Data Abstraction & Modularity

Edward Z. Yang
Today's topic is data abstraction and modularity. This is a little different from the other lectures, where there was a "thing" to learn; today's topic is a bit more squishy. But roughly, I'll try to organize the lecture two ways: first, I want to talk about WHAT modularity is, and why we might care about it. Then, I'll talk about various programming language features in the service of modularity.

1. Philosophy

2. Language Support
What is abstraction? If there is only one thing you take away from this lecture, it is the answer to this question. Abstraction is a black box within which you put all of the implementation involved with making some component work. Then, when you use the component, you NEVER look at the contents. It's a black box.
If you are ever so tempted to stick your hand into the box again, to see what is actually going on...
...don't be surprised if your hand gets bitten off by an angry weasel.
The weasel represents everything that could go wrong when you violate the abstraction boundary (the black box). Perhaps today, when you reach into the black box, the weasel won't bite, but some day, assuredly, he will get you back.

Yeah, it's all angry weasels in the box. Don't stick your hand in.
To put it another way, the abstraction barrier is this black line between the interface and specification of a software component, and the implementation of the software component. You, the user of software, are allowed to touch and use the specified interface. The abstraction barrier separates this nice interface from the angry weasels in the implementation: it provides us a way of INFORMATION HIDING: the implementation is HIDDEN!
Example: Function Abstraction

One of the simplest examples of abstraction is functional abstraction. Take this example of a square root function. As a user, all you need to know is that it takes a float as an input, returns a float, and that the returned float, when squared, will roughly equal the input.

**INTERFACE**

```plaintext
float sqrt(float x)
```

**SPECIFICATION**

```plaintext
if x > 1 then
  sqrt(x) * sqrt(x) ≈ x
```

You don't need to know how the function is implemented. Maybe it does some sort of guess and iterate approach. By the POWER OF ABSTRACTION, you don't care.

```plaintext
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 Abstraction

Another classic example is data abstraction. Here is an example of a priority queue data structure. Its interface consists of a type, as well as some operations on that type. We might understand what the queue is supposed to do by means of a *reference implementation*; in this case a simple implementation in terms of sorted lists.

**INTERFACE**

```
data PQ a  
empty :: PQ a  
insert :: a → PQ a → PQ a  
deleteMin :: PQ a → (a, PQ a)  
```

**SPECIFICATION**

```
data PQ a = [a]  
empty = []  
insert a pq = sort (a:pq)  
deleteMin (a:pq) = (a, pq)  
```

You don't need to know how the data structure is implemented. Maybe it is backed by a binary heap; maybe there's some more fancy data structure going on. By the POWER OF ABSTRACTION, you don't care.

**IMPLEMENTATION**

```
data PQ a = PQ (Heap a)  
empty = PQ emptyHeap  
insert a (PQ pq) = PQ (insertHeap a pq)  
deleteMin (PQ pq) = ...  
```
Another example is type-classes. The functor type-class specifies that a type in question supports one operation, the "fmap" operation, and also specifies what ALGEBRAIC LAWS the operation should uphold.

**INTERFACE**

class Functor f where

\[
fmap :: (a \rightarrow b) \rightarrow f a \rightarrow f b
\]

**SPECIFICATION**

\[
fmap id = id
\]

\[
fmap (f \cdot g) = fmap f \cdot fmap g
\]

When we use the fmap function, we don't care what the actual data type in question is.

**IMPLEMENTATION**

```
instance Functor [] where
    fmap = map
```
Example: Type Classes

It could even be some weird implementation for a type that we didn't even think was a "container" all that matters is that the specification is upheld (which it is. You're encouraged to check it out!)

```
interface Functor f where
  fmap :: (a -> b) -> f a -> f b
specification
  fmap id = id
  fmap (f . g) = fmap f . fmap g

instance Functor ((->) r) where
  fmap f g = f . g
```
By the way, an executable specification can serve dually as a test for the implementation itself. For example, if I want to test my square root function, I can simply run sqrt on an input, multiple the result by itself, and check if it's approximately equal. This is way better than hard-coding the outputs for the test! (The same concept applies when you have a reference implementation: the reference is easier to implement and more likely to be correct, and a good sanity check.)
QuickCheck

QuickCheck is a library for Haskell which takes this concept to its logical extreme: it combines executable specifications with the ability to randomly generate test-inputs. Testing now is the process of randomly generating a pile of inputs and running your specification on them.

```
prop_sqrt x =
  x > 1 ==> approxEq (sqrt(x) * sqrt(x)) x
```

There actually is a small cottage industry of refinements on this basic idea. SmallCheck is a variant which exhaustively tests your function on small inputs; SmartCheck is a variant which can automatically MINIMIZE your test-cases when they fail, making it much easier to figure out what is going wrong.

+++ OK, passed 100 tests
Software components should not be entwined in conjugal embrace
The abstraction barrier gives us a well defined interface by which our components can communicate.
The benefits of this separation are best seen when building large software projects. Imagine a software system as a collection of components which build upon each other in some way.
A well defined specification makes it possible to swap one component for another, if the second also satisfies the specification. To put it another way, you can make internal improvements (which don't affect the visible behavior) without breaking the system.
One of the tricky questions of software design is how you come to design this collection of components. There seem to be two schools of thought here: the top-down school says you should start with a high-level program description and successively refine the problem statement until you have something you can implement.

The bottom up school says to start off building small reusable components that are basic to the system, and combine them to create the full system.

It's probably the case that most real programmers apply a combination of these two methodologies.
There were two culturally important essays written on this topic back in the 70s. In "Program Development by Stepwise Refinement", Niklaus Wirth gave an example of how one might go about solving the 8-queens problem by starting with a high-level description of the program and gradually decomposing the instructions into more detailed ones.

If you've never had to solve the eight-queens problem, it's a classic, and worth a try. Then go and read how Wirth solved the problem. One of the big insights from this essay is that program refinement also involves DATA refinement: the choice of an appropriate data representation.

The 8-queens problem, by the way, asks how to place eight queens on a chess board such that none of them could capture any other of the queens.
I also like the David Parnas essay, which asks how one decides where to put the abstraction boundaries, by which the system will be modularized. His running example is the "KWIC Index", a simple text-based program which takes a list of titles and creates an alphabetized list appropriate for an index (in particular, with duplicate entries for multi-word titles with the words shifted around, in case you search for "A Bat" under "B" rather than "A".)
What is so good about this essay is Parnas gives an example of how NOT to modularize the system, which he also describes as the "conventional" modularization. I've reproduced it here: the "obvious" way to build KWIC is to think of it as some sort of flow-chart, where input flows in, is parsed, circular shifted, alphabetized, and then output. So create a "module" for each of these components, and a master control to glue it altogether.

What Parnas shows is that, while this organization seems intuitively appealing, it has major problems. First, there is major, shared architecture throughout the entire program which must be decided upon in advance before any development can occur. What is the "specification" of the circular shift component? It's impossible to know without knowing how the input is parsed. Second, it is very difficult to make otherwise reasonable changes to the program. If you're interested, please see the paper!
Parnas's recommended modularization takes a different approach, centered on the problem of information hiding. We start off with "line storage", which provides an abstract interface for manipulating lines. The INPUT module interacts with this component, as does the circular shifter; but the circular shifter NEVER needs to know about the INPUT. This means if the input format changes, no changes to the circular shifter are necessary. Modularity!

See the paper for more!
From here, we'll move on to the second part of the lecture, language support for modularity. That is to say, HOW does your programming language assist in the development of modular programs?
Let's return to our earlier picture, the black abstraction barrier separating interface and implementation. We can summarize the two things our language can do for us. First, they can provide us with a way to articulate the interface (e.g. specification) of our components in a more machine-readable way than just plain old documentation. Second, they can provide a way of enforcing that users NOT stick their hands in the black box: a mechanism of isolation which programmers can use to hide their data.
Example: Function Abstraction

If we return to our function abstraction example, in many languages (like Haskell), the type signature provides a way of saying the interface of the black box (a function), which is a sort of mini-specification of the what code is supposed to do. And the way they enforce the abstraction is, well, you CALL A FUNCTION (instead of copy-pasting the code). So when the implementation is updated in one place, you get the updates too.

Another minor screed: when people tell people to use types, they often emphasize how it catches bugs early. But I think this is missing the most important reason why dynamically typed languages are doomed: and this is the fact that, by eschewing types you have doomed yourself to never be able to express in a machine-readable way one of the most BASIC contracts that your code is supposed to specify. No wonder it is so hard to tell if anything broke when you upgraded your library.

By the way, if type signatures are the most important for "programming in the large", then it is an interesting question how to adopt types at the high-level organization of a program, while still allowing the flexibility of dynamic typing in the small. This is what the research program on gradual typing is about. It's a very interesting program; Here is Robby Findler's keynote on the topic at ICFP'14

https://www.youtube.com/watch?v=gXTbMPVFP1M
Example: Data Abstraction

For our example of data abstraction, the key recipe that a language can give you is the ability to hide data representation: to make it impossible for a client to reach their hand in and see how you've organized a type internally.

"Abstract Data Types"

Interface

Ability to hide data representation

like user-defined "built-in" types
In Haskell, data abstraction is achieved by a little trick: when we export a type associated with some data structure, we do NOT have to export a constructor for the type. This means that it is not possible for a client to actually see how the data type is implemented under the hood.

```haskell
empty :: PQ a
insert :: a -> PQ a -> PQ
deleteMin :: PQ a -> (a, PQ a)
```

```haskell
module PQ (PQ, ...) where
data PQ = PQ MinHeap
```

↑ hidden
In Java, data abstraction is achieved in a more direct way, using the "private" annotation which makes it illegal to access a property outside of the class. Externally, a private property may as well not exist.

```java
class PQ<A> {
    private A[] backingArray;

    static public PQ<A> empty() { ... }
    public PQ<A> insert(A x) { ... }
    public Tuple<A, PQ> deleteMin() { ... }
}
```
We've actually already seen how data abstraction is achieved in JavaScript: by the use of closures. JavaScript doesn't have types, and is a very flexible language, so other methods of enforced abstraction are few and far between.

Example: Data Abstraction in JavaScript

```javascript
function New() {
    var hiddenState = false;
    return {
        toggle: function () {
            hiddenState = !hiddenState;
        }
    }
}
```
Example: Separate interface from implementation in Haskell

Our last example was type classes in Haskell. We can think of this as the ability to separate our interface from the implementation, so that a client can refer to the interface in abstract, and not commit to any particular implementation.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

```
instance Functor [] where
  fmap = map
```
Example: Separate interface and implementation in Java

In Java, interfaces play a similar role for this sort of polymorphism.

```java
interface PQ<A> {
    static public PQ<A> empty();
    public PQ<A> insert(A);
    public Tuple<A, PQ> deleteMin();
}

class Heap implements PQ {
    ...
}
```
A brief comparison between type classes and interfaces. They serve a similar role, but they differ in some important ways.

- Type dispatch or Value dispatch
- Multiple constraints?
- Retroactive instances?
- Subtyping versus Variance
Now I want to talk about a hugely important technical landmark in language features for modularity: the ML module system. It is probably not an exaggeration to say that the ML module system is the MOST INFLUENTIAL programming language feature made in the service of modularity. What can the ML module system do that Java interfaces and Haskell type classes can't? They key is that in ML, a module can contain types (that's plural).

**big idea:** modules contain types

Unfortunately, the ML module system also gets a little bit of a bad reputation for being obscure and hard to understand. While it is probably true that that the module system can be somewhat annoying and fiddly to use, and that the design of the module system has unearthed some legitimately challenging research problems, I think at its core the ML module system is very simple, and that's what I want to convince you about today.

These days, the way I actually like to think about the ML module system is as a *limited form of dependent types based on intensional equality*. Don't worry if you didn't understand, but if you can parse what these individual words mean, it's something worth thinking about.
Let's take our priority queue example and rewrite it in ML. A few pointers about ML syntax: the "type" keyword is used to declare new types, polymorphic type variables are declared BEFORE the name, with a tick in front ('a queue). All type names are lowercase while constructors are upper-case, and a declaration is made with the "let" keyword. If you eyeball it, it should look pretty clear.

In ML, this combination of types and functions which operate on these types is referred to as a "structure". You can think of it as a module.
The crux of the ML module system is that we can assign this structure a "type", which we call a *signature*. Actually, the signature is very similar to the informal signatures we've been writing for Haskell modules (which are not writeable in today's Haskell, although we hope to change that soon.)

**Structure**
```
type priority = int
type 'a queue = ...

let empty = Empty
let rec insert q p e = ...
```

**Signature**
```
type priority
type 'a queue

val empty : 'a queue
val insert : 'a queue -> priority -> 'a -> 'a queue
```
Module Systems

**Signature**
- type priority
- type 'a queue

val empty : 'a queue
val insert : 'a queue -> priority -> 'a -> 'a queue

**Structure**
- type priority = int
- type 'a queue = ...

let empty = Empty
let rec insert q p e = ...

(ML)
Functors (not Haskell!)

When you have a signature, what can you do with it? ML provides modules which are parametrized over other modules, (confusingly) caused functors (not the Haskell functor!)

functor (Elt : ORDERED_SIG) ->

struct
    type element = Elt.t
    type set = element list
    :
    :

In this example functor, we take a module named Elt, which is required to implement the signature ORDERED_SIG (which defines some type t as well as order comparison functions on it. We can then locally refer to the type from this module, and use the operations of Elt on it.
The ability to parametrize an ML module by another module is what distinguishes it from Haskell modules, or really the modules of most other programming languages.
ML modules versus Haskell typeclasses

So much so that the two aren't really comparable; really, Haskell's type classes serve the role more.
ML modules are more expressive than Haskell's type classes, but they also take more work to use. Type classes, on the other hand, are a very lightweight mechanism for specifying interfaces. There are, of course, situations where ML-style modules are really useful; and this is what the Backpack work in Haskell (one of my research interests) has been about.
Conclusion

the all important black box!