Object Oriented Design Issues

FP vs OOP

- In FP (and procedural), programs break down into functions that perform some operations
  - Functions may take one or more arguments
- In OOP, programs break down into classes that give behavior to some kind of data
- These two approaches are exactly opposite and provide complementary perspectives to the same problem
- Which approach to take depends on how the software is planned to be extended

Basic set-up

- Expressions for a small "language" such as for arithmetic
- Different variants of expressions, such as *integer expressions, negation expressions or addition expressions*
- Different operations over expressions, such as *evaluating them, converting them to strings*

	Evaluate	toString
Integer		
Negation		
Addition		

- Conceptual matrix must be populated indicating how the program would behave for each grid of the matrix

Functional Approach

- Define a *datatype for expressions, with one constructor for each variant*
  - For languages with dynamic typing we don't need to name the data types
- Define a *function for each operation*
- In each function, have a branch (e.g., via pattern-matching) for each variant of data
  - If there is a default for many variants, we can use something like a wildcard pattern to avoid enumerating all the branches

```
exception BadResult of string

datatype exp =
    Int    of int
  | Negate of exp
  | Add    of exp * exp

fun eval e =
    case e of
        Int _       => e
      | Negate e1   => (case eval e1 of
                          Int i => Int (~i)
                        | _ => raise BadResult "non-int in negation")
      | Add(e1,e2) => (case (eval e1, eval e2) of
                          (Int i, Int j) => Int (i+j)
                        | _ => raise BadResult "non-ints in addition")

fun toString e =
    case e of
        Int i       => Int.toString i
      | Negate e1   => "-(" ^ (toString e1) ^ ")"
      | Add(e1,e2) => "("  ^ (toString e1) ^ " + " ^ (toString e2) ^ ")"
```

Object Oriented Approach

- Define a *class for expressions, with one abstract method for each operation*
- Define a *subclass for each variant of data*
- In each subclass, have a method definition for each operation
  - If there is a default for many variants we can use a method definition in the superclass so that via inheritance we can avoid enumerating all the branches
- This approach is data-oriented decomposition: breaking the problem down into classes corresponding to each data variant

Comparison

- Functional decomposition breaks programs down into functions that perform some operation and object-oriented decomposition breaks programs down into classes that give behavior to some kind of data
- Its about deciding whether to lay out a program “by column” or “by row”
- This is needed in conceptualizing software or deciding how to decompose a problem
- Various tools and IDEs help to view a program in a way different than how the code is decomposed

Comparison: Context of the Example

- It is “more natural” to have the cases for eval together rather than the operations for Negate together

- For problems like implementing graphical user interfaces, the object-oriented approach is probably more popular
  - It is “more natural” to have the operations for a kind of data (like a MenuBar) together (such as backgroundColor, height, and doIfMouseIsClicked rather than have the cases for doIfMouseIsClicked together (for MenuBar, TextBox, SliderBar, etc.)

Extensibility

- Adding a new operation is easy following functional approach

```
fun noNegConstants e =
    case e of
        Int i       => if i < 0 then Negate (Int(~i)) else e
      | Negate e1   => Negate(noNegConstants e1)
      | Add(e1,e2)  => Add(noNegConstants e1, noNegConstants e2)
```

- Adding a new data variant, such as Mult of exp * exp is less pleasant
- In OOP approach it is just the opposite

Plan for Unplanned Extensions

- Making software that is both robust and extensible is valuable but difficult
- Extensibility can make the original code more work to develop, harder to reason about locally, and harder to change(without breaking extensions)
- In fact, languages often provide constructs exactly to *prevent extensibility*
  - *Final* class cannot be extended

More Extensions to the Problem

- The methods may take two or more arguments or evaluating the expressions can be more complicated if different kinds of data are considered
- Add can be modified as follows
  - If the arguments are ints or rationals, do the appropriate arithmetic
  - If either argument is a string, convert the other argument to a string (unless it already is one) and return the concatenation of the strings.

	Int	String	Rational
Int			
String			
Rational			

More Extensions to the Problem

	Int	String	Rational
Int			
String			
Rational			

- If many cases work the same way, we can use wildcard patterns and/or helper functions to avoid redundancy
- One common source of redundancy is *commutativity, i.e., the order of values not mattering*
  - adding a rational and an int is the same as adding an int and a rational

```
Value eval() {
        return new Int(((Int)(e1.eval())).i + ((Int)(e2.eval())).i);
}
```

Binary methods in OOP

```
Value eval() {
        return e1.eval().add_values (e2.eval());
}
```

- An Int, MyRational, or MyString should "know how to add itself to another value"
- By putting add_values methods in the Int, MyString, and MyRational classes, the work is divided into three pieces using dynamic dispatch depending on the class of the object that e1.eval returns, i.e., the receiver of the add_values call in the eval method in Add

Binary methods in OOP

- But then each of these three needs to handle three of the nine cases, based on the class of the second argument
  - Any of addInt(..), addMyString(..), addMyRational(...) of Int class would be called if the first argument is Int
- While this approach works, it is really not object-oriented programming. Rather, it is a mix of object-oriented decomposition (dynamic dispatch on the first argument) and functional decomposition (using is_a? to figure out the cases in each method)

- The extensibility advantage of OOP is lost!

Multimethods

- OOP languages with *multimethods, do not require the manual double dispatch*
- Each add_values method would indicate the class it expects for its argument
- Then e1.eval.add_values e2.eval would pick the right one of the 9 by, at run-time, considering the class of the result of e1.eval *and the class of the result of e2.eval*

Multimethods

- In languages that support multimethods, one *can have multiple methods in a class* with the same name and the method-call semantics does use the types of the arguments to choose what method to call

- But it uses the *types of the arguments, which are determined at compile-time and not the* run-time class of the result of evaluating the arguments. This semantics is called *static overloading*

- *Static overloading is* considered useful and convenient, but it is not multimethods and does not avoid needing double dispatch

Interfaces-Multiple Inheritance

- Implementing interfaces does not inherit code
- It is purely related to type-checking in statically typed languages like Java and C#
- It makes the type systems in these languages more flexible
- So Ruby- dynamically typed language, does not need interfaces
- If two interfaces have a method-name conflict, it does not matter — a class can still implement them both
- If two interfaces disagree on a method's type, then no class can possibly implement them both but the type-checker will catch that

Abstract Class

- We can have expressions with the *type of the superclass* and know that at run-time the object will actually be one of the subclasses

- Furthermore, type-checking ensures the object's class has implemented all the abstract methods, so it is safe to call these methods

- In C++, abstract methods are called "pure virtual methods" and serve much the same purpose

Abstract Methods vs Higher Order Functions

- The language supports a programming pattern where some code is passed other code in a flexible and reusable way
- In OOP, different subclasses can implement an abstract method in different ways and code in the superclass, via dynamic dispatch, can then uses these different implementations
- With higher-order functions, if a function takes another function as an argument, different callers can provide different implementations that are then used in the function body

Subtyping

- Static types for object-oriented programs, such as those found in Java is important
- If everything is an object (which is less true in Java than in Ruby), then the main thing we would want our type system to prevent is “method missing” errors, i.e., sending a message to an object that has no method for that message
- If objects have fields accessible from outside the object (e.g., in Java), then we also want to prevent “method missing" errors
- There are other possible errors as well, like calling a method with the wrong number of arguments
- While languages like Java and C# have generics these days, the source of type-system expressiveness most fundamental to object-oriented style is subtype polymorphism, also known as subtyping
- A key source of expressiveness in ML‘s type system is parametric polymorphism, also known as generics

Subtyping-small example

- In the expression {f1=e1, f2=e2, …, fn=en}, each fi is a field name and each ei is an expression.
  - The semantics is to evaluate each ei to a value vi and the result is the record value {f1=v1, f2=v2, …, fn=vn}
  - So a record value is just a collection of fields, where each field has a name and a content

- For the expression e.f, we evaluate e to a value v.
  - If v is a record with an f field, then the result is the contents of the f field
  - Our type system will ensure v has an f field

- For the expression e1.f = e2, we evaluate e1 and e2 to values v1 and v2
  - If v1 is a record with an f field, then we update the f field to have v2 for its contents.
  - Our type system will ensure v1 has an f field
    - Like in Java, we will choose to have the result of e1.f = e2 be v2, though usually we do not use the result of a field-update

Type System

- {x : real, y : real} would describe records with two fields named x and y that hold contents of type real

- If `e1` has type `t1`, `e2` has type `t2`, ..., `en` has type `tn`, then `{f1=e1, f2=e2, ..., fn=en}` has type `{f1:t1, f2:t2, ..., fn:tn}`.
- If `e` has a record type containing `f : t`, then `e.f` has type `t` (else `e.f` does not type-check).
- If `e1` has a record type containing `f : t` and `e2` has type `t`, then `e1.f = e2` has type `t` (else `e1.f = e2` does not type-check).

Problem of Type System

- With these typing rules the program would not type-check

```
fun distToOrigin (p:{x:real,y:real}) =
  Math.sqrt(p.x*p.x + p.y*p.y)

val c : {x:real,y:real,color:string} = {x=3.0, y=4.0, color="green"}
val five : real = distToOrigin(c)
```

- In the call distToOrigin(c), the type of the argument is {x:real,y:real,color:string} and the type the function expects is {x:real,y:real}, breaking the typing rule that functions must be called with the type of argument they expect

- Yet the program above is safe: running it would not lead to accessing a field that does not exist.

Subtyping

- Requires addition of
  - If some expression has a record type {f1:t1, …, fn:tn}, then let the expression also have a type where some of the fields are removed
- Letting an expression that has one type also have another type that has less information is the idea of subtyping
  - A subtype has more information
  - There are “fewer" values of the subtype than of the supertype because values of the subtype have more obligations, e.g., having more fields

Subtyping rule

- The subtyping rule added to the type system is

- The idea of one type being a subtype of another: We will write `t1 <: t2` to mean `t1` is a subtype of `t2`.
- One and only new typing rule: If `e` has type `t1` and `t1 <: t2`, then `e` (also) has type `t2`.

- Now t1<: t2 needs to be defined
- We have separated the idea of subtyping into a single binary relation that we can define separately from the rest of the type system

Defining Subtype

- One guiding principle for defining subtyping rule is substitutability
  - If we allow t1 <: t2, then any value of type t1 must be able to be used in every way a t2 can be
  - For records, that means t1 should have all the fields that t2 has and with the same types

Subtyping Rules

- "Width" subtyping: A supertype can have a subset of fields with the same types, i.e., a subtype can have "extra" fields
- "Permutation" subtyping: A supertype can have the same set of fields with the same types in a different order.
- Transitivity: If `t1 <: t2` and `t2 <: t3`, then `t1 <: t3`.
- Reflexivity: Every type is a subtype of itself: `t <: t`.

These 2 rules are common in any language with subtyping because they combine well with other rules

- Width subtyping lets us forget fields
- Permutation subtyping lets us reorder fields
  - we can pass a {x:real,y:real} in place of a {y:real,x:real}) and transitivity with those rules lets us do both
  - we can pass a {x:real,foo:string,y:real} in place of a {y:real,x:real})

Depth Subtyping

- Subtyping rules do not allow

```
{center:{x:real,y:real,z:real}, r:real} <: {center:{x:real,y:real}, r:real}
```

- To allow depth subtyping :

- "Depth" subtyping: If `ta <: tb`, then `{f1:t1,...,f:ta,...,fn:tn} <: {f1:t1,...,f:tb,...,fn:tn}`

Depth Subtyping

- In a language with records (or objects) with getters and setters for fields, depth subtyping is unsound; you cannot have a different type for a field in the subtype and the supertype
- However, if a field is not settable (i.e., it is immutable), then the depth subtyping rule is sound
- So this is yet another example of how not having mutation makes programming easier. In this case, it allows more subtyping, which lets us reuse code more.
- Another way to look at the issue is that given the three features of
  - (1) setting a field,
  - (2) letting depth subtyping change the type of a field, and
  - (3) having a type system actually prevent field-missing errors,
- we can have any two of the three.

Java Perspective

- Subtyping, even depth subtyping is implemented in java with mutable fields at the cost of runtime checks
  - Adds flexibility to the language as the same sort methods could sort both Point and ColorPoint objects
- A better solution is generics

Function Subtyping

- it is safe to pass in a function with a return type that promises more, i.e., returns a subtype of the needed return type for the function
  - Returns ColorPoint instead of Point

- In general, the rule here is
  - if ta <: tb, then t -> ta <: t -> tb, i.e., the subtype can have a return type that is a subtype of the supertype's return type

- Return types are covariant for function subtyping meaning the subtyping for the return types works \the same way" (co) as for the types overall.

Function Subtyping

- Argument types are not covariant for function subtyping
- For arguments
  - ta <: tb, does not mean ta -> t <: tb -> t

- The treatment of argument types for function subtyping is “backwards“
  - If tb <: ta, then ta -> t <: tb -> t
- Contravariance, meaning the subtyping for argument types is the reverse (contra) of the subtyping for the types overall

- The general rule is
  - If t3 <: t1 and t2 <: t4, then t1->t2 <: t3->t4
  - {x:real} -> {x:real,y:real,color:string} is a subtype of
  - {x:real,y:real} -> {x:real,y:real}

- Function subtyping is needed for higher-order functions or for storing functions themselves in records

Function Subtyping in Java/C#

- In Java/C#, subtyping rules soundly prevents \eld missing" and \method missing" errors
  - A subclass can add elds but not remove them
  - A subclass can add methods but not remove them
  - A subclass can override a method with a covariant return type
  - A class can implement more methods than an interface requires or implement a required method with a covariant return type

- A class defines an object's behavior
- Subclassing inherits behavior, modifying behavior via extension and override

- A type describes what fields an object has and what messages it can respond to
- Subtyping is a question of substitutability and checking for a type error

- In Java/C#, every class declaration introduces a class an a type with the same name

Generics vs Subtyping

- Generics are good for functions that operate over collections/containers where different collections/containers can hold values of different types:
  - length of a list or swapping two elements of a list
- Subtyping works well in graphical user interfaces
  - Much of the code for graphics libraries works fine for any sort of graphical element (“paint it on the screen," “change the background color," “report if the mouse is clicked on it," etc.) where different elements such as buttons, slider bars, or text boxes can then be subtypes
- In a language with generics (and no subtyping), higher order functions can be used for code reuse but it may become cumbersome

Bounded Polymorphism

- The key idea is to have bounded generic types, where instead of just saying “a subtype of T" or “for all types 'a," we can say, “for all types 'a that are a subtype of T“

```
class Pt {
    double x, y;
    double distance(Pt pt) { return Math.sqrt((x-pt.x)*(x-pt.x)+(y-pt.y)*(y-pt.y)); }
    Pt(double _x, double _y) { x = _x; y = _y; }
}
```

```
static List<Pt> inCircle(List<Pt> pts, Pt center, double radius) {
    List<Pt> result = new ArrayList<Pt>();
    for(Pt pt : pts)
      if(pt.distance(center) <= radius)
        result.add(pt);
      return result;
}
```

Nongeneric Method

```
static List<Pt> inCircle(List<Pt> pts, Pt center, double radius) {
    List<Pt> result = new ArrayList<Pt>();
    for(Pt pt : pts)
      if(pt.distance(center) <= radius)
        result.add(pt);
      else
        result.add(center);
      return result;
}
```

Bounded polymorphism

```
static <T extends Pt> List<T> inCircle(List<T> pts, T center, double radius) {
    List<T> result = new ArrayList<T>();
    for(T pt : pts)
      if(pt.distance(center) <= radius)
        result.add(pt);
      return result;
}
```

if center has type Pt, then the call result.add(center) does not type-check since Pt may not be a subtype of T

Thank you