



## Overloading and Subtyping

Rob Sison  
UNSW  
Term 3 2024

# myExperience

The myExperience survey is out.

I would be very grateful if you could take 5 minutes out of your day to go on Moodle and fill out the survey. Even if you don't have much to say.

## Motivation

Suppose we added `Float` to `MinHS`.

Ideally, the arithmetic operations should be able to work on both `Int` and `Float`.

$4 + 6 :: \text{Int}$

$4.3 + 5.1 :: \text{Float}$

Similarly, a numeric literal should take on whatever type is inferred from context.

$(5 :: \text{Int}) \bmod 3$

$\sin(5 :: \text{Float})$

# Without Overloading

We effectively have two functions:

$$(+_{\text{Int}}) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$$
$$(+_{\text{Float}}) :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float}$$

# Without Overloading

We effectively have two functions:

$$(+_{\text{Int}}) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$$
$$(+_{\text{Float}}) :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float}$$

We would like to refer to both of these functions by the **same name** and have the specific implementation chosen based on the **type**.

## Without Overloading

We effectively have two functions:

$$(+_{\text{Int}}) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$$
$$(+_{\text{Float}}) :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float}$$

We would like to refer to both of these functions by the **same name** and have the specific implementation chosen based on the **type**.

Such type-directed name resolution is called *ad-hoc polymorphism* or *overloading*.

# Type Classes

Type classes are a common approach to ad-hoc polymorphism, and exist in various languages under different names:

- Type Classes in Haskell
- Traits in Rust
- Implicits in Scala
- Protocols in Swift
- Contracts in Go 2
- Concepts in C++
- Other languages approximate with *subtype polymorphism* (coming)

# Type Classes

A *type class* is a *set of types* for which implementations (*instances*) have been provided for various functions, called *methods*<sup>1</sup>.

---

<sup>1</sup>Nothing to do with OO methods.



# Type Classes

A *type class* is a *set of types* for which implementations (*instances*) have been provided for various functions, called *methods*<sup>1</sup>.

## Example (Numeric Types)

In Haskell, the types `Int`, `Float`, `Double` etc. are all instances of the type class `Num`, which has methods such as `(+)`, `negate`, etc.

---

<sup>1</sup>Nothing to do with OO methods.

# Type Classes

A *type class* is a *set of types* for which implementations (*instances*) have been provided for various functions, called *methods*<sup>1</sup>.

## Example (Numeric Types)

In Haskell, the types `Int`, `Float`, `Double` etc. are all instances of the type class `Num`, which has methods such as `(+)`, `negate`, etc.

## Example (Equality)

In Haskell, the `Eq` type class contains methods `(==)` and `(/=)` for computable equality. What types cannot be an instance of `Eq`?

---

<sup>1</sup>Nothing to do with OO methods.

# Notation

We write:

$$f :: \forall a. P \Rightarrow \tau$$

To indicate that  $f$  has the type  $\tau$  where  $a$  can be instantiated to any type **under the condition** that the constraint  $P$  is satisfied.

Typically,  $P$  is a list of *instance constraints*, such as  $\text{Num } a$  or  $\text{Eq } b$ .

# Notation

We write:

$$f :: \forall a. P \Rightarrow \tau$$

To indicate that  $f$  has the type  $\tau$  where  $a$  can be instantiated to any type **under the condition** that the constraint  $P$  is satisfied. Typically,  $P$  is a list of *instance constraints*, such as `Num a` or `Eq b`.

## Example

- $(+) :: \forall a. (\text{Num } a) \Rightarrow a \rightarrow a \rightarrow a$
- $(==) :: \forall a. (\text{Eq } a) \Rightarrow a \rightarrow a \rightarrow \text{Bool}$

Is  $(1 :: \text{Int}) + 4.4$  a well-typed expression?

# Notation

We write:

$$f :: \forall a. P \Rightarrow \tau$$

To indicate that  $f$  has the type  $\tau$  where  $a$  can be instantiated to any type **under the condition** that the constraint  $P$  is satisfied. Typically,  $P$  is a list of **instance constraints**, such as `Num  $a$`  or `Eq  $b$` .

## Example

- $(+) :: \forall a. (\text{Num } a) \Rightarrow a \rightarrow a \rightarrow a$
- $(==) :: \forall a. (\text{Eq } a) \Rightarrow a \rightarrow a \rightarrow \text{Bool}$

Is  $(1 :: \text{Int}) + 4.4$  a well-typed expression?

**No.** The type of  $(+)$  requires its arguments to have the same type.

# Extending MinHS

Extending implicitly typed MinHS with type classes:

Predicates  $P ::= C \tau$

Polytypes  $\pi ::= \tau \mid \forall a. \pi \mid P \Rightarrow \pi$

Monotypes  $\tau ::= \text{Int} \mid \text{Bool} \mid \tau + \tau \mid \dots$

Class names  $C$

# Extending MinHS

Extending implicitly typed MinHS with type classes:

Predicates  $P ::= C \tau$   
 Polytypes  $\pi ::= \tau \mid \forall a. \pi \mid P \Rightarrow \pi$   
 Monotypes  $\tau ::= \text{Int} \mid \text{Bool} \mid \tau + \tau \mid \dots$   
 Class names  $C$

Our typing judgement  $\Gamma \vdash e : \pi$  now includes a set of type class **axiom schema**:

$$\mathcal{A} \mid \Gamma \vdash e : \pi$$

This set contains predicates for all known type class instances.

# Typing Rules

The existing rules now just thread  $\mathcal{A}$  through.



# Typing Rules

The existing rules now just thread  $\mathcal{A}$  through.

To use an overloaded type, we must show that the predicate is **satisfied** by the known axioms:

$$\frac{\mathcal{A} \mid \Gamma \vdash e : P \Rightarrow \pi \quad \mathcal{A} \Vdash P}{e : \pi} \text{INST}$$

Right now,  $\mathcal{A} \Vdash P$  iff  $P \in \mathcal{A}$ , but we will complicate this situation later.

# Typing Rules

The existing rules now just thread  $\mathcal{A}$  through.

To use an overloaded type, we must show that the predicate is **satisfied** by the known axioms:

$$\frac{\mathcal{A} \mid \Gamma \vdash e : P \Rightarrow \pi \quad \mathcal{A} \Vdash P}{e : \pi} \text{INST}$$

Right now,  $\mathcal{A} \Vdash P$  iff  $P \in \mathcal{A}$ , but we will complicate this situation later.

We can introduce constrained types with the GEN rule:

$$\frac{P, \mathcal{A} \mid \Gamma \vdash e : \pi}{\mathcal{A} \mid \Gamma \vdash e : P \Rightarrow \pi} \text{GEN}$$

## Example

Suppose we wanted to show that  $3.2 + 4.4 :: \text{Float}$ .

- ❶  $(+) :: \forall a. (\text{Num } a) \Rightarrow a \rightarrow a \rightarrow a \in \Gamma.$
- ❷  $\text{Num Float} \in \mathcal{A}.$

## Example

Suppose we wanted to show that  $3.2 + 4.4 :: \text{Float}$ .

- 1  $(+) :: \forall a. (\text{Num } a) \Rightarrow a \rightarrow a \rightarrow a \in \Gamma.$
- 2  $\text{Num Float} \in \mathcal{A}.$
- 3 Using `ALLE` (from previous lecture), we can conclude  $(+) :: (\text{Num Float}) \Rightarrow \text{Float} \rightarrow \text{Float} \rightarrow \text{Float}.$

## Example

Suppose we wanted to show that  $3.2 + 4.4 :: \text{Float}$ .

- ❶  $(+) :: \forall a. (\text{Num } a) \Rightarrow a \rightarrow a \rightarrow a \in \Gamma.$
- ❷  $\text{Num Float} \in \mathcal{A}.$
- ❸ Using ALLE (from previous lecture), we can conclude  $(+) :: (\text{Num Float}) \Rightarrow \text{Float} \rightarrow \text{Float} \rightarrow \text{Float}.$
- ❹ Using INST (on previous slide) and ❷, we can conclude  $(+) :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float}$

## Example

Suppose we wanted to show that  $3.2 + 4.4 :: \text{Float}$ .

- ❶  $(+) :: \forall a. (\text{Num } a) \Rightarrow a \rightarrow a \rightarrow a \in \Gamma$ .
- ❷  $\text{Num Float} \in \mathcal{A}$ .
- ❸ Using ALLE (from previous lecture), we can conclude  $(+) :: (\text{Num Float}) \Rightarrow \text{Float} \rightarrow \text{Float} \rightarrow \text{Float}$ .
- ❹ Using INST (on previous slide) and ❷, we can conclude  $(+) :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float}$
- ❺ By the function application rule, we can conclude  $3.2 + 4.4 :: \text{Float}$  as required.

## Dictionaries and Resolution

This is called *ad-hoc polymorphism* because the type checker removes it — it is not a fundamental language feature, but merely a naming convenience.

The type checker converts ad-hoc polymorphism to parametric polymorphism.

Type classes are converted to types:

```
class Eq a where
  (==) : a → a → Bool
  (/=) : a → a → Bool
```

## Dictionaries and Resolution

This is called *ad-hoc polymorphism* because the type checker removes it — it is not a fundamental language feature, but merely a naming convenience.

The type checker converts ad-hoc polymorphism to parametric polymorphism.

Type classes are converted to types:

```
class Eq a where
  (==) : a → a → Bool
  (/=) : a → a → Bool
```

becomes

```
type EqDict a = (a → a → Bool × a → a → Bool)
```

A *dictionary* contains all the method implementations of a type class for a specific type.



# Dictionaries and Resolution

Instances become **values** of the dictionary type:

```
instance Eq Bool where
```

```
  True  == True  = True
```

```
  False == False = True
```

```
  _     == _     = False
```

```
  a /= b = not (a == b)
```

## Dictionaries and Resolution

Instances become **values** of the dictionary type:

**instance** Eq Bool **where**

```
True  == True  = True
False == False = True
_      == _      = False
a /= b = not (a == b)
```

becomes

```
True  ==_Bool True  = True
False ==_Bool False = True
_      ==_Bool _      = False
a /=_Bool b = not (a ==_Bool b)
```

*eqBoolDict* = ((==<sub>Bool</sub>), (/=<sub>Bool</sub>))

## Dictionaries and Resolution

Programs that rely on overloading now take dictionaries as parameters:

$$\text{same} :: \forall a. (\text{Eq } a) \Rightarrow [a] \rightarrow \text{Bool}$$
$$\text{same } [] = \text{True}$$
$$\text{same } (x : []) = \text{True}$$
$$\text{same } (x : y : xs) = x == y \wedge \text{same } (y : xs)$$

## Dictionaries and Resolution

Programs that rely on overloading now take dictionaries as parameters:

```
same :: ∀a. (Eq a) ⇒ [a] → Bool
same [] = True
same (x : []) = True
same (x : y : xs) = x == y ∧ same (y : xs)
```

Becomes:

```
same :: ∀a. (EqDict a) → [a] → Bool
same eq [] = True
same eq (x : []) = True
same eq (x : y : xs) = (fst eq) x y ∧ same eq (y : xs)
```

## Generative Instances

We can make **instances** also predicated on some constraints:

```
instance (Eq a) ⇒ (Eq [a]) where
  []      == []      = True
  (x : xs) == (y : ys) = x == y ∧ (xs == ys)
  _       == _       = False
  a  /=  b  = not (a == b)
```

Such instances are transformed into **functions**:

```
eqList :: EqDict a → EqDict [a]
```

Our set of axiom schema  $\mathcal{A}$  now includes **implications**, like  $(\text{Eq } a) \Rightarrow (\text{Eq } [a])$ . This makes the relation  $\mathcal{A} \Vdash P$  much more complex to solve.

# Coherence

Some languages (such as Haskell and Rust) insist that there is only **one instance per class per type** in the entire program. It achieves this by requiring that all instances are either:

- Defined along with the definition of the type class, or
- Defined along with the definition of the type.

This rules out so-called **orphan** instances.

# Coherence

Some languages (such as Haskell and Rust) insist that there is only **one instance per class per type** in the entire program. It achieves this by requiring that all instances are either:

- Defined along with the definition of the type class, or
- Defined along with the definition of the type.

This rules out so-called **orphan** instances.

There are a number of trade-offs with this decision:

- Modularity has been compromised but,
- Types like `Data.Set` can exploit this coherence to enforce invariants.

# Static Dispatch

Typically, the compiler can *inline* all dictionaries to their usage sites, thus eliminating all run-time cost for using type classes. This is only not possible if the exact type being used cannot be determined at compile-time, such as with *polymorphic recursion* etc.



# Subtyping

## 14

### Object-Oriented Languages

**ob-ject:** to feel distaste for something

*Webster's Dictionary*

engineering principle is information hiding or encapsula-  
tion, but the representation of

# Subtyping

To add subtyping to a language, we define a *partial order*<sup>2</sup> on types  $\tau \leq \rho$  and a *rule of subsumption*:

$$\frac{\Gamma \vdash e : \tau \quad \tau \leq \rho}{\Gamma \vdash e : \rho}$$

Type inference with subtyping is undecidable in general. Therefore, subsumptions (called *upcasts*) are sometimes made explicit (e.g. in OCaml):

$$\frac{\Gamma \vdash e : \tau \quad \tau \leq \rho}{\Gamma \vdash \text{upcast } \rho \ e : \rho}$$

---

<sup>2</sup>Remember discrete maths, or check the glossary.

# What is Subtyping?

What this partial order  $\tau \leq \rho$  actually means is up to the language.  
There are two main approaches:

# What is Subtyping?

What this partial order  $\tau \leq \rho$  actually means is up to the language. There are two main approaches:

- **Most common**: upcasts have no dynamic behaviour, i.e.  $\text{upcast } v \mapsto v$ . This requires that any **value** of type  $\tau$  could also be judged to have type  $\rho$ . If types are viewed as sets, this could be viewed as a **subset** relation.

# What is Subtyping?

What this partial order  $\tau \leq \rho$  actually means is up to the language. There are two main approaches:

- **Most common**: upcasts have no dynamic behaviour, i.e.  $\text{upcast } v \mapsto v$ . This requires that any **value** of type  $\tau$  could also be judged to have type  $\rho$ . If types are viewed as sets, this could be viewed as a **subset** relation.
- **Uncommon**: where upcasts cause a **coercion** to occur, actually converting the value from  $\tau$  to  $\rho$  at runtime.

**Observation**: By using an identity function as a coercion, the coercion view is **more general**.

# Desirable Properties

Coercion is more general, but can have confusing results.

## Example

Suppose  $\text{Int} \leq \text{Float}$ ,  $\text{Float} \leq \text{String}$  and  $\text{Int} \leq \text{String}$ .  
There are now two ways to coerce an `Int` to a `String`:

- 1 Directly: "3"
- 2 via `Float`: "3.0"

# Desirable Properties

Coercion is more general, but can have confusing results.

## Example

Suppose  $\text{Int} \leq \text{Float}$ ,  $\text{Float} \leq \text{String}$  and  $\text{Int} \leq \text{String}$ .  
There are now two ways to coerce an `Int` to a `String`:

- 1 Directly: "3"
- 2 via `Float`: "3.0"

Typically, we would enforce that the subtype coercions are **coherent**, such that no matter which coercion is chosen, the same result is produced.

# Behavioural Subtyping

Another constraint is that the **syntactic** notion of subtyping should correspond to something **semantically**. In other words, if we know  $\tau \leq \rho$ , then it should be reasonable to replace any value of type  $\rho$  with a value of type  $\tau$  without any observable difference.

## Liskov Substitution Principle

Let  $\varphi(x)$  be a property provable about objects  $x$  of type  $\rho$ . Then  $\varphi(y)$  should be true for objects  $y$  of type  $\tau$  where  $\tau \leq \rho$ .

Languages such as Java and C++, which allow for user-defined subtyping relationships (**inheritance**), put the onus on the user to ensure this condition is met.



## Product Types

Assuming a basic rule  $\text{Int} \leq \text{Float}$ , how do we define subtyping for our compound data types?

What is the relationship between these types?

- $(\text{Int} \times \text{Int})$
- $(\text{Float} \times \text{Float})$
- $(\text{Float} \times \text{Int})$
- $(\text{Int} \times \text{Float})$

## Product Types

Assuming a basic rule  $\text{Int} \leq \text{Float}$ , how do we define subtyping for our compound data types?

What is the relationship between these types?

- $(\text{Int} \times \text{Int})$
- $(\text{Float} \times \text{Float})$
- $(\text{Float} \times \text{Int})$
- $(\text{Int} \times \text{Float})$

$$\frac{\tau_1 \leq \rho_1 \quad \tau_2 \leq \rho_2}{(\tau_1 \times \tau_2) \leq (\rho_1 \times \rho_2)}$$

# Sum Types

What is the relationship between these types?

- $(\text{Int} + \text{Int})$
- $(\text{Float} + \text{Float})$
- $(\text{Float} + \text{Int})$
- $(\text{Int} + \text{Float})$

# Sum Types

What is the relationship between these types?

- $(\text{Int} + \text{Int})$
- $(\text{Float} + \text{Float})$
- $(\text{Float} + \text{Int})$
- $(\text{Int} + \text{Float})$

$$\frac{\tau_1 \leq \rho_1 \quad \tau_2 \leq \rho_2}{(\tau_1 + \tau_2) \leq (\rho_1 + \rho_2)}$$

# Sum Types

What is the relationship between these types?

- $(\text{Int} + \text{Int})$
- $(\text{Float} + \text{Float})$
- $(\text{Float} + \text{Int})$
- $(\text{Int} + \text{Float})$

$$\frac{\tau_1 \leq \rho_1 \quad \tau_2 \leq \rho_2}{(\tau_1 + \tau_2) \leq (\rho_1 + \rho_2)}$$

Any other **compound** types?

# Functions

What is the relationship between these types?

- $(\text{Int} \rightarrow \text{Int})$
- $(\text{Float} \rightarrow \text{Float})$
- $(\text{Float} \rightarrow \text{Int})$
- $(\text{Int} \rightarrow \text{Float})$

# Functions

What is the relationship between these types?

- $(\text{Int} \rightarrow \text{Int})$
- $(\text{Float} \rightarrow \text{Float})$
- $(\text{Float} \rightarrow \text{Int})$
- $(\text{Int} \rightarrow \text{Float})$

The relation is **flipped** on the left hand side!

# Functions

What is the relationship between these types?

- $(\text{Int} \rightarrow \text{Int})$
- $(\text{Float} \rightarrow \text{Float})$
- $(\text{Float} \rightarrow \text{Int})$
- $(\text{Int} \rightarrow \text{Float})$

The relation is **flipped** on the left hand side!

$$\frac{\rho_1 \leq \tau_1 \quad \tau_2 \leq \rho_2}{(\tau_1 \rightarrow \tau_2) \leq (\rho_1 \rightarrow \rho_2)}$$



# Variance

The way a *type constructor* (such as  $+$ ,  $\times$ , Maybe or  $\rightarrow$ ) interacts with subtyping is called its *variance*.

# Variance

The way a *type constructor* (such as  $+$ ,  $\times$ , Maybe or  $\rightarrow$ ) interacts with subtyping is called its *variance*. For a type constructor  $C$ , and  $\tau \leq \rho$ :

- If  $C \tau \leq C \rho$ , then  $C$  is *covariant*.

**Examples:** Products (both arguments), Sums (both arguments), Function return type, ...

# Variance

The way a *type constructor* (such as  $+$ ,  $\times$ , Maybe or  $\rightarrow$ ) interacts with subtyping is called its *variance*. For a type constructor  $C$ , and  $\tau \leq \rho$ :

- If  $C \tau \leq C \rho$ , then  $C$  is *covariant*.

**Examples:** Products (both arguments), Sums (both arguments), Function return type, ...

- If  $C \rho \leq C \tau$ , then  $C$  is *contravariant*.

**Examples:** Function argument type, ...

# Variance

The way a *type constructor* (such as  $+$ ,  $\times$ , Maybe or  $\rightarrow$ ) interacts with subtyping is called its *variance*. For a type constructor  $C$ , and  $\tau \leq \rho$ :

- If  $C \tau \leq C \rho$ , then  $C$  is *covariant*.

**Examples:** Products (both arguments), Sums (both arguments), Function return type, ...

- If  $C \rho \leq C \tau$ , then  $C$  is *contravariant*.

**Examples:** Function argument type, ...

- If it is neither *covariant* nor *contravariant* then it is (confusingly) called *invariant*.

**Examples:** `data Endo a = E (a  $\rightarrow$  a)`

## Stuffing it up

Many languages have famously stuffed this up, at the expense of type safety.

### 19 Types

Dart supports optional typing based on interface types.

*The type system is unsound, due to the covariance of generic types. This is a deliberate choice (and undoubtedly controversial). Experience has shown that sound type rules for generics fly in the face of programmer intuition. It is easy for tools to provide a sound type analysis if they choose, which may be useful for tasks like refactoring.*

## Stuffing it up

Many languages have famously stuffed this up, at the expense of type safety.

### 19 Types

Dart supports optional typing based on interface types.

*The type system is unsound, due to the covariance of generic types. This is a deliberate choice (and undoubtedly controversial). Experience has shown that sound type rules for generics fly in the face of programmer intuition. It is easy for tools to provide a sound type analysis if they choose, which may be useful for tasks like refactoring.*

A few years later...

## Language and libraries

- Dart's type system is now sound.
  - Fixing common type problems

## Java too

Java (and its Seattle-based cousin, C<sup>‡</sup>) also broke type safety with incorrect variance in **arrays**.

We will demonstrate how this violates **preservation**, time permitting.

(Java redeemed itself by introducing invariant collections along with parametric polymorphism in 2004. These were believed sound until 2016.)