CSC324 Lecture 19

Announcements

No lab on Monday: happy August long weekend!
(next year: come to the Heritage Days festival in Edmonton!)

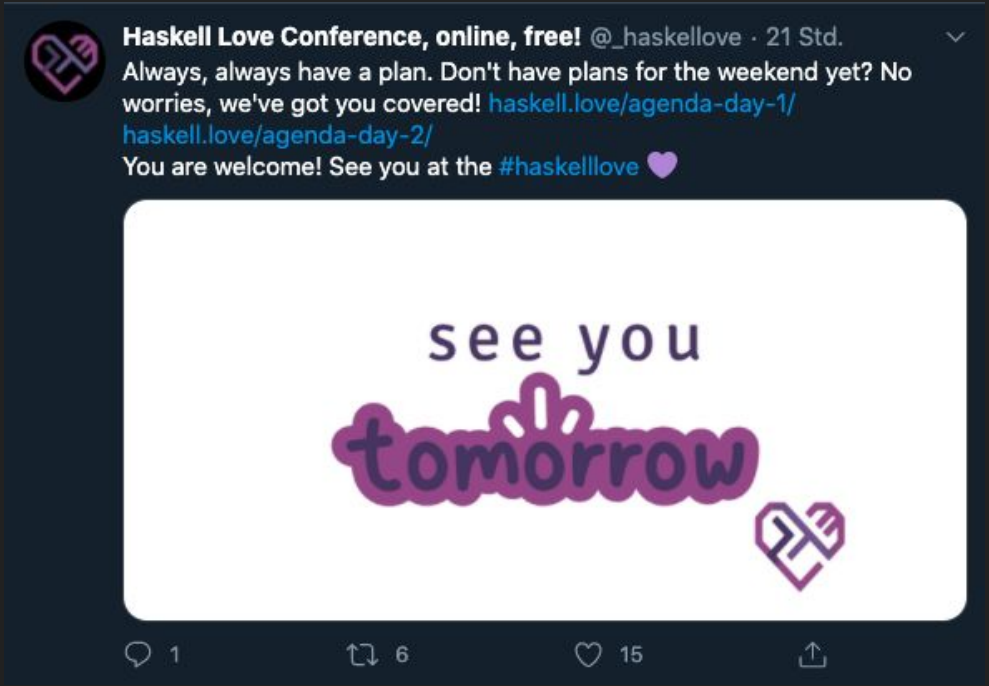
Extra office hours next week:

- Nick: 12:00-13:00 EDT Tuesday
- Victor: 14:00-15:00 EDT Tuesday

Free online Haskell conference this weekend

The talks have already started since it begun in the afternoon, European-time, but runs tomorrow, too.

Speakers look good and they invite beginner as well as advanced attendees too!



Today

- Subtyping rules, covariance, and contravariance
- Functors

Subtyping

We will wrap up our discussion of polymorphism with a brief discussion of OOP-style subtyping. You have seen throughout your programming career that values of a certain type can be used in an expression that clearly expects a different type.

Subtyping

We will wrap up our discussion of polymorphism with a brief discussion of OOP-style subtyping. You have seen throughout your programming career that values of a certain type can be used in an expression that clearly expects a different type.

Here's a straightforward example: an integer, when used as a float, "has its decimal point implicitly added to it"

```
[MSFT] ~ jshell
|  Welcome to JShell -- Version 9-internal
|  For an introduction type: /help intro


-> int i = 42
|  Added variable i of type int with initial value 42

-> float f = i
|  Added variable f of type float with initial value 42.0

->
```

Subtyping

We will wrap up our discussion of polymorphism with a brief discussion of OOP-style subtyping. You have seen throughout your programming career that values of a certain type can be used in an expression that clearly expects a different type.

In OOP, we can give a child class to something that merely expects the parent class

```
[MSFT] ~ jshell
|  Welcome to JShell -- Version 9-internal
|  For an introduction type: /help intro


-> /open Nat.java

-> Nat n1 = new Zero()
|  Added variable n1 of type Nat with initial value Zero

-> Nat n2 = new Add1(n1);
|  Added variable n2 of type Nat with initial value (Add1 Zero

->
```

The subtyping relation `<:`

Definition: We denote two types S and T to be **subtypes** of each other by `S <: T`.

`<:` can be mean "can be used in place of": if Dog is a subtype of Animal, a Dog object can be used in place of a less-specific Animal object, so Dog <: Animal.

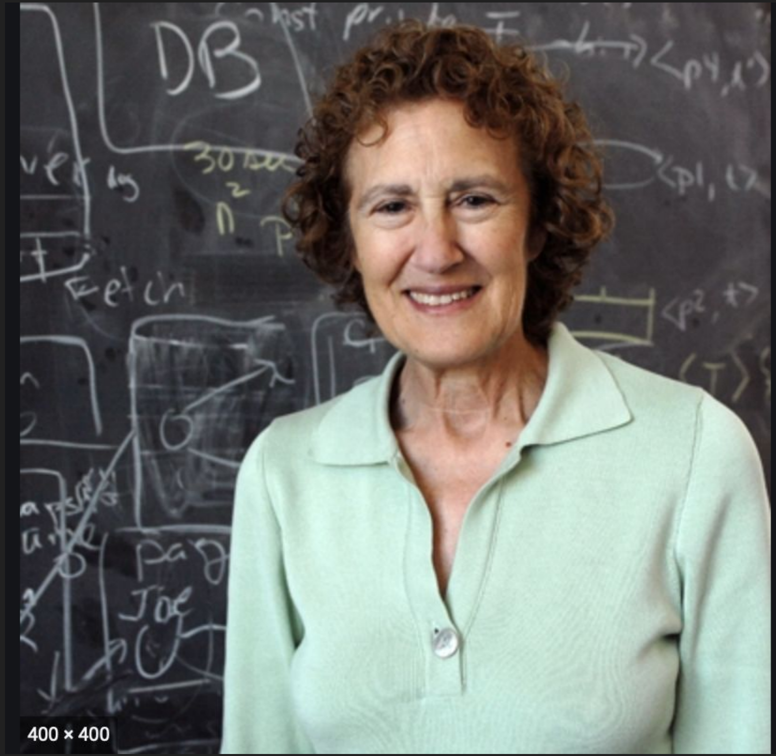
The subtyping relation `<:`

__Definition__: We denote two types S and T to be **subtypes** of each other by `S <: T`.

`<:` is a little bit like a subset relation: Consider Dog <: Animal; every Dog is an Animal, but there might be Animals that are not Dogs.

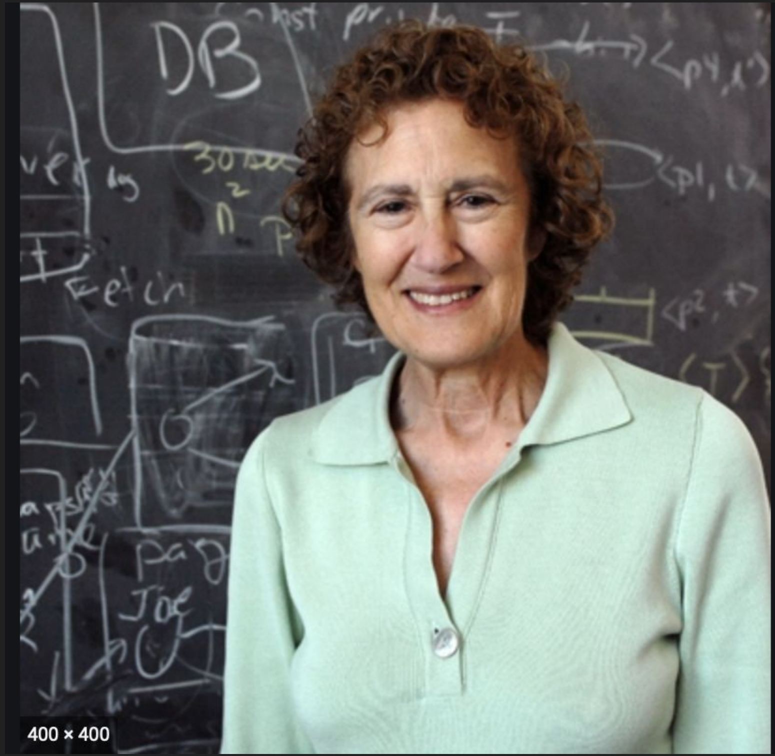
The Liskov Substitution Principle

The **Liskov Substitution Principle** states "Let φ(t) be a property provable about objects t of type T. Then φ(s) should be true for objects s of type S where `S <: T`."



The Liskov Substitution Principle

The **Liskov Substitution Principle** states “Let φ(t) be a property provable about objects t of type T. Then φ(s) should be true for objects s of type S where `S <: T`.”



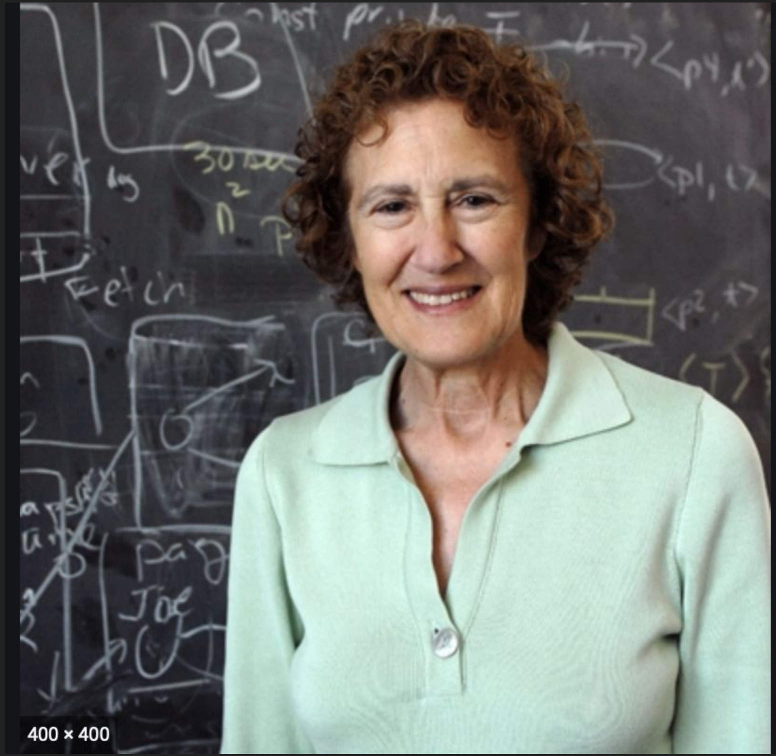
This is another way of saying "if S extends T, then S needs to be able to do everything that an T can do".

The Liskov Substitution Principle

The **Liskov Substitution Principle** states "Let φ(t) be a property provable about objects t of type T. Then φ(s) should be true for objects s of type S where `S <: T`."

This is another way of saying "if S extends T, then S needs to be able to do everything that an T can do".

Your favourite OOP tutorial example may in fact violate the LSP!!!



Some rules for subtyping

Subtyping is reflexive and transitive.

$$S <: S$$

$$\frac{S <: T \qquad T <: U}{S <: U}$$

Some rules for subtyping

Subtyping is reflexive and transitive.

$$S <: S$$

$$\frac{S <: T \qquad T <: U}{S <: U}$$

The **rule of subsumption:** This formalises the "subset" intuition: any time we please, we can treat a subtype term as if it was typed as a supertype.

$$\frac{t : S \qquad S <: T}{t : T}$$

The Top type

It is typical for a type system to have a type Top, denoted ⊤ (the upside-down version of the bottom type symbol), such that `T <: Top` for all other types in the type system.

Examples: in JavaScript and Java: `Object`, in Scala and Kotlin: `Any`

```
T <: Top
```

The Top and bottom types

We can also say something about Bottom w.r.t. subtyping: it is the subtype of every other type.

```
T <: Top
```

```
Bottom <: T
```

The Top and bottom types

What is the type of this entire expression (recalling that error terms typecheck to Bottom)?

```
if (x > 0)
    then x + 1
    else error "invalid!"
```

`T <: Top`

`Bottom <: T`

The Top and bottom types

What is the type of this entire expression (recalling that error terms typecheck to Bottom)?

```
if (x > 0)
    then x + 1              ; Num
    else error "invalid!"   ; Bottom
```

`T <: Top`

`Bottom <: T`

The Top and bottom types

What is the type of this entire expression (recalling that error terms typecheck to Bottom)?

```
if (x > 0)
    then x + 1               ; Num
    else error "invalid!"    ; Bottom
```

`T <: Top`

`Bottom <: T`

The Top and bottom types

What is the type of this entire expression (recalling that error terms typecheck to Bottom)?

```
if (x > 0)
    then x + 1              ; Num
    else error "invalid!"   ; Bottom
```

$$\frac{t : S \qquad S <: T}{t : T}$$

$$T <: Top$$

$$Bottom <: T$$

The Top and bottom types

What is the type of this entire expression (recalling that error terms typecheck to Bottom)?

```
if (x > 0)
    then x + 1                ; Num
    else error "invalid!"     ; Bottom <: Num
```

$$\frac{t : S \qquad S <: T}{t : T}$$

$$T <: Top$$

$$Bottom <: T$$

The Top and bottom types

What is the type of this entire expression (recalling that error terms typecheck to Bottom)?

```
if (x > 0)
    then x + 1                ; Num
    else error "invalid!"     ; Num
```

$$\frac{t : S \qquad S <: T}{t : T}$$

$$T <: Top$$

$$Bottom <: T$$

The Top and bottom types

Error terms typing to Bottom means their terms can be "subsumed" into the happy codepath, that will actually produce a valid value.

```
if (x > 0) ; whole expression: Num
    then x + 1              ; Num
    else error "invalid!"   ; Num
```

$$\frac{t : S \qquad S <: T}{t : T}$$

$$T <: Top$$

$$Bottom <: T$$

A polymorphic view of Bottom

An equivalent way to think about Bottom is that it's a polymorphic function that produces a type of … whatever the typechecker says it will produce.

```
Prelude> :t error
error :: [Char] -> a
Prelude>
```

A polymorphic view of Bottom

An equivalent way to think about Bottom is that it's a polymorphic function that produces a type of … whatever the typechecker says it will produce.

what type should be substituted for a to make this whole expression typecheck?

```
if (x > 0)
    then x + 1              ; Num
    else error "invalid!"   ; a
```

```
Prelude> :t error
error :: [Char] -> a
Prelude>
```

A polymorphic view of Bottom

An equivalent way to think about Bottom is that it's a polymorphic function that produces a type of … whatever the typechecker says it will produce.

```
Prelude> :t error
error :: [Char] -> a
Prelude>
```

what type should be substituted for `a` *to make this whole expression typecheck?*

```
if (x > 0) ; whole expression: Num
    then x + 1             ; Num
    else error "invalid!"  ; Num
```

Covariance and Contravariance

We should probably have a typing rule for how subtyping behaves with respect to arguments to functions, and values produced from function application.

$$\frac{\texttt{t1:T1} \qquad \texttt{t2:T2}}{\texttt{(}\lambda\texttt{ (t1) t2) : T1->T2}}$$

$$\frac{\texttt{f: T1->T2} \qquad \texttt{t:T1}}{\texttt{(f t) : T2}}$$

(Note: `Int "<:" Float ":<" Double` in Java)

Covariance and Contravariance

Let's say `foo()` is our function with type `S1->S1->S2`. Clearly it takes arguments of type `float`.

```
[MSFT] ~ jshell
|  Welcome to JShell -- Version 9-internal
|  For an introduction type: /help intro


-> float foo(float a, float b) {
>>     return a + b;
>> }
|  Added method foo(float,float)
```

We will use this example to build up the typing rule for subtyping function abstractions.

$$\frac{??? \qquad ???}{???}$$

(Note: `Int "<:" Float ":<" Double` in Java)

Covariance and Contravariance

Naturally, we can pass floats to this method and we get a float back.

Based on your intuition, what is another type of value that we can pass to `foo()`?

(Note: `Int "<:" Float ":<" Double` in Java)

```
[MSFT] ~ jshell
|  Welcome to JShell -- Version 9-internal
|  For an introduction type: /help intro


-> float foo(float a, float b) {
>>     return a + b;
>> }
|  Added method foo(float,float)
```

```
-> foo(3.0f,4.0f)
|  Expression value is: 7.0
|    assigned to temporary variable $2 of type float
```

$$\frac{???\qquad ???}{???}$$

Covariance and Contravariance

Naturally, we can pass floats to this method and we get a float back.

```
[MSFT] ~ jshell
|  Welcome to JShell -- Version 9-internal
|  For an introduction type: /help intro


-> float foo(float a, float b) {
>>     return a + b;
>> }
|  Added method foo(float,float)
```

```
-> foo(3,4)
|  Expression value is: 7.0
|    assigned to temporary variable $3 of type float
```

We understand that subtypes of `float`, like `ints`, can be arguments (this follows directly from the LSP)

How about a type that won't work?

$$\frac{T1 <: S1 \quad ???}{???}$$

(Note: `Int "<:" Float ":<" Double` in Java)

Covariance and Contravariance

Naturally, we can pass floats to this method and we get a float back.

We can't pass doubles to this method, though.

(Note: `Int "<:" Float ":<" Double` in Java)

```
[MSFT] ~ jshell
|  Welcome to JShell -- Version 9-internal
|  For an introduction type: /help intro


-> float foo(float a, float b) {
>>      return a + b;
>> }
|  Added method foo(float,float)
-> foo(3.0,4.0)
|  Error:
|  incompatible types: possible lossy conversion from double
to float
|  foo(3.0,4.0)
|      ^-^
```

$$\frac{T1 <: S1 \quad ???}{???}$$

Covariance and Contravariance

OK, so how do we encode this as a typing rule for the whole function?

Recalling `<:` to mean "can be substituted for", consider what function signatures we could substitute for foo's, such that it does the right thing for `Int` and `Floats`

(Note: `Int "<:" Float ":<" Double` in Java)

```
[MSFT] ~ jshell
|  Welcome to JShell -- Version 9-internal
|  For an introduction type: /help intro


-> float foo(float a, float b) {
>>     return a + b;
>> }
|  Added method foo(float,float)

-> foo(3,4)
|  Expression value is: 7.0
|    assigned to temporary variable $3 of type float
```

$$\frac{T1 <: S1 \qquad ???}{???}$$

Covariance and Contravariance

"If T1 is a subtype of S1, T1 is "less specific" than S1, but T1 can be used in place of S1."

A function that consumes the "more specific" S1 can always also consume the "more general" T1.

```
[MSFT] ~ jshell
|  Welcome to JShell -- Version 9-internal
|  For an introduction type: /help intro


-> float foo(float a, float b) {
>>     return a + b;
>> }
|  Added method foo(float,float)

-> foo(3,4)
|  Expression value is: 7.0
|    assigned to temporary variable $3 of type float
```

$$\frac{T1 <: S1 \quad ???}{S1\text{->}??? <: T1\text{->}???}$$

(Note: `Int "<:" Float ":<" Double` in Java)

Covariance and Contravariance

```
[MSFT] ~ jshell
|  Welcome to JShell -- Version 9-internal
|  For an introduction type: /help intro


-> float foo(float a, float b) {
>>     return a + b;
>> }
|  Added method foo(float,float)

-> foo(3,4)
|  Expression value is: 7.0
|    assigned to temporary variable $3 of type float
```

So, the relation of T1 and S1, in subtyping the function arguments, are *reversed!*

$$\frac{T1 <: S1 \quad ???}{S1 \text{->} ??? <: T1 \text{->} ???}$$

(Note: `Int "<:" Float ":<" Double` in Java)

Covariance and Contravariance

Definition: If a subtyping relation is reversed in a typing rule, we say it is **contravariant**.

```
[MSFT] ~ jshell
|  Welcome to JShell -- Version 9-internal
|  For an introduction type: /help intro


-> float foo(float a, float b) {
>>     return a + b;
>> }
|  Added method foo(float,float)

-> foo(3,4)
|  Expression value is: 7.0
|    assigned to temporary variable $3 of type float
```

$$\frac{T1 <: S1 \quad ???}{S1 \text{->} ??? <: T1 \text{->} ???}$$

(Note: `Int "<:" Float ":<" Double` in Java)

Covariance and Contravariance

Now, let's see what we can say about what this function produces.

As before: we will ask "what can be changed about the return value's type while still preserving `foo()`'s behaviour?"

(Note: `Int "<:" Float ":<" Double` in Java)

```
[MSFT] ~ jshell
|  Welcome to JShell -- Version 9-internal
|  For an introduction type: /help intro


-> float foo(float a, float b) {
>>     return a + b;
>> }
|  Added method foo(float,float)

-> foo(3,4)
|  Expression value is: 7.0
|    assigned to temporary variable $3 of type float
```

$$\frac{T1 <: S1 \quad ???}{S1 \text{->} ??? <: T1 \text{->} ???}$$

Covariance and Contravariance

We can certainly use the resulting value of calling foo() as a float, because that's exactly the return type.

We can also use it as a double, since `Float ":<" Double`.

(Note: `Int "<:" Float ":<" Double` in Java)

```
[MSFT] ~ jshell
|  Welcome to JShell -- Version 9-internal
|  For an introduction type: /help intro


-> float foo(float a, float b) {
>>     return a + b;
>> }
|  Added method foo(float,float)

-> float f = foo(1.0f, 2.0f);
|  Added variable f of type float with initial value 3.0

-> double d = foo(1.0f, 2.0f);
|  Added variable d of type double with initial value 3.0
```

$$\frac{T1 <: S1 \quad ???}{S1 \text{->} ??? <: T1 \text{->} ???}$$

Covariance and Contravariance

We can certainly use the resulting value of calling foo() as a float, because that's exactly the return type.

```
[MSFT] ~ jshell
|  Welcome to JShell -- Version 9-internal
|  For an introduction type: /help intro


-> float foo(float a, float b) {
>>     return a + b;
>> }
|  Added method foo(float,float)

-> float f = foo(1.0f, 2.0f);
|  Added variable f of type float with initial value 3.0

-> double d = foo(1.0f, 2.0f);
|  Added variable d of type double with initial value 3.0
```

We can also use it as a double, since `Float ":<" Double`.

(Note: `Int "<:" Float ":<" Double` in Java)

$$\frac{T1 <: S1 \qquad S2 <: T2}{S1 \text{->} S2 \;<:\; T1 \text{->} T2}$$

Covariance and Contravariance

In this case, the subtyping rule keeps the same ordering.

This is a **covariant** subtyping relation.

```
[MSFT] ~ jshell
|  Welcome to JShell -- Version 9-internal
|  For an introduction type: /help intro


-> float foo(float a, float b) {
>>     return a + b;
>> }
|  Added method foo(float,float)

-> float f = foo(1.0f, 2.0f);
|  Added variable f of type float with initial value 3.0

-> double d = foo(1.0f, 2.0f);
|  Added variable d of type double with initial value 3.0
```

$$\frac{T1 <: S1 \qquad S2 <: T2}{S1 \text{->} S2 <: T1 \text{->} T2}$$

(Note: `Int "<:" Float ":<" Double` in Java)

Covariance and Contravariance

One more covariance and contravariance example: arrays

```
-> class Parent {}
|  Added class Parent

-> class Child extends Parent {}
|  Added class Child
```

Covariance and Contravariance

One more covariance and contravariance example: arrays

```
-> class Parent {}
|  Added class Parent

-> class Child extends Parent {}
|  Added class Child

-> Parent p = new Child()
|  Added variable p of type Parent with initial value Child@4c70fda8
```

Clearly, `Child <: Parent`.

Covariance and Contravariance

One more covariance and contravariance example: arrays

```
-> class Parent {}
|  Added class Parent

-> class Child extends Parent {}
|  Added class Child

-> Child[] children = new Child[10];
|  Added variable children of type Child[] with initial value [LChild;@39c0f4a
```

If `Child <: Parent`, can we say anything about the subtyping between a `[Child]` and a `[Parent]`?

Covariance and Contravariance

One more covariance and contravariance example: arrays

If `Child <: Parent`, can we say anything about the subtyping between a `[Child]` and a `[Parent]`?

Suggestion: think about what the Top type (`Object`) would do!

```
-> class Parent {}
|  Added class Parent

-> class Child extends Parent {}
|  Added class Child

-> Child[] children = new Child[10];
|  Added variable children of type Child[] with initial value [LChild;@39c0f4a

-> Object[] whoknows = children
|  Added variable whoknows of type Object[] with initial value [LChild;@39c0f4a
```

Covariance and Contravariance

One more covariance and contravariance example: arrays

```
-> class Parent {}
|  Added class Parent

-> class Child extends Parent {}
|  Added class Child

-> Child[] children = new Child[10];
|  Added variable children of type Child[] with initial value [LChild;@39c0f4a

-> Object[] whoknows = children
|  Added variable whoknows of type Object[] with initial value [LChild;@39c0f4a

-> Parent[] parents = children
|  Added variable parents of type Parent[] with initial value [LChild;@39c0f4a
```

If `Child <: Parent`, can we say anything about the subtyping between a `[Child]` and a `[Parent]`?

We see `Child[] <: Parent[]`.

Covariance and Contravariance

Therefore, in general:

If `S <: T`, then

```
-> Parent p = new Child()
|  Added variable p of type Parent with initial value Child@4c70fda8
```

`[S] <: [T]`

```
-> Parent[] ps = new Child[10]
|  Added variable ps of type Parent[] with initial value [LChild;@1bce4f0a
```

Array subtyping is **covariant.**

A danger with mutable covariant arrays!

In Java, Number is an abstract class that all (boxed) numeric ty inherit from. So,

- `Integer <: Number`
- `Double <: Number`

```
-> Integer[] is = new Integer[]{1,2,3};
|  Added variable is of type Integer[] with initial value [Ljava.lang.Integer;@2
b98378d

-> Number[] ns = is
|  Added variable ns of type Number[] with initial value [Ljava.lang.Integer;@2b
98378d
```

A danger with mutable covariant arrays!

In Java, Number is an abstract class that all (boxed) numeric ty inherit from. So,

```
-> Integer[] is = new Integer[]{1,2,3};
|  Added variable is of type Integer[] with initial value [Ljava.lang.Integer;@2
b98378d

-> Number[] ns = is
|  Added variable ns of type Number[] with initial value [Ljava.lang.Integer;@2b
98378d
```

- `Integer <: Number`
- `Double <: Number`

Of course, "under the hood", ns is an array of integers, but we've allowed us to forget that through the rule of subsumption.

A danger with mutable covariant arrays!

In Java, Number is an abstract class that all (boxed) numeric ty inherit from. So,

- `Integer <: Number`
- `Double <: Number`

```
-> Integer[] is = new Integer[]{1,2,3};
|  Added variable is of type Integer[] with initial value [Ljava.lang.Integer;@2
b98378d

-> Number[] ns = is
|  Added variable ns of type Number[] with initial value [Ljava.lang.Integer;@2b
98378d

-> ns[0] = 42
|  Expression value is: 42
|    assigned to temporary variable $3 of type Number
```

Because `Integer <: Number`, assigning an integer into a `Number[]` is well-typed...

A danger with mutable covariant arrays!

In Java, Number is an abstract class that all (boxed) numeric ty inherit from. So,

- `Integer <: Number`
- `Double <: Number`

```
-> Integer[] is = new Integer[]{1,2,3};
|  Added variable is of type Integer[] with initial value [Ljava.lang.Integer;@2
b98378d

-> Number[] ns = is
|  Added variable ns of type Number[] with initial value [Ljava.lang.Integer;@2b
98378d

-> ns[0] = 42
|  Expression value is: 42
|    assigned to temporary variable $3 of type Number
```

Because `Double <: Number`, assigning a double into a `Number[]` is well-typed...

A danger with mutable covariant arrays!

In Java, Number is an abstract class that all (boxed) numeric ty inherit from. So,

- `Integer <: Number`
- `Double <: Number`

```
-> Integer[] is = new Integer[]{1,2,3};
|  Added variable is of type Integer[] with initial value [Ljava.lang.Integer;@2
b98378d

-> Number[] ns = is
|  Added variable ns of type Number[] with initial value [Ljava.lang.Integer;@2b
98378d

-> ns[0] = 42
|  Expression value is: 42
|    assigned to temporary variable $3 of type Number

-> ns[0] = 42.0
|  java.lang.ArrayStoreException thrown: java.lang.Double
|        at (#12:1)
```

Because `Double <: Number`, assigning a double into a `Number[]` is well-typed…

… but throws a runtime exception, because the underlying array can't hold one!!

A danger with mutable covariant arrays!

In Java, Number is an abstract class that all (boxed) numeric ty inherit from. So,

- `Integer <: Number`
- `Double <: Number`

```
-> Integer[] is = new Integer[]{1,2,3};
|  Added variable is of type Integer[] with initial value [Ljava.lang.Integer;@2
b98378d

-> Number[] ns = is
|  Added variable ns of type Number[] with initial value [Ljava.lang.Integer;@2b
98378d

-> ns[0] = 42
|  Expression value is: 42
|    assigned to temporary variable $3 of type Number

-> ns[0] = 42.0
|  java.lang.ArrayStoreException thrown: java.lang.Double
|        at (#12:1)
```

In the Java community, it is generally understood that making arrays covariant was probably a bad design decision.

A danger with mutable covariant arrays!

In Java, Number is an abstract class that all (boxed) numeric types inherit from. So,

- `Integer <: Number`
- `Double <: Number`

```
-> ArrayList<Integer> is = new ArrayList<Integer>()
|  Added variable is of type ArrayList<Integer> with initial value []

-> ArrayList<Number> ns = is
|  Error:
|  incompatible types: java.util.ArrayList<java.lang.Integer> cannot be converted
to java.util.ArrayList<java.lang.Number>
|  ArrayList<Number> ns = is;
|                         ^^
```

When they implemented parametric polymorphism later on, they decided to not have any **implicit** subtyping rule between a type parameter and a generic class.

(You can still do it explicitly; look up Java wildcards if you want to know more.)

Computational contexts

By now, we are very familiar with types like List and Option. In particular, we know that the `Some` value constructor wraps a value of type a, and List wraps any number of elements of type `a`.

We also saw a function `map` that, for every value in an Option or List, applies a function to that element.

```haskell
data Option a = None
              | Some a
              deriving (Show)
optMap :: (a -> b) -> Option a -> Option b
optMap _ None = None
optMap f (Some x) = Some (f x)

data List a = Empty
            | Cons a (List a)
listMap :: (a -> b) -> List a -> List b
listMap _ Empty = Empty
listMap f (Cons x xs) = (Cons (f x) (listMap f xs))
```

Computational contexts

(Also, by now, you know how to give the `optMap` and `listMap` functions the same name, by putting them in a typeclass whose instances need a function called `map`. We'll get there, but for the moment, we'll just refer to these functions collectively as "map".)

```
data Option a = None
              | Some a
              deriving (Show)
optMap :: (a -> b) -> Option a -> Option b
optMap _ None = None
optMap f (Some x) = Some (f x)

data List a = Empty
            | Cons a (List a)
listMap :: (a -> b) -> List a -> List b
listMap _ Empty = Empty
listMap f (Cons x xs) = (Cons (f x) (listMap f xs))


```

Computational contexts

In this "data-oriented" view, these types are like containers that, respectively, hold zero or one element, or, any number of elements, and `map` changes the values inside those containers.

Does it makes sense to talk about "mapping over" a datatype that doesn't wrap values like a container?

```haskell
data Option a = None
              | Some a
              deriving (Show)
optMap :: (a -> b) -> Option a -> Option b
optMap _ None = None
optMap f (Some x) = Some (f x)

data List a = Empty
            | Cons a (List a)
listMap :: (a -> b) -> List a -> List b
listMap _ Empty = Empty
listMap f (Cons x xs) = (Cons (f x) (listMap f xs))
```

Computational contexts

In order to answer this, let's look at some facts about each of these map functions.

These datatypes and their map functions each have certain rules associated with them, and they're consistent between the two:

```
data Option a = None
              | Some a
              deriving (Show)
optMap :: (a -> b) -> Option a -> Option b
optMap _ None = None
optMap f (Some x) = Some (f x)

data List a = Empty
            | Cons a (List a)
listMap :: (a -> b) -> List a -> List b
listMap _ Empty = Empty
listMap f (Cons x xs) = (Cons (f x) (listMap f xs))
```

Computational contexts: the neutral operation

1) There is a **neutral operation**, which we can call **id,** in the Option and List type, such that that value, when mapped over, is left unchanged (it has no **effect**)

```haskell
data Option a = None
              | Some a
              deriving (Show)
optMap :: (a -> b) -> Option a -> Option b
optMap _ None = None
optMap f (Some x) = Some (f x)

data List a = Empty
            | Cons a (List a)
listMap :: (a -> b) -> List a -> List b
listMap _ Empty = Empty
listMap f (Cons x xs) = (Cons (f x) (listMap f xs))
```

In Haskell,

```haskell
id :: a -> a
id x = \ x -> x
```

1) map id = id

Computational contexts: composition

How do we describe this expression in words?

```
> map (*2) (map (+1) [1,2,3])
[4,6,8]
```

"Every element in the list gets 1 added to it, then every element in the list gets 2 multiplied to it."

```haskell
data Option a = None
              | Some a
              deriving (Show)
optMap :: (a -> b) -> Option a -> Option b
optMap _ None = None
optMap f (Some x) = Some (f x)

data List a = Empty
            | Cons a (List a)
listMap :: (a -> b) -> List a -> List b
listMap _ Empty = Empty
listMap f (Cons x xs) = (Cons (f x) (listMap f xs))
```

1) map id = id

Computational contexts: composition

How do we describe this expression in words?

dot means function composition!

```
> (map (*2) . map (+1)) [1,2,3]
[4,6,8]
```

"Every element in the list gets 1 added to it, then every element in the list gets 2 multiplied to it."

```
data Option a = None
              | Some a
              deriving (Show)
optMap :: (a -> b) -> Option a -> Option b
optMap _ None = None
optMap f (Some x) = Some (f x)

data List a = Empty
            | Cons a (List a)
listMap :: (a -> b) -> List a -> List b
listMap _ Empty = Empty
listMap f (Cons x xs) = (Cons (f x) (listMap f xs))


```

1) map id = id

Computational contexts: composition

How do we describe this expression in words?

```
> map (\x -> (2 * (1 + x)) [1,2,3]
[4,6,8]
```

"Every element in the list gets 1 added, then 2 multiplied to it."

```haskell
data Option a = None
              | Some a
              deriving (Show)
optMap :: (a -> b) -> Option a -> Option b
optMap _ None = None
optMap f (Some x) = Some (f x)

data List a = Empty
            | Cons a (List a)
listMap :: (a -> b) -> List a -> List b
listMap _ Empty = Empty
listMap f (Cons x xs) = (Cons (f x) (listMap f xs))


```

1) map id = id

Computational contexts: composition

How do we describe this expression in words?

dot means function composition!

```
> map ((*2) . (+1)) [1,2,3]
[4,6,8]
```

"Every element in the list gets 1 added, then 2 multiplied to it."

```haskell
data Option a = None
              | Some a
              deriving (Show)
optMap :: (a -> b) -> Option a -> Option b
optMap _ None = None
optMap f (Some x) = Some (f x)

data List a = Empty
            | Cons a (List a)
listMap :: (a -> b) -> List a -> List b
listMap _ Empty = Empty
listMap f (Cons x xs) = (Cons (f x) (listMap f xs))


```

1) map id = id
2) (map g) . (map f) x = map (g . f) x

Computational contexts: composition

The point of Rule 2 is to say that map **preserves context:** mapping over something inside a List still produces a List, and mapping over something in an Option always produces an Option.

```
data Option a = None
              | Some a
              deriving (Show)
optMap :: (a -> b) -> Option a -> Option b
optMap _ None = None
optMap f (Some x) = Some (f x)

data List a = Empty
            | Cons a (List a)
listMap :: (a -> b) -> List a -> List b
listMap _ Empty = Empty
listMap f (Cons x xs) = (Cons (f x) (listMap f xs))
```

1) map id = id
2) (map g) . (map f) x = map (g . f) x

Two intuitions:

"Map takes a function and a container and changes the value inside the container"

"Map applies a value to a function within some **computational context**, changing the value inside but leaving the outside context undisturbed."

```
data Option a = None
              | Some a
              deriving (Show)
optMap :: (a -> b) -> Option a -> Option b
optMap _ None = None
optMap f (Some x) = Some (f x)

data List a = Empty
            | Cons a (List a)
listMap :: (a -> b) -> List a -> List b
listMap _ Empty = Empty
listMap f (Cons x xs) = (Cons (f x) (listMap f xs))
```

1) map id = id
2) (map g) . (map f) x = map (g . f) x

Two intuitions of functors:

A type that satisfies the two properties on the right are said to be **functors.**

If you like, you can think of a functor as a type that implements a `Mappable` interface (or, in our case, is an instance of a typeclass containing a `map` function!)

```haskell
data Option a = None
              | Some a
              deriving (Show)
optMap :: (a -> b) -> Option a -> Option b
optMap _ None = None
optMap f (Some x) = Some (f x)

instance Functor Option where
    fmap = optMap

data List a = Empty
            | Cons a (List a)
listMap :: (a -> b) -> List a -> List b
listMap _ Empty = Empty
listMap f (Cons x xs) = (Cons (f x) (listMap f xs))

instance Functor List where
    fmap = listMap

```

1) map id = id
2) (map g) . (map f) x = map (g . f) x

Lifting computation

Suppose we have some function `f :: a -> b`. What can we say about the curried function `fmap f`?

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

```
fmap f :: Functor f => f a -> f b
```

Partially-applying (ie. currying) f to fmap **lifts** the computation appled to an a such that it now executes within the functor's context.

```haskell
data Option a = None
              | Some a
              deriving (Show)
optMap :: (a -> b) -> Option a -> Option b
optMap _ None = None
optMap f (Some x) = Some (f x)

instance Functor Option where
    fmap = optMap

data List a = Empty
            | Cons a (List a)
listMap :: (a -> b) -> List a -> List b
listMap _ Empty = Empty
listMap f (Cons x xs) = (Cons (f x) (listMap f xs))

instance Functor List where
    fmap = listMap

```

1) map id = id
2) (map g) . (map f) x = map (g . f) x

why do we care

~~Nathan enjoys making you suffer with weird nonsensical math, why didn't I just take a French class this semester~~

We'll see that this broader value/context definition **is more general;** it lets us describe both data structures like List but also other kinds of computation that we'll see in the next few classes.

```haskell
data Option a = None
              | Some a
              deriving (Show)
optMap :: (a -> b) -> Option a -> Option b
optMap _ None = None
optMap f (Some x) = Some (f x)

instance Functor Option where
    fmap = optMap

data List a = Empty
            | Cons a (List a)
listMap :: (a -> b) -> List a -> List b
listMap _ Empty = Empty
listMap f (Cons x xs) = (Cons (f x) (listMap f xs))

instance Functor List where
    fmap = listMap

```

1) map id = id
2) (map g) . (map f) x = map (g . f) x

Our first Functors

By making these data types instances of Functor, we can apply fmap to them like any other functor in Haskell.

```haskell
data Option a = None
              | Some a
              deriving (Show)
optMap :: (a -> b) -> Option a -> Option b
optMap _ None = None
optMap f (Some x) = Some (f x)

instance Functor Option where
    fmap = optMap

data List a = Empty
            | Cons a (List a)
listMap :: (a -> b) -> List a -> List b
listMap _ Empty = Empty
listMap f (Cons x xs) = (Cons (f x) (listMap f xs))

instance Functor List where
    fmap = listMap

```