Subclassing errors, OOP, and practically checkable rules to prevent them

- OOP easily breaks the separation between interface and implementation
- Subtyping is, in general, non-trivial and undecidable
- An alternative: *BRules*
  - a *statically enforceable* restriction
  - guarantees subclassing $\Rightarrow$ subtyping
  - easily implements type restrictions and polymorphic programming (ADT)

Decoupling of abstraction from implementation is one of the holy grails of good design. Object-oriented programming in general and encapsulation in particular are claimed to be conducive to such separation, and therefore to more reliable code. This talk will show a very simple example that questions if OOP indeed helps separate interface from implementation. The example is a very familiar one, illustrating the difference between subclassing and subtyping. The difference is important – but generally impossible to decide. On the constructive side, we introduce BRules – a *statically enforceable* restriction on a object-oriented system or language. BRules guarantee that subclassing always implies subtyping.

Bag vs. Set in OOP

```
class CBag
  size:: () → int
  count:: int → int
  put:: int → ()
  del:: int → bool
  clone:: CBag → CBag

operator += :: CBag,CBag → ()
operator <= :: CBag,CBag → bool


class CSet extends CBag
  memberof(x): return count(x) > 0
  put(x): if(!memberof(x)) CBag.put(x)
  clone:: CSet → CSet
```

On to the example. It is deliberately written in pseudo-code, to illustrate that the problem occurs in *any* OO system. A website referenced in the paper implements that pseudo-code in C++ – so that you can run the code and see the problem for yourselves.

Suppose I was given a task to implement a Bag – an unordered collection of possibly duplicate items (integers in this example). I chose *the following interface.* The class `CBag` defines usual methods to determine the number of all elements in a bag, to count the number of occurrences of a specific element, to put a new element into a bag and to remove one. We also define a method to make a copy of the bag. Other operations of the `CBag` package are implemented without the knowledge of `CBag`'s internals: the union (merge) operator +=, and operators to compare CBags. These functions use only the public interface of the `CBag` class.

It has to be stressed that the package was designed to minimize the number of functions that need to know details of CBag's implementation. Following good practice, I wrote validation code that tests all the functions and methods of the CBag package and verifies common invariants.

Suppose you are tasked with implementing a Set package, an unordered collection where each element has a single occurrence. You have found my CBag package and realized that it can be re-used with few additional changes.

You too wrote a validation suite to test all `CSet` methods (newly defined as well as inherited from a bag) and to verify common expected properties, e.g., `a+=a` $\equiv$ `a`.

In my package, I have defined and implemented a function that, given three bags `a`, `b`, and `c`, decides if `a+b` is a subbag of `c`:

An insidious fault

function foo in CBag, version 1

```
bool foo(Cbag a, CBag b, CBag c):
  CBag ab = a.clone()
  ab += b
  return ab <= c
```

function foo in CBag, version 2

```
bool foo(CBag a, CBag b, CBag c):
  CBag ab = new CBag
  ab += a
  ab += b
  return ab <= c
```

[Cover the first half!] This is a pure function, which does not change its arguments. First I clone the argument a to avoid clobbering it, add to the clone all elements of bag b, and compare the resulting union with the bag c. The function was verified in the test suite. You have tried this function on sets, and found it satisfactory.

Later on, I revisited my code and found my implementation of `foo()` inefficient (in C++ that implementation is indeed inefficient). I *rewrote the function as this.* This function creates an empty bag first, adds to it all elements of bags a and b, and compares the result with c. It has exactly the same interface as the original `foo()`. The code hardly changed. The behavior of the new implementation is also the same – as far as I and the package CBag are concerned. Remember, I have no idea that you are re-using my package. I re-ran the validation test suite with the new `foo()`: everything tested fine.

However, when you run your code with the new implementation of `foo()`, you notice that something *has* changed! All of the sudden it returns different results. Quiz to the audience: can you tell where is the problem? The complete source code contains tests that you can run and see the difference for yourself. It is debatable which implementation of `foo()` gives truer results for CSets. In any case, changing internal algorithms of a pure function `foo()` while keeping the same interfaces is not supposed to break your code. What happened?

What makes this problem more unsettling is that both you and I tried to do everything by the book. We wrote a safe, typechecked code. We both wrote validation tests for our implementations of a CBag and a CSet, and they passed. And yet, despite all my efforts to separate interface and implementation, I failed.

Liskov Substitution Principle (LSP)

"If for each object `o1` of type `S` there is another object `o2` of type `T` such that for all programs `P` defined in terms of `T`, the behavior of `P` is unchanged when `o1` is substituted for `o2`, then `S` is a subtype of `T`."

As objects, a Set is not a subtype of a Bag

```
fnb (Bag bag):
  bag.put(5)
  bag.put(5)
  return bag.count(5)
ensure
  fnb(bag) = 2 + (old bag).count(5)
```

The breach of separation between CBag's implementation and interface is caused by CSet design's violating the Liskov Substitution Principle (LSP). CSet has been declared a subclass of CBag. Therefore, a compiler's typechecker permits passing a CSet object to a function that expects a CBag object. However, it is well known that a CSet is not a *subtype* of a CBag.

We have considered bags and sets from the OOP perspective – as objects that encapsulate state and behavior. Behavior means an object can accept a message, send a reply and possibly change its state. From this point of view, bags and sets are not subtypes of each other. Indeed, let us define a Bag as an object that accepts two messages: one that puts an element into the Bag, and another that gives the occurrence count for an element in the Bag. Likewise, a Set is defined as an object that accepts two similar messages. In case of sets, the count message is replyed with either 0 or 1.

Let us consider a function *fnb*. The behavior of this function, its contract, can be summed as follows: given a Bag, the function adds two elements into it and returns `(+ 2 (send orig-bag 'count 5))`. Technically you can pass to `fnb` a Set object as well. Just as a Bag, a Set object accepts messages `'put` and `'count`. However applying `fnb` to a Set object will break the function's post-condition stated above. Therefore, passing a set object where a bag was expected changes the behavior of a program. According to the LSP, a Set is not substitutable for a Bag – a Set cannot be a subtype of a Bag. Similarly, the paper shows that a Bag is not a subtype of a Set. The paper also shows that a Set is not a subtype of a Bag even if Set and Bag *objects* are immutable. Essentially, an object can override parent's behavior without altering the parent.

Mechanically preventing subclassing errors

We have seen:

- Bags and sets – *as objects* – are not subtypes of each other

- Subclassing them violates LSP $\Rightarrow$ insidious errors

- LSP for objects with overridable behavior is generally undecidable:
  `S.m()` overrides `T.m()`, `S` a subtype of `T` $\Rightarrow$
  (`T.m()` terminates $\Rightarrow$ `S.m()` terminates)

Let us summarize. We have seen that bags and sets – *considered as objects* – are not subtypes of each other. Subclassing them violates LSP, which leads to insidious errors. A violation of LSP for objects generally cannot be detected statically – furthermore, LSP for objects with overridable behavior is generally undecidable. Indeed, a mechanical application of LSP must at least be able to verify that all methods overridden in S terminate whenever the corresponding methods in T terminate. This is generally impossible.

Mechanically preventing subclassing errors

We affirm:

- A set, *as a pure value*, **is–a** bag. A set is substitutable for a bag, yet is still a set

- LSP for subset of values is trivially satisfied

- Given a statically checked rules, subclassing $\Rightarrow$ subtyping

- C++ contains a purely functional subset

Previously we have considered CBag and CSet as objects that encapsulate a hidden state – a collection of integers. The objects had an ability to react to messages, e.g., `put` and `del`, by altering their state. In this section we re-do our example using a different approach. We will take bags and sets as pure values, without any state or intrinsic behavior – just like the numbers are. Bags and sets will no longer have a state that is distinct from their identity and that can be altered. Equally important we do not allow any changes to the behavior of bags and sets with respect to applicable operations, by overriding or otherwise. In other words, every postcondition of a bag or a set constructor holds throughout the lifespan of the constructed values. This approach makes the subclassing problems and breach of encapsulation disappear. It turns out that a set is truly subtype of a bag; a set is substitutable for a bag, a set can always be manipulated as a bag, a set maintains every invariant of a bag – and it is still a set. We will also show that if we abide by statically checkable rules we obtain a guarantee that the subtle subclassing errors cannot occur in principle. The rules do not reduce the power of a language. In the remainder of this talk, we will show that all these assertions hold at the same time.

BRules

- no virtual methods or virtual inheritance

- no visible members or methods in any public data structure

- no mutations to public data structures
  - a strict form: no assignments or mutations whatsoever
  - a less strict form: no function may alter, directly or indirectly, any data it receives as arguments

Again, suppose I was given a task to implement a Bag – an unordered collection of possibly duplicate items (integers in this example). This time my boss laid out the rules, which we will refer to as *BRules*. The rules break the major tenets of OOP: for example, values no longer have a state that is separate from their identity. Prohibitions on virtual methods and on modifications of public objects are severe. It appears that not much of C++ is left. Surprisingly I still can implement my assignment without losing expressiveness – and perhaps even gaining some. **{**The exercise will also illustrate that C++ does indeed have a pure functional subset, and that you can program in C++ without assignments.**}**

Interface and implementation of a FBag

```
class FBag
  FBag:: () → FBag
  FBag:: FBag → FBag
  private collection:: List
```

```
count:: (FBag,int) → int
put:: (FBag,int)→ FBag
del:: (FBag,int)→ FBag
fold::(FBag,('a→'a),'a)→'a
remove_duplicates:: FBag → FBag
operator + :: (FBag,FBag) → FBag
```

Indeed, there are no overridable functions, no methods or public members. Only a constructor and a copy constructor are visible in a class. We also declare functions that take a FBag as (one of the) arguments and return the count of a *specific element* in the bag, fold over the bag, compare two bags for structural equivalence, merge two bags, add or delete an element. The latter three functions do not modify their arguments; they return a new FBag as their result. The three emphasized functions are privy to FBag's implementation – all others are not. It must be stressed that *these functions* are not FBag's methods; in particular, they are not a part of the class FBag, they are not inherited and they cannot be overridden. We have managed to implement bags without using assignments, in C++. The code that accompanies the article shows that this is indeed possible.

Following good practice, I wrote a validation code that tests all the functions of the FBag package and verifies common invariants.

Implementation of a FSet

```
class FSet extends FBag
  FSet: FBag()
  FSet(FBag bag): FBag(remove_duplicates(bag))

bool memberof(FSet set, int elem):
  return count(set,elem) > 0

-- That is all --
```

FSet is a *subtype* of a FBag

Suppose you are tasked with implementing a Set package. You have found my FBag package and realized that it can be used with few additional changes. Surprisingly, this is the *whole* implementation of a FSet. A set is fully a bag. Because FSet constructors eventually call FBag constructors and do no alter the latter's result, every post-condition of a FSet constructor implies a post-condition of a FBag constructor. Since FBag and FSet values are immutable, the post-conditions that hold at their birth remain true through their lifespan. Because all FSet values are created by an FBag constructor, all FBag operations automatically apply to an FSet value. This concludes the proof that an FSet is a *subtype* of a FBag.

The source code that accompanies the article has extensive validation test suite. It verifies that an FSet instance passes all the tests for FBags as well as its own. Despite the proof of the subtyping just given, the statement that an FSet is an FBag – and it is still an FSet – seems to cause a disbelief. Let us consider one example in more detail.

Indeed, an FSet is an FBag
– and it is still an FSet

```
     // an FSet is an FBag
int cntb(FBag v):
  FBag b1 = put(v, 5)
  FBag b2 = put(b1, 5)
  FBag b3 = del(b2, 5)
  return count(b3, 5)
int cb1 = cntb(new FBag) // cb1 will be 1
int cb2 = cntb(new FSet) // cb2 will be 1

     // An an FSet does act like a set
int cnts(FSet v):
  FSet s1 = put(v, 5)
  FSet s2 = put(s1, 5)
  FSet s3 = del(s2, 5)
  return count(s3, 5)
int cs = cnts(new FSet) // cs will be 0
```

This example is one of the test cases. Given a bag, we add two elements into that, remove one and return the count of the remaining elements. You can pass function cntb either an empty FBag or an empty FSet – *and the result will be the same*. Function cnts does the same procedure specifically for sets – and the result is consistent with our expectations. Indeed, an FSet is a FBag and is still an FSet. You can run the corresponding code and check the results for yourself. Yet it is puzzling: how come `cs` has the value different from that of `cb1` if there is no custom `del()` function for FSets? The statement `FSet s2 = put(s1, 5);` is the most illuminating. On the right-hand side is an expression: putting an element 5 to a FBag/FSet that already has this element in it. The result of that expression is a FBag {5,5}, with two instances of element 5. The statement then constructs a FSet `s2` from that bag. A FSet constructor is invoked. The constructor takes the bag {5,5}, removes the duplicate element 5 from it, and "blesses" the resulting FBag to be a FSet as well. Thus `s2` will be a FBag and a FSet, with one instance of element 5. In fact, `s1` and `s2` are identical. A FSet constructor guarantees that a FBag it constructs contains no duplicates. As objects are immutable, this invariant holds forever.

From an extensional point of view, a type denotes a set of values. These sets may overlap. Therefore, a value can be of several types. This is quite common: an empty list is an empty list of integers and an empty list of characters and an empty list of lists, etc. In our example, a collection without duplicates can be called both a bag and a set. Such a value can participate in operations that deal with sets and in operations that deal with bags.

Unlike CSets and CBags objects considered previously, FSets and FBags do not carry any behavior. FSets and FBags cannot react to messages – they are messages themselves. FSet is indeed a subtype of FBag, while CSet is not a subtype of a CBag as CSet has a different behavior. Incidentally LSP is trivially satisfied for values that do not carry their own behavior. FBags and FSets can be sent to various processes that take one message and emit another. The processes (procedures) can accept messages of several types – and a message can have several types at the same time. Thus polymorphism is easily possible without OOP. In the following we will show that abstract data types and more conventional polymorphic programming is possible when the BRules are in effect.

Polymorphic programming with BRules

An Abstract Data Type: Shape

```
data Shape = Shape {
          draws::     IO (),
          offset_by:: (Float,Float) -> Shape,
          set_dim::   (Float,Float) -> Shape }

data Shapes = Shapes (Shape, Shape, Shape)

pack:: Shapes -> Shapes
pack (Shapes (s1,s2,s3)) =
   Shapes ( offset_by s1 (0,1),
            offset_by s2 (2,3),
            offset_by s3 (4,5) )

resize (Shapes (s1,s2,s3)) =
    Shapes ( set_dim s1 (10,10),
             set_dim s2 (10,11),
             set_dim s3 (11,12) )
```

As an illustration, let's consider an existential type that is defined solely by its behavior. A *Shape* is an existential type that knows how to draw, move and resize itself. Here's a usage example: a list of shapes. For compatibility with some other examples, the list is a triple. We can define operations draw, pack and resize on the list that work no matter what concrete shapes are placed into the list. The example is a familiar one; its OOP implementation is well-known. This slide however shows an implementation with BRules in effect – written in a language where BRules are easily enforceable.

Two concrete shapes

```
rectangle (x,y,width,height) =
  if width == height then
     square(x,y,width)
  else Shape {
    draws = ...,
    offset_by = \ (new_x,new_y) ->
       rectangle (x+new_x,y+new_y,width,height),
    set_dim = \ (new_width,new_height) ->
       rectangle (x,y,new_width,new_height) }

square (x,y,size) = Shape {
    draws = ...,
    offset_by = \ (new_x,new_y) ->
        square (x+new_x,y+new_y,size),
    set_dim = \ (new_width,new_height) ->
        rectangle (x,y,new_width,new_height) }
```

We will now consider two concrete instances of the abstract Shape class – a rectangle and a square. The *constructor of the rectangle* can actually construct a square if the width and the height are the same. Also note that the set_dim method of a square returns a new rectangle – which may happen to be a square if the new dimensions are the same. The draw method is different in squares and rectangles. There are no assignments.

A sample test

```
main = do {
  let shapes = Shapes (rectangle (0,0,1,2),
                       rectangle (0,0,5,5),
                       rectangle (0,0,7,7))
      shapes_packed = pack shapes
      shapes_resized = resize shapes_packed
  in
    do { putStrLn "Drawing ...
           some rectangles are actually squares"
         draw shapes;
         putStrLn "Packing ...";
         draw shapes_packed;
         putStrLn "Resizing ...
           rectangles turn squares and vv";
         draw shapes_resized };
}
```

We initialize the list of shapes with rectangles. After we execute a draw operation, some of the rectangles will show up as squares. Although some of the rectangles in the list of shapes are actually squares, we can resize them anyway. All shapes can be resized. Depending on the new dimensions, a square can *become* a rectangle or a rectangle square. This exercise shows that BRules do not constrain the power of a language even when abstract data types are involved.

Conclusions

- OOP is prone to subclassing errors
- Subclassing errors cannot, in general, be detected statically
- BRules prevent subclassing errors by outlawing behavior overriding
- BRules can be enforced statically
- BRules do not diminish the power of the language

It is known, albeit not so well, that following the OOP letter and practice may lead to insidious errors. We have shown how subtle the errors can be even in simple cases. In theory, there are rules – LSP – that could prevent the errors. Alas, the rules are in general undecidable and not practically reinforceable.

In contrast, BRules introduced in this article can be statically checked at compile time. When BRules are in effect, the subtle subclassing errors become impossible. To be more precise, with BRules, subclassing implies subtyping. **{** Subclassing by definition is a way of creating (derived) values by extending, restricting, or otherwise specializing other, parent values. A derived value constructor must invoke a parent value constructor to produce the parent value. The former constructor often has a chance to alter the parent constructor's result before it is cast or incorporated into the derived value. If this chance is taken away, the post-condition of a derived value constructor implies the post-condition of the parent value. Disallowing any further mutations guarantees the behavioral substitutability of derived values for parent values at all times.**}**

As the examples in this article showed, following BRules does not diminish the power of the language. We can still benefit from polymorphism, we can still develop practically relevant code. Yet BRules blur the distinction between the identity and the state, a characteristic of objects. BRules are at odds with the practice if not the very mentality of OOP.

OOP considered harmful

OOP:

1. inflates statement's context

2. increases the gap lexical structure $\leftrightarrow$ dynamic behavior

3. limits design choices and overlooks alternatives

"While opinions concerning the benefits of OOSD [Object-Oriented Software Development] abound in OO literature, there is little empirical proof of its superiority"

Johnson, R.A. "The Ups and Downs of Object-Oriented Systems Development." Comm. ACM 43(10), October 2000, pp. 69-73.

Here's the conclusion you might've anticipated *[cover the rest of the slide but the title]*. I use this phrase not because I want to appear hip. I contend that OOP is to be considered harmful for exactly the same reasons Dijkstra objected to GOTO. Because both inflate the context we have to keep in mind to understand a phrase of code. In a OOP program the compiler is of little help to guarantee us that subclassing is always proper. We have to verify it ourselves at each class derivation – we have to keep the whole hierarchy in mind. Inheritance means that in order to understand the behavior of one object, I have to understand the behavior of all its ancestors first. This is a problem, as human (short-term) memory is rather limited.

Both GOTO and inheritance weaken the correspondence between the static structure of a program before programmer's eyes, and the dynamic behavior of the corresponding process. Both GOTO and OOP hinder the readability of the code. It becomes difficult for a developer to have a clear idea what a piece of code is doing. As the empirical evidence shows, this indeed makes OOP code more difficult to develop and debug, compared to other alternatives.

Finally, the very notion of an object as an entity with the identity, state, and behavior restricts a developer. Granted, the notion of object is sometimes useful – in simulation, for example, where it originated. As this talk has attempted to show, by thinking that everything is an object we miss opportunities. Indeed, how easy it was to derive a set from a bag once we realized that they don't have to be objects.

It is a good idea to end a controvertial talk by appealing to authorities. In our case, the authorities are well worth appealing to.

"My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible."

Edsger W. Dijkstra,
Communications of the ACM, Vol. 11, No. 3,
March 1968, pp. 147-148.

http://www.acm.org/classics/oct95/