

Subtyping

Edward Z. Yang

In today's lecture, we'll be delving into a ubiquitous and surprisingly subtle concept: subtyping.

Subtyping = relationship between interfaces
in contrast: Inheritance = relationship between implementation

What does it mean when we say that "ColorPoint is a subtype of Point"? When we declare a subtyping relationship, we are saying something about the interfaces of the two objects (not the implementations!)

e.g. ColorPoint is a subtype of Point

Specifically, we say that Colorpoint is a subtype of Point because it supports all of the interface of Point (and perhaps some of its own.) This means that anywhere we need a Point, we can use a ColorPoint instead.

If **interface** A contains all of
interface B, then A is a subtype of B


 the set of messages the object understands

ColorPoint is a subtype of Point
x y color **x y**

Subtyping in JavaScript/Smalltalk

In a language like JavaScript (or Smalltalk), the interfaces are implicit, in the sense that they are not recorded by any type system. Subtyping relationship is something implicitly understood by the developer, when they pass around objects with interfaces that are bigger than what is needed.

Objects **implicitly** have an interface.
set of understood messages

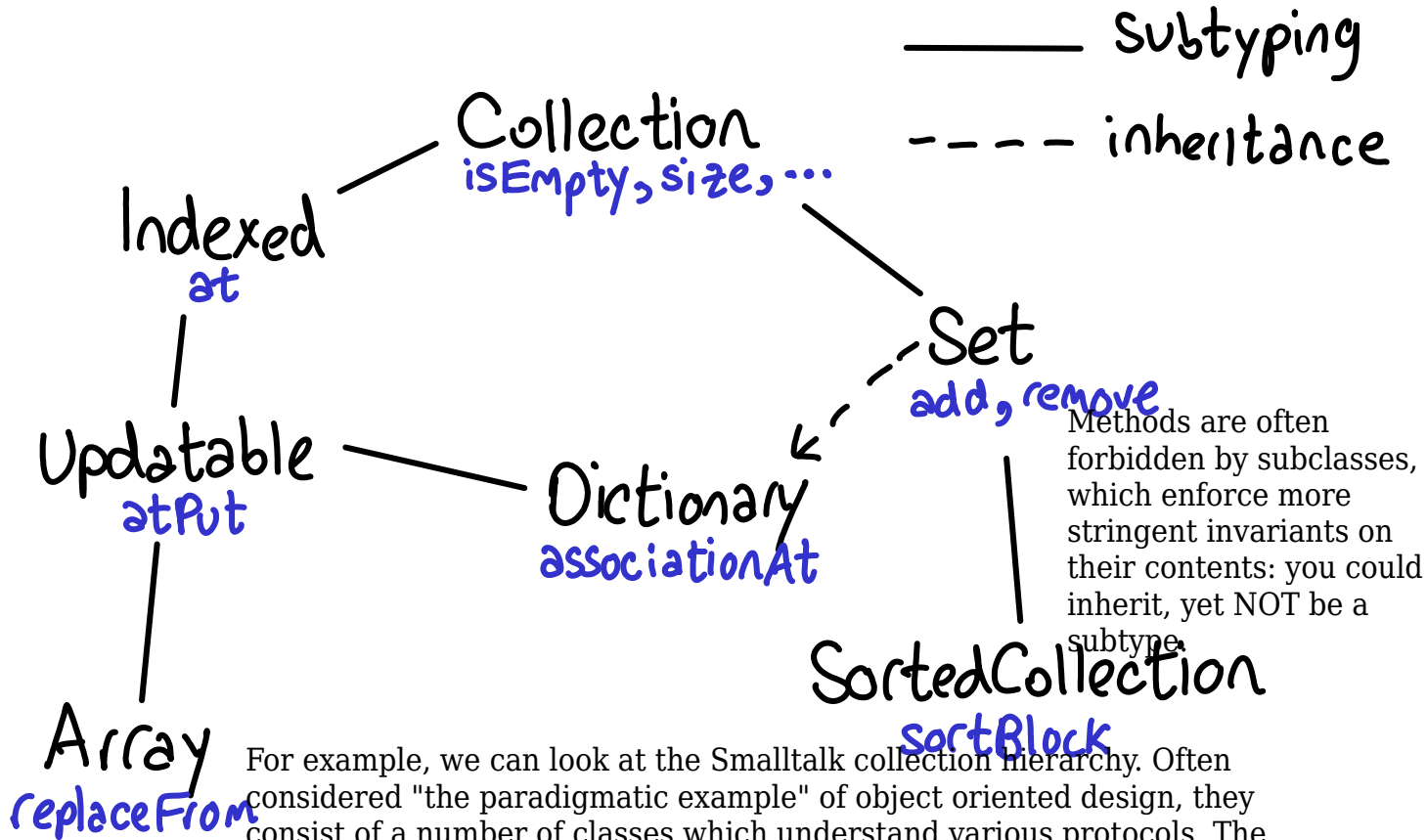
subtype of  Point { x: y: , x, y, draw }
ColorPoint { x: y: , x, y, draw, color }

If p msg:args OK, then q msg:args OK

No relationship to inheritance
can delete methods!

Duck typing, i.e., if it walks like a duck and talks like a duck, it is a duck, is applicable here. There doesn't need to be any source level connection between two objects for there to be a subtyping relationship.

Smalltalk collection hierarchy



For example, we can look at the Smalltalk collection hierarchy. Often considered "the paradigmatic example" of object oriented design, they consist of a number of classes which understand various protocols. The "protocols" (as they're called) Indexed/Updatable, aren't "inherited" by any of the classes: they simply specify methods which are understood to be implemented by objects that understood the protocol.

Subtyping in C++

Subtyping in C++ is a bit murkier: even if two classes define the same methods, there does not necessarily exist a subtyping relation between objects of the two classes; the relation only holds when A publicly inherits from B.


A is a subtype of B if A has a public base class B

(C++ does not do to the independence of subtyping and inheritance by admitting the possibility of private inheritance though!)

```
class Point {  
public:  
    virtual int getX();  
    virtual void move(int);  
private:  
    ...  
}
```

```
class ColorPoint {  
public:  
    virtual int getX();  
    virtual void move(int);  
    virtual int getColor();  
private:  
    ...  
}
```

not subtype of



Subtyping in C++

Semantics

Impl

In the case of C++, "interface" is a much more stringent concept of binary layout. So it makes sense that subtyping requires inheritance: otherwise, there isn't a way for the C++ compiler to ensure that the layouts (i.e. interfaces) are compatible.

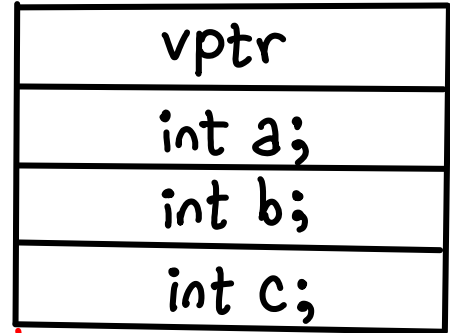
Interface is literally memory layout of objects

Remember, this is the tradeoff that C++ makes: fast vcalls for less flexibility in subtyping.

Inheritance

⇒ compatible memory layout

⇒ subtype relation



Subtype conversion = no-op
(or pointer offset)

Bigger?

So let's go back to our idea about subtyping. We stated that A is a subtype of B if A has a superset of B's interface. Does this seem backwards? (It's a *sub*type?)

If interface A contains all of interface B, then A is a subtype of B

the set of messages the object understands

ColorPoint is a subtype of Point
x y color x y

Smaller?

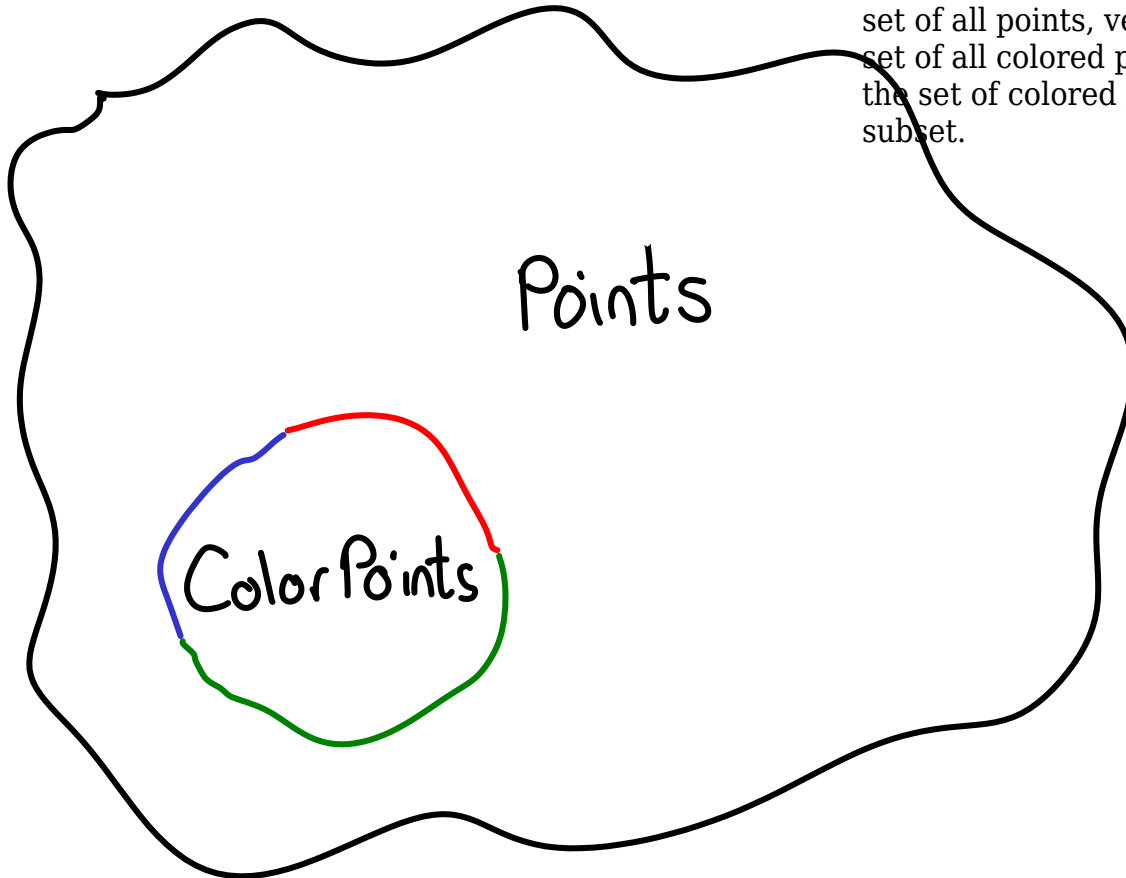
The sense of "subtype" comes from a different, related formulation, in which we treat types as "sets". For example, natural numbers are a subtype of integers because the natural numbers are a subset of integers.

If type A classifies a **set** of values that is a **subset** of type B , then A is a subtype of B

\mathbb{N} is a subtype of \mathbb{Z}
 $\{\emptyset, 1, 2, \dots\}$ $\{\dots -2, -1, \emptyset, 1, 2, \dots\}$

Viewpoints are compatible

Although these two interpretations seem to be in conflict with each other, they are actually compatible. In particular, if we consider the set of all points, versus the set of all colored points, clearly the set of colored points is a subset.



A useful, universal view on the definition of subtyping is the formulation below. The emphasis is on whether or not it can be "used".

S is a **subtype** of T if any term of type S can be used^{*} in a context where a term of type T is expected.

Why the asterisk? What it means for a term of type S to be "usable" is something of a philosophical question. The usual formulation is that it is usable if no type errors would occur if you used it in the context requiring T . The Liskov substitution principle asserts that the required invariant is something stronger, however!

So, to summarize, subtyping is a runtime phenomenon, in the sense that it describes when an object can be used in another context. But we can use static types to check if we've gotten it right!

Subtyping

- a runtime phenomenon...
- ... which can be captured in the static type system

So, let's talk a bit more about these static types.

Details

The way to read this slide: if an expression e has type S , and S is a subtype of T , then e ALSO has type T . This is called the "subsumption rule", and if you're talking about the type system of a language, if you add this rule you get subtyping.

S is a subtype of T
mnemonic: "is less than" \approx "sub"


$$e :: S$$
$$S <: T$$

$$e :: T$$

The subsumption rule

Type system designer's job:

Define $S <: T$

Since the typing rule is always the same, in order to say what the subtyping relationships in a language are, all you have to do is write down the subtyping relationship.

C++ example:

if A is a public subclass of B , $A <: B$

nothing else!

In some cases, the subtyping relation is very trivial; i.e., as in C++ the only way to get a subtype is by declaring subtyping relation. This form of subtyping is often called "nominal" because two interfaces are only related if *explicitly* declared so.

What is the form of a subtyping system where we can do it implicitly? We can discuss structural subtyping rules.

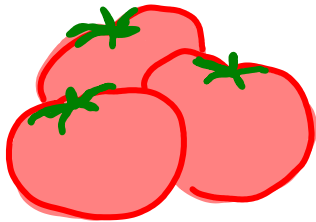
Here's a more precise type system...

Blackboard time! (The rules are recapped at the end of this slide deck)

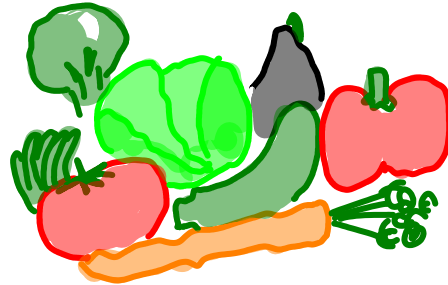
(why do you care? read on!)

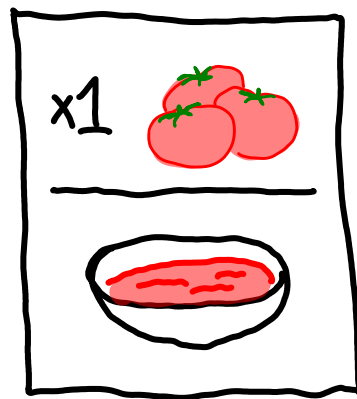
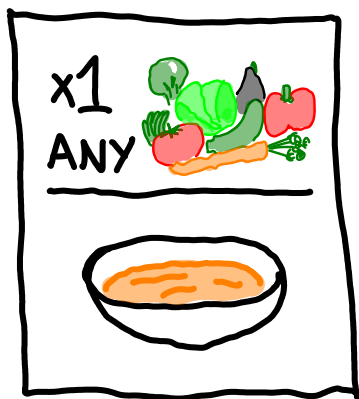
Some intuition for the function rules.

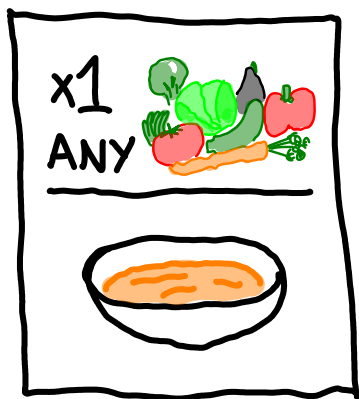
I blogged about this, see <http://blog.ezyang.com/2014/11/tomatoes-are-a-subtype-of-vegetables/>



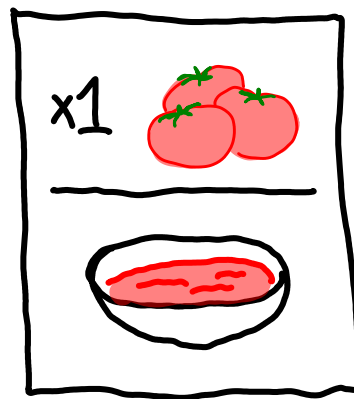
$<$



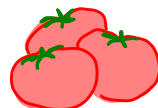


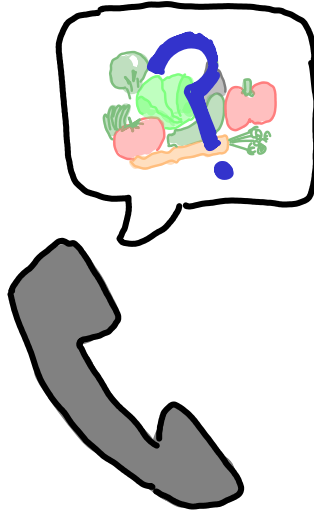


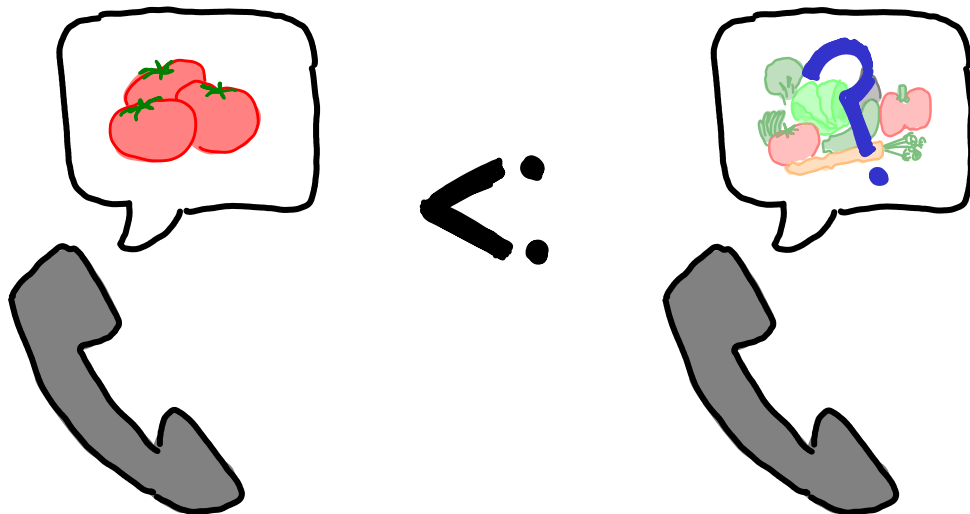
<:

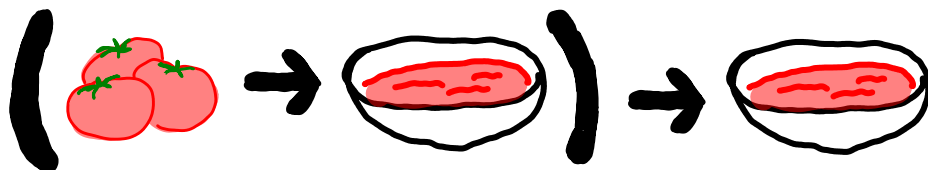


<:









<:



Discussion

```

class A {
public:
    bool equals(A&);
    A *clone(void);
}

```

```

μA.
{ equals : A → Bool
  clone  : () → A }

```

{ The structural typing rules I gave may seem a bit impoverished, because they only reflect "functional-ish" constructs (records, functions) rather than **objects** in which subtyping shows up frequently. But using records and **recursive types** (remember the μ) we can simulate objects. The basic idea is that an object is a record of functions, some of which take in/produce the very same object we're defining. The encoding is shown above.

This is very powerful, because it means we can use the structural rules we described above to start understanding the specific subtyping rules that are baked into the languages we know (one could say, the real life rules languages use are **generated** from these structural rules.)

Return covariance (C++/Java allowed!)

1992 1.5

```
class B: public A {
```

```
public:
```

```
    bool equals(A&);
```

```
    B *clone(void);
```

MB.

```
{ equals : A → Bool  
  clone : () → B }
```

```
}
```

Our first such example is that of return covariance. Return covariance states that if you want to override a virtual function in a subclass, it is OK to override the function so that it returns a more specific type (read subtype of the original return type). In the example above, B is permitted to define clone() as returning a B object; while this type does not exactly match the type of the parent class, it is a subtype of the parent class, and is perfectly acceptable in any case where we needed a clone on As (just get the B and upcast it to A). Return covariance can be quite convenient because if clone was forced to return A, then you'd have to define another clone function which returned Bs, if you wanted to avoid an unsafe downcast.

Invalid!

```
class B: public A {
```

```
public:
```

```
    bool equals(B&);
```

```
    B* clone(void);
```

```
}
```

MB.

```
{ equals : B → Bool  
  clone : () → B }
```

Something that beginning Java programmers ask for is the ability to also relax the type restrictions on 'equals' in the same way. Why doesn't this work?

Well, the input parameter to equals lives in **contravariant** position, so if B is a subtype of A, then it is A → Bool that is a subtype of B → Bool, not vice versa! Upon further reflection, this makes sense: the implementation of equals must be prepared for **any** object of type A to be passed in. This is one of the reasons why defining binary operations in OO languages is so miserable.

(why?!)

```
class A {
```

```
    public void f() throws IOException,  
                           IndexOutOfBoundsException;
```

```
}
```

Another rule adopted by Java is that an overriding method can be declared to throw *less* exceptions than the parent class. Once again, if we think *contextually*, it makes sense. `A::f()` can be used in any context which is ready to handle `IOException` and `IndexOutOfBoundsException`. Then clearly, `B::f()` which only throws `IOException`, will be fine in this context.

```
class B extends A {
```

```
    public void f() throws IOException;
```

```
}
```

This alludes to the subtyping rules for variants; we think of a function that throws exceptions as one that returns an error *variant type*, and the type explains what errors could be returned.

(rule for variants!)

Taste of generics (Java)

Array type

$$\frac{S <: T}{S[] <: T[]}$$

Let us preview some of the difficulties which we will study in more detail next lecture.

In Java, there exists an array type `S[]` which denotes an array of `S`s. Java specified the subtype rule above: if `S` is a subtype of `T`, then an array of `S` is a subtype of array of `T`.

Taste of generics (Java)

Array type

$$\frac{S <: T}{S[] <: T[]}$$

There is something deeply questionable about this rule, and it is shown by the code below, which will raise a type error exception AT RUNTIME.

Simply put, array covariance is unsound. It's not right!

Broken:

```
class A {}  
class B extends A {}  
B[] bArray = new B[10];  
A[] aArray = bArray;  
aArray[0] = new A;
```

Taste of generics (Java)

Array type

$$\frac{S <: T}{S[] <: T[]}$$

We can actually explain the unsoundness under the framework we have developed. Let us model an array as an object which supports two operation: getting elements from the array, and setting elements in the array.

In one case, S is in covariant position, while in another, S is in contravariant position. This means none of our subtyping rules apply for the record.

$$S[] = \{ \text{get} : \text{Int} \rightarrow S, \\ \text{put} : (\text{Int}, S) \rightarrow () \}$$

(also applies to references; recall `unsafePerformIO!`!)

Beyond subtyping

OK, let's talk a little philosophy

Beyond subtyping: Behavioural subtyping

Liskov substitution principle:

Let $\Phi(x)$ be a property provable about objects x of type T ; then $\Phi(y)$ should be true for objects y of type S where S is a subtype of T .

In 1987, Barbara Liskov proposed a stronger, *semantic* condition on subtyping, beyond the simple syntactic conditions we've described earlier in this lecture.

The definition takes a little unpacking, but intuitively...

What we want to do is somehow define subtyping in a way that avoids undesirable subtyping relationships like the one below. There isn't any semantic relationship between cowboys and pictures, and if we have a type system that permits us to use these interchangeably, this seems undesirable.

```
interface Cowboy {  
    void draw();  
}
```



```
interface Picture {  
    void draw();  
}
```

Example: Bags and Sets

[Kiselyov]

A bag is an unordered collection of possibly duplicate items.

Let's do a more complex example from Oleg Kiselyov.

```
class CBag {  
    public:  
        int size(void);  
        int count(int elem);  
        virtual void put(int elem);  
        virtual bool del(int elem);  
        virtual CBag *clone(void);  
        ...  
};
```

Example: Bags and Sets

[Kiselyov]

A set is an unorded collection where each element occurs once.

“A set is a bag with no duplicates.”

↑ inheritance?!

Example: Bags and Sets

[Kiselyov]

A set is an unordered collection where each element occurs once.

```
class CSet: public CBag {  
public:
```

```
    bool memberof(int elem) { return count(elem) > 0; }
```

```
    void put(int elem) {
```

```
        if (!memberof(elem)) CBag::put(elem); }
```

```
    CSet *clone(void) {
```

```
        CSet *new_set = new CSet;
```

```
        ...
```

Example: Bags and Sets

[Kiselyov]

A set is an unordered collection where each element occurs once.

```
bool foo(CBag &a, CBag &b, CBag &c) {  
    CBag &ab = *(a.clone());  
    ab += b;  
    bool result = ab <= c;  
    delete &ab;  
    return result; }  
  
foo({1}, {1}, {1,1}) == true
```

tests if the bag is
"contained in", e.g.

$\{1,2\} \leq \{1,2,2\}$

$\{1,1\} \neq \{1\}$

Example: Bags and Sets

[Kiselyov]

A set is an unordered collection where each element occurs once.

```
bool foo(CBag &a, CBag &b, CBag &c) {  
    CBag &ab = *(a.clone());  
    ab += b;  
    bool result = ab <= c;  
    delete &ab;  
    return result; }  
  
foo({1}, {1}, {1,1}) == true
```

tests if the bag is
"contained in", e.g.


$\{1,2\} \leq \{1,2,2\}$

$\{1,1\} \not\leq \{1\}$

Example: Bags and Sets

[Kiselyov]

A set is an unordered collection where each element occurs once.

```
bool foo2(CBag &a, CBag &b, CBag &c) {  
    CBag ab;    
    ab += a;  
    ab += b;  
    result = ab <= c;  
}
```

don't allocate on heap

Example: Bags and Sets

[Kiselyov]

Disaster!

$\text{foo}(\text{CSet}(1), \text{CSet}(1), \text{CSet}(1)) == \text{true}$

$\text{foo2}(\text{CSet}(1), \text{CSet}(1), \text{CSet}(1)) == \text{false}$

CSet not subtype of CBag

[Kiselyov]

↑ according to LSP

```
int addTwice(CBag &a) {  
    a.put(5);  
    a.put(5);  
    return a.count();  
}
```

violated by
CSet

// post-condition: returns 2+orig.count()

Moral:

- Inheritance for code reuse *rarely* corresponds to behavioral subtyping
- Behavioral subtyping is recoverable with *immutability* and *disallowing behavior overriding* (but one can hardly call this OO anymore)

Subtyping

$$\boxed{\tau_1 <: \tau_2}$$

$$\{l_1:\tau_1, \dots, l_i:\tau_i, \dots, l_j:\tau_j\} <: \{l_1:\tau_1, \dots, l_i:\tau_i\}$$

$$\frac{\tau_1 <: \tau_1' \quad \dots \quad \tau_i <: \tau_i'}{\{l_1:\tau_1, \dots, l_i:\tau_i\} <: \{l_1:\tau_1', \dots, l_i:\tau_i'\}}$$

$$\frac{\tau_1' <: \tau_1 \quad \tau_2 <: \tau_2'}{\tau_1 \rightarrow \tau_2 <: \tau_1' \rightarrow \tau_2'}$$

$$\frac{\tau_1 <: \tau_2 \Rightarrow \sigma(\tau_1) <: \pi(\tau_2)}{\mu\tau_1. \sigma(\tau_1) <: \mu\tau_2. \pi(\tau_2)}$$