dict |
      Black of 'a dict * string * 'a dict
    ...
  end
```

Example: Red-Black Trees

Representation invariants:

1. Every child of a red node is black. That is, no red node has a red child.

2. The **black height** of the left and the right child of a node is the same. Black height = maximum number of black nodes on any path to a leaf.

If the representation invariant holds, the tree is **balanced** — the height of an n-node tree is $O(\lg n)$. That is, there are no long paths in the tree.

Example: Red-Black Trees

The job of the implementor is to **maintain** the red-black invariants:

1. Each operation may **assume** that the invariant holds on entry. That is, if the input is a value of type `'a dict`, then the rbt invariants may be assumed to hold.

2. Each operation must **ensure** that the invariants hold on exit. That is, if the result is a value of type `'a dict`, it is incumbent on the implementor to ensure that the rbt invariants hold.

Example: Red-Black Trees

From an **external** point of view, dictionaries are always balanced. The client only ever sees well-balanced trees.

From an **internal** point of view, the dictionary may go through intermediate states in which the red-black invariant does not hold.

- A newly-inserted element is colored red.

- The code removes red-red violations while preserving the black height condition.

ADT's as Abstract Machines

Abstract types may be thought of as **API's (Application Programmer Interfaces)** or **Abstract Machines**.

The values of the representation type are **states** of an abstract machine.

The operations of the ADT are **instructions** that transform states.

Any invariants governing the state are private to the implementor of the machine, and are invisible to the programmer of that machine.

ADT's as Critical Sections

We may also think of the implementation of an ADT is a **critical section**.

During execution of the operations, the dictionary may be in a "bad" state.

The operations restore it before anyone notices.

Representation Independence

Abstract types enjoy **representation independence**: the behavior of client code does not change if we change the implementation of an ADT, provided that the operations "respect" the change.

For example, the behavior of the client of the `Queue` structure is not affected by whether we use the one-list or two-lists representation of queues.

Why? Because the client code is **polymorphic** in the representation type of the ADT!

Advantages of Data Abstraction

A data abstraction mechanism lets you **enforce** invariants in a program.

- Client code cannot violate these invariants.

- Any violations are traceable to a **single location** in the code.

Advantages of Data Abstraction

A data abstraction mechanism supports **maintenance** and **team development**.

- Client code is insulated from changes to the abstraction.

- Teams can work relatively independently, with little fear of integration nightmares.

Advantages of Data Abstraction

Most popular languages **do not support** enforced data abstraction.

- C, C++, Lisp, Cobol, Fortran do not.
- Java does, via `final` classes.

Disadvantages of Data Abstraction

Using ADT's requires **careful planning**.

- Must carefully design interfaces. This takes time and effort. Revision is unavoidable.

- Must pay a sizable "up front" cost to design abstractions. Pay-off is not immediate, but the benefits are huge.

Disadvantages of Data Abstraction

If absolute maximum efficiency is paramount, the cost of abstraction may be too high.

- A good abstraction mechanism will let you selectively expose the representation, so you need not abandon ADT's entirely.

- Do those tiny constant factors really matter?

Prospectus

The **core mechanism** of data abstraction is **polymorphism**!

- Client code is **abstract** in the representation type.

- This is the same as being **polymorphic**!

Polymorphism and data abstraction are modelled by **universal** and **existential** types.