CSC324 Lecture 18

Note about Friday's lecture

Guest lecture from Penelope Phippen (https://penelope.zone/), who is a core contributor to several ubiquitous Ruby libraries. I think it'll be very interesting!

She will be talking about IRL metaprogramming, in the context of extending Ruby's object system to have object fields stored remotely in a SQL database (Such functionality is often called an **object-relational mapping.**)

If you haven't taken a database class yet, you may want to spend a few minutes reading up on the basics of SQL databases beforehand.

A2 groups...

If you are adjusting your groups for A2 owing to your partner dropping, etc, and Markus is giving you problems, email me (/ccing your partner) with both your IDs and I'll adjust the groups manually.

Since people have asked: yes, if you want to, you can keep working with your A1 partner for A2!

Last time...

We saw **parametric polymorphism** and saw how **type variables** can stand in for any concrete type.

Today:

- Wrapping up parametric polymorphism
- Practicalities of implementing polymorphic types in the language runtime
- Ad-hoc polymorphism with typeclasses
- (if time) subtyping

Type annotations

Note: When we want to talk about functions with arguments of particular types, sometimes I will write function abstractions in this form:

```
(λ (t1:T1 t2:T2) ...)
```

As we've seen, Haskell can often **infer types** without us manually needing to **annotate types**, so we'll only do this when it's necessary or illuminating to do so.

Type substitution and application

Definition: We say that a **type substitution** (or simply a substitution) is a mapping of type variables to types.

Example: Given the following polymorphic type definition:

```
data Option a = None
              | Some a
     deriving (Show)
```

`Some [1,2,3]` would have a type substitution `{a -> [Num]}`, because the value inside the `Some` is of type `[Num]`.

Type substitution and application

Definition: We say that a **type application** is the process by which type variables are replaced with concrete types, given a substitution.

Type substitution and application

Example: `(λ (f a) (f (f a)))`

=> this function application expression has type: `(X -> X) -> X -> X`

Type substitution and application

Example: `(λ (f a) (f (f a)))`
=> this function application expression has type: `(X -> X) -> X -> X`

(Recall: we can write the abstraction, with type annotations, as the expression `(λ (f: X->X a: X) (f (f a)))` if we wanted.)

Type substitution and application

Example: `(λ (f a) (f (f a)))` with type substitutions `{X / Int}`
=> this function application expression has type: `(X -> X) -> X -> X`

Type substitution and application

Example: `(λ (f a) (f (f a)))` with type substitutions `{X / Int}`
=> this function application expression has type: `(X -> X) -> X -> X`

`(X -> X) -> X -> X` , after substitution with `{X / Int}` , is
=> `(Int -> Int) -> Int -> Int`

Type substitution and application

Example: `(λ (f a) (f (f a)))` with type substitutions `{X / Int}`
=> this function application expression has type: `(X -> X) -> X -> X`

`(X -> X) -> X -> X` , after substitution with `{X / Int}` , is
=> `(Int -> Int) -> Int -> Int`

(Recall: we can write the substituted abstraction, with type annotations, as the expression `(λ (f: Int->Int, a: Int) (f (f a)))` if we wanted.)

Type substitution and application

Example: `(λ (f a) (f (f a)))` with type substitutions `{X / Int}`
=> this function application expression has type: `(X -> X) -> X -> X`

A more succinct notation, in line with parametric polymorphism in languages like Java, Scala, Kotlin, C++...

`(λ (f: X->X a: X) (f (f a)))` **`[Int]`**
=> `(λ (f: Int->Int a: Int) (f (f a)))`

Type substitution and application

Example: `(λ (f a) (f (f a)))` with type substitutions `{X / Int}`
=> this function application expression has type: `(X -> X) -> X -> X`

A more succinct notation, in line with parametric polymorphism in languages like Java, Scala, Kotlin, C++...

"type application"

Int is applied to the type variable X in the term.

```
(λ (f: X->X a: X) (f (f a))) [Int]
=> (λ (f: Int->Int a: Int) (f (f a)))
```

Type substitution and application

Example: `(λ (f a) (f (f a)))` with type substitutions `{X / Int}`
=> this function application expression has type: `(X -> X) -> X -> X`

A more succinct notation, in line with parametric polymorphism in languages like Java, Scala, Kotlin, C++...

```
(λ (f: X->X a: X) (f (f a))) [Int]
=> (λ (f: Int->Int a: Int) (f (f a)))
```

Note that this looks very similar to applying an argument (the `[int]`) to a function abstraction!

Type substitution and application

Assignment 2 connection: If we treat patmat variables as type variables, a patmat environment as our substitution, and a datum as the type signature, then...

=> (X -> X) -> X -> X **[Int]** == (Int -> Int) -> Int -> Int

```
Welcome to DrRacket, version 7.6 [3m].
Language: racket, with debugging; memory limit: 256 MB.
> (substitute-bindings
   (hash '?X 'Nat)
   '((?X -> ?X) -> ?X -> ?X))
'((Nat -> Nat) -> Nat -> Nat)
>
```

Our favourite combination of sum and product types

What could typing rules for a `List[T]` look like?

```
; A list of T is
; - 'empty
; - (cons x xs) where:
;   - x is a T
;   - xs is a list of T
```

`'empty: List[T]`

`cons[T]: List[T]`

$$\frac{\texttt{t1:T} \quad \texttt{t2: List[T]}}{\texttt{(cons t1 t2): List[T]}}$$

$$\frac{\texttt{t: cons[T]}}{\texttt{(car t) : T}}$$

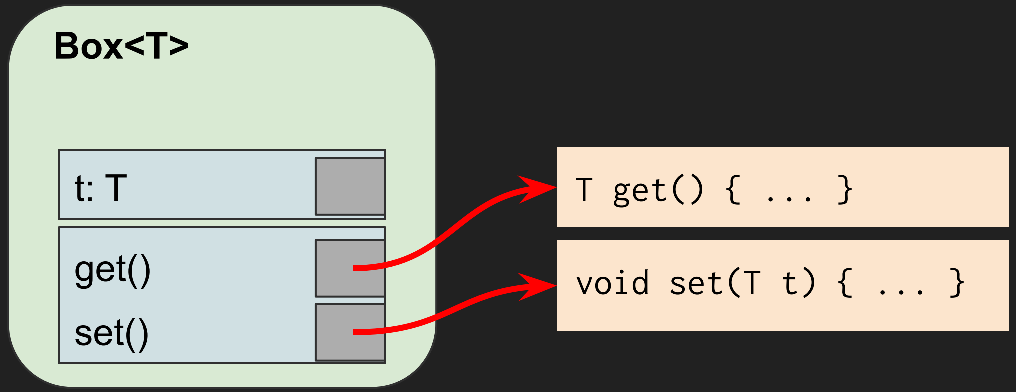
$$\frac{\texttt{t: cons[T]}}{\texttt{(cdr t) : List[T]}}$$

Practicalities of implementing para. poly.

Recall that a vtable-based object is:

- A structure containing the object's fields
- A function pointer table containing the object's methods

What is complicated by introducing polymorphism here?

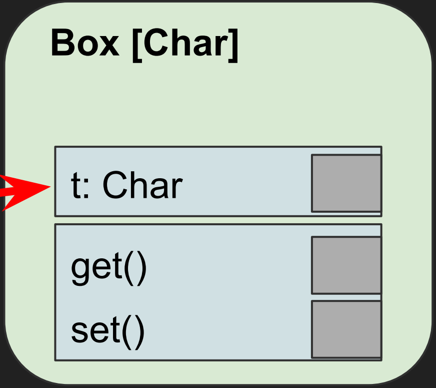


One example: different memory requirements

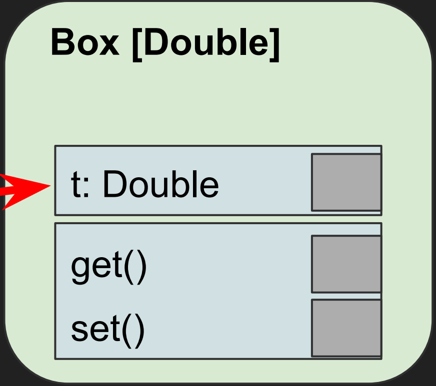
Recall that structs are fixed-size in memory, but what happens if the size of the type T can vary?

We need a way of implementing a data structure with a type variable that the language runtime can reason about correctly...

A char is one byte



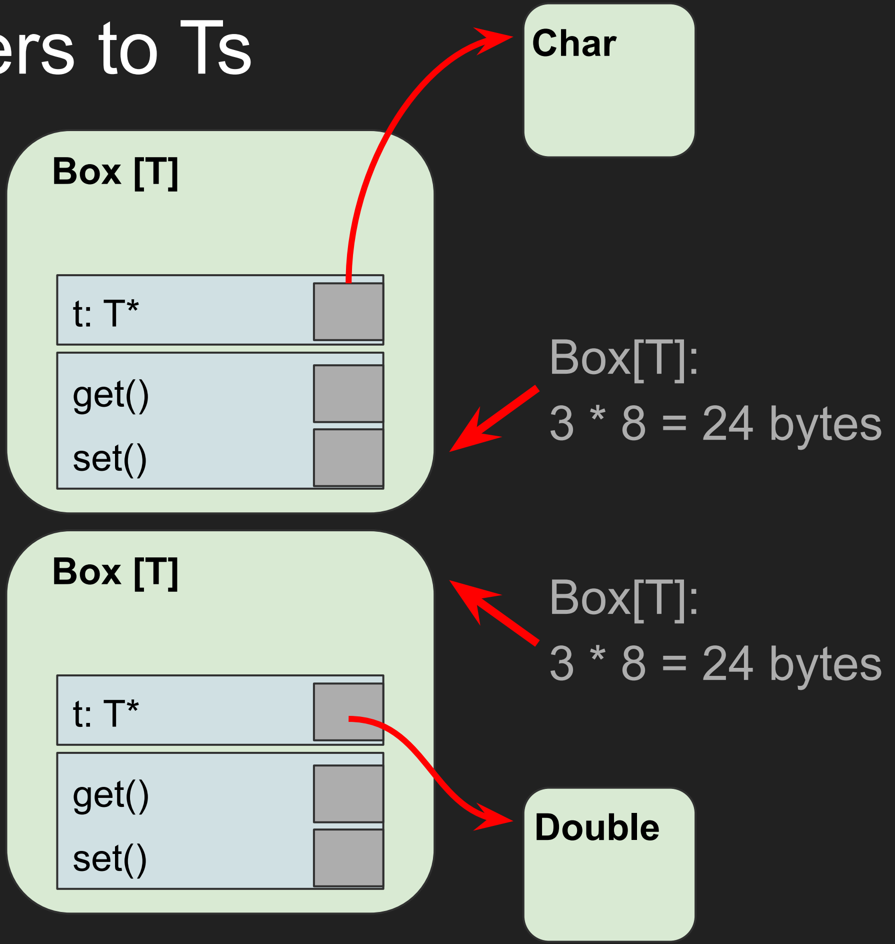
A double is eight bytes!



Solution: only store pointers to Ts

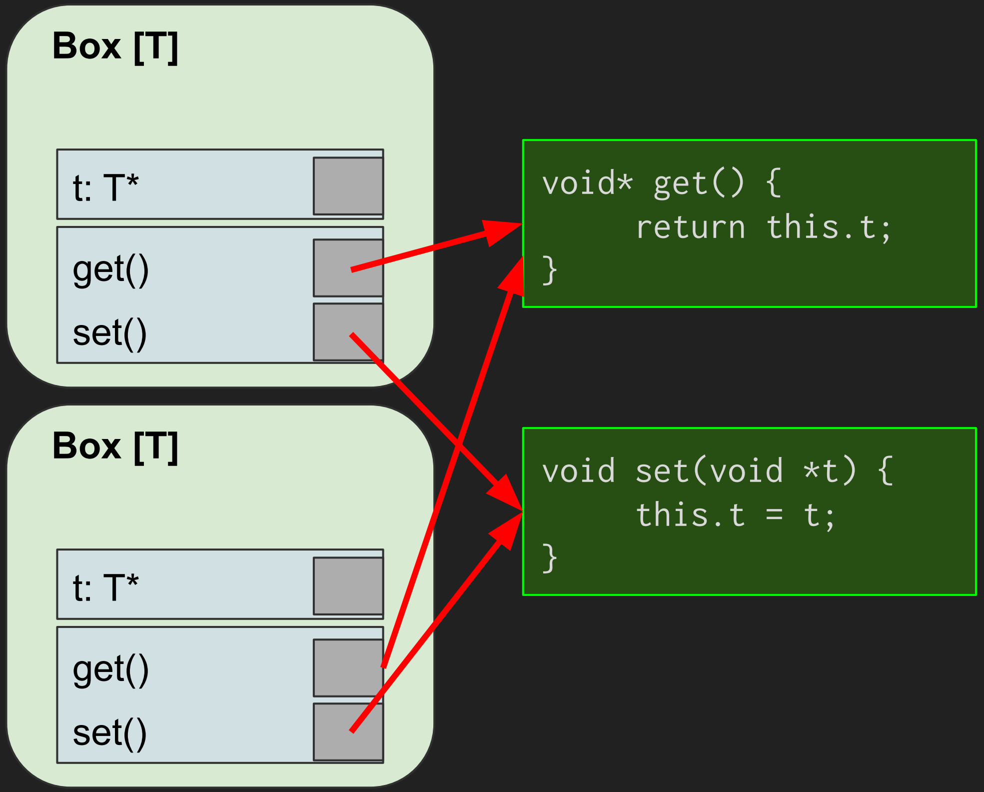
One way to resolve this is to, under the hood, only have pointers to the actual T t field but leave the implementation otherwise polymorphic in T. This means that the abstract Box [T] machine code doesn't need to vary for different Ts.

Java does this; all fields that are typed generically are stored by reference.



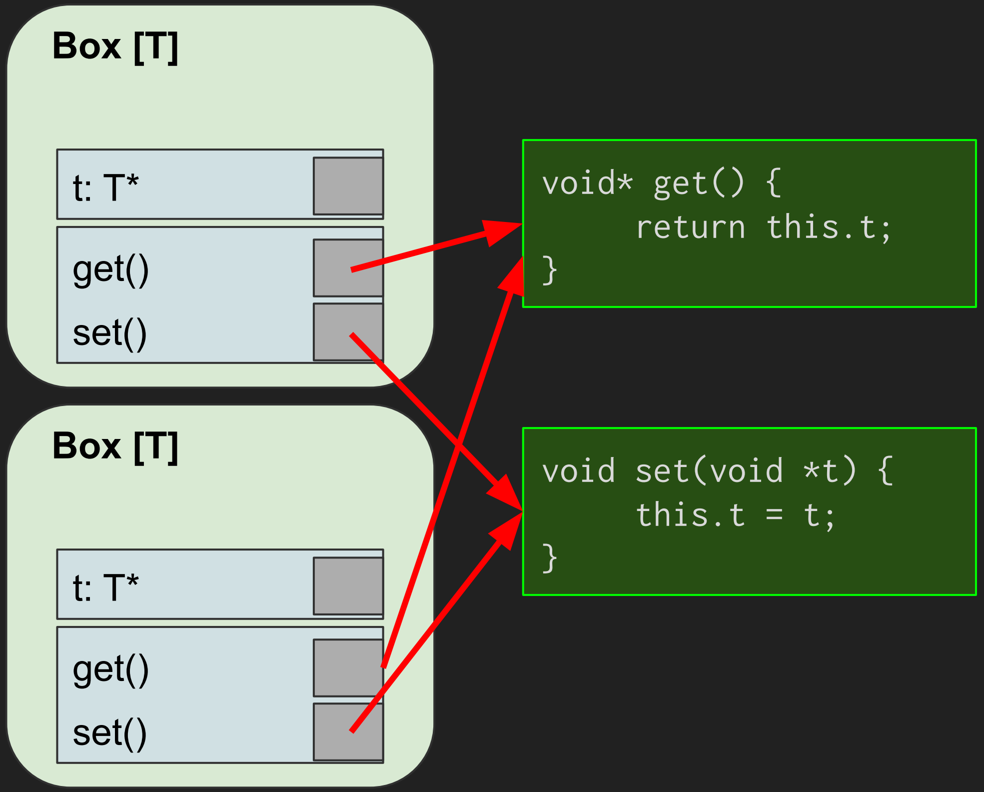
Solution: only store pointers to Ts

Because `get()` and `set()` are methods that operate on, essentially, an untyped void* pointer, their implementation can be shared between `Box [Char]` and `Box [Double]`.



Solution: only store pointers to Ts

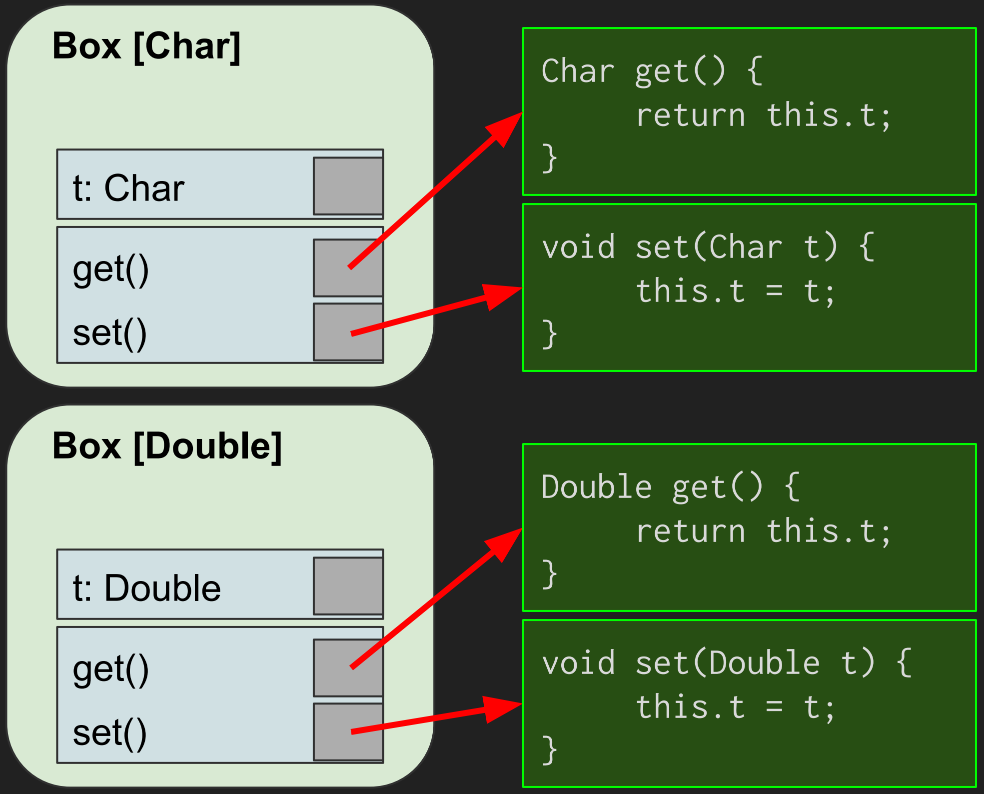
This remains typesafe because the compiler will, at compile-time, ensure that the top Box is only used in the context of holding a Char, and the bottom Box is only used in the context of holding a Double.



This loss of runtime type information is called **type erasure**.

Solution: specialise for each Box [T]

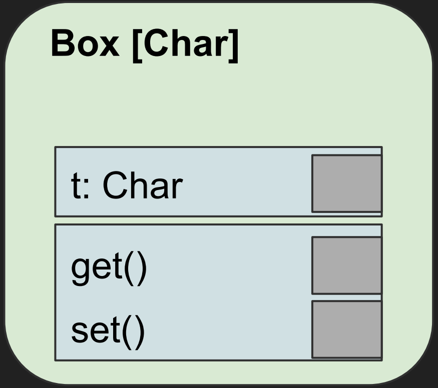
Another way is for the compiler to "copy and paste" the abstract data definition for every type variable, as if the programmer had manually written distinct "box containing a Char" and "box containing a Double" data definitions.



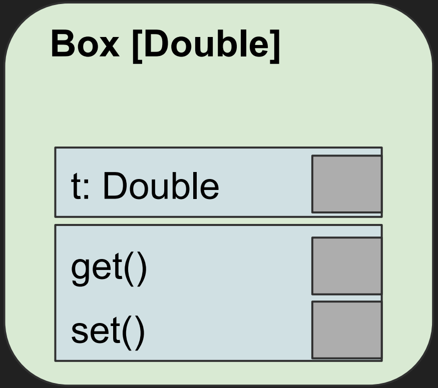
Solution: specialise for each Box [T]

Note that the size of each structure can vary, as the size of the field inside it can vary.

Box[Char]:
1 + 2 * 8 = 17 bytes



Box[Double]:
8 + 2 * 8 = 24 bytes



(Note: IRL, there will be structure padding to some boundary, so the structs won't necessarily be exactly this size)

Solution: specialise for each Box [T]

This is the approach C++ takes.

Each concrete Box type is completely distinct at runtime.

Our two types have lost their type variable; they are now **monomorphic** types.

foo.cpp

```
#include <iostream>

template <typename T>
class Box {
    T t;

public:
    Box(T _t) : t{_t} {}
    T get() const { return t; }
    void set(T _t) { this->t = _t; }
};

int main(int argc, char **argv) {
    Box<char> b1('c');
    Box<double> b2(42.0);
}
```

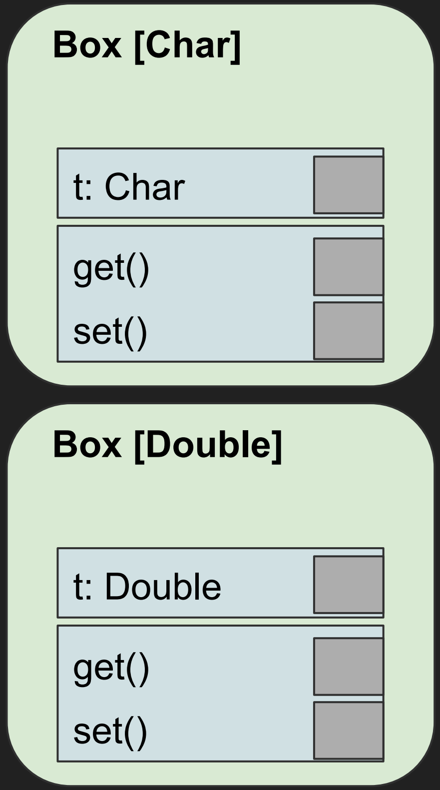
Solution: specialise for each Box [T]

By compiling the C++ program on the previous line and objdumping the binary executable, we can see that there are distinct functions for Box<char> and Box<double> (and, if we were to keep scrolling, the getter and setter)

```
0000000100001100 Box<char>::Box(char):
100001100: 55                              pushq   %rbp
100001101: 48 89 e5                        movq    %rsp, %rbp
100001104: 48 83 ec 10                     subq    $16, %rsp
100001108: 48 89 7d f8                     movq    %rdi, -8(%rbp)
10000110c: 40 88 75 f7                     movb    %sil, -9(%rbp)
100001110: 48 8b 7d f8                     movq    -8(%rbp), %rdi
100001114: 0f be 75 f7                     movsbl  -9(%rbp), %esi
100001118: e8 83 01 00 00                  callq   387 <__ZN3BoxIcEC2Ec>
10000111d: 48 83 c4 10                     addq    $16, %rsp
100001121: 5d                              popq    %rbp
100001122: c3                              retq
100001123: 66 2e 0f 1f 84 00 00 00 00 00   nopw    %cs:(%rax,%rax)
10000112d: 0f 1f 00                        nopl    (%rax)

0000000100001130 Box<double>::Box(double):
100001130: 55                              pushq   %rbp
100001131: 48 89 e5                        movq    %rsp, %rbp
100001134: 48 83 ec 10                     subq    $16, %rsp
100001138: 48 89 7d f8                     movq    %rdi, -8(%rbp)
10000113c: f2 0f 11 45 f0                  movsd   %xmm0, -16(%rbp)
100001141: 48 8b 7d f8                     movq    -8(%rbp), %rdi
100001145: f2 0f 10 45 f0                  movsd   -16(%rbp), %xmm0
10000114a: e8 71 01 00 00                  callq   369 <__ZN3BoxIdEC2Ed>
10000114f: 48 83 c4 10                     addq    $16, %rsp
100001153: 5d                              popq    %rbp
100001154: c3                              retq
100001155: 66 2e 0f 1f 84 00 00 00 00 00   nopw    %cs:(%rax,%rax)
10000115f: 90                              nop
```

Solution: specialise for each Box [T]

This "copy and paste for each data definition" feels a bit like how a macro "copies and pastes" syntax transformations on each instantiation; indeed, C++ was originally written as a macro that expands polymorphism to each concrete type!



A limitation of parametric polymorphism

Recall this observation from last time: because nothing is known about what kind of type is stored in T. This implies a universal quantification: "for all types `a`, `Box a` is a type"

Parametric polymorphism

With parametric polymorphism, type variables are held abstract during typechecking. This ensures that any well-typed term will behave correctly no matter what concrete type is substituted later on.

This is powerful, but constricting: we can't assume anything about `T`, so we're limited in what we can actually do with it inside the class. (We can print the `T` in `toString` only because that's a method implemented on every object in Java, so every T is guaranteed to have such a method.)

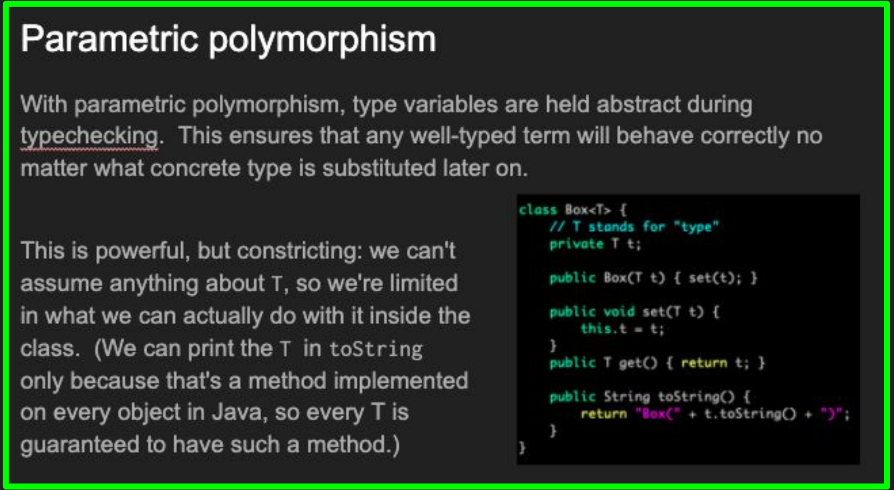
```java
class Box<T> {
    // T stands for "type"
    private T t;

    public Box(T t) { set(t); }

    public void set(T t) {
        this.t = t;
    }
    public T get() { return t; }

    public String toString() {
        return "Box(" + t.toString() + ")";
    }
}
```

A limitation of parametric polymorphism

Recall this observation from last time: because nothing is known about what kind of type is stored in T. This implies a universal quantification: "for all types `a`, `Box a` is a type"

Things we might like to say:

- "Whatever T is, it implements some interface"
- "Whatever T is, it extends some parent class"



Parametric polymorphism

With parametric polymorphism, type variables are held abstract during typechecking. This ensures that any well-typed term will behave correctly no matter what concrete type is substituted later on.

This is powerful, but constricting: we can't assume anything about `T`, so we're limited in what we can actually do with it inside the class. (We can print the `T` in `toString` only because that's a method implemented on every object in Java, so every T is guaranteed to have such a method.)

```
class Box<T> {
    // T stands for "type"
    private T t;

    public Box(T t) { set(t); }

    public void set(T t) {
        this.t = t;
    }
    public T get() { return t; }

    public String toString() {
        return "Box(" + t.toString() + ")";
    }
}
```

Two solutions:

- Ad-hoc polymorphism (ie. Typeclasses)
- Subclasses (ie. inheritance)

Qualified types

Suppose π(t) is a predicate that consumes a type and produces a boolean.

Definition: We say that a polymorphic type Q is a **qualified type** if its type variable must satisfy a particular π(t).

Qualified types

Example: We may wish to define an addition function as

`(+): ∀ t.π(t) => t -> t -> t, where π(t) = (t == Integer || t == Double)`

Where the implementation of (+) might defer to a specialised addition function depending on the actual type:

```
(define (+ a b)
        (if (= (typeof a) Integer) ; pseudocode
            (integer-+ a b)
            (double-+ a b)))
```

Qualified types

Example: We may wish to define an addition function as

`(+): ∀ t.π(t) => t -> t -> t, where π(t) = (t == Integer || t == Double)`

Where the implementation of (+) might defer to a specialised addition function depending on the actual type:

This would happen at runtime; wouldn't it be nice if this could happen at compile-time!

```
(define (+ a b)
        (if (= (typeof a) Integer) ; pseudocode
            (integer-+ a b)
            (double-+ a b)))
```

Ad-hoc polymorphism

We say that if a polymorphic value exhibits different behaviours when "viewed" in a different typing contexts, it features "ad-hoc" polymorphism.

"Ad-hoc" here means that we have total freedom to tailor the behaviour for each typing context as much as we want, not that it's "unsound" or "implemented without consideration".

Ad-hoc polymorphism

You have seen ad-hoc polymorphism before in the form of **method overloading** in languages like Java.

```
class Foo {
    static int bar(int i, int j) {
        System.out.println("Adding two integers...");
        return i + j;
    }
    static double bar(double i, double j) {
        System.out.println("Adding two doubles...");
        return i + j;
    }

    public static void main(String[] args) {
        bar(1,1);
        bar(3.14, 2.71);
    }
}
```

Ad-hoc polymorphism

You have seen ad-hoc polymorphism before in the form of **method overloading** in languages like Java.

```java
class Foo {
    static int bar(int i, int j) {
        System.out.println("Adding two integers...");
        return i + j;
    }
    static double bar(double i, double j) {
        System.out.println("Adding two doubles...");
        return i + j;
    }

    public static void main(String[] args) {
        bar(1,1);
        bar(3.14, 2.71);
    }
}
```

This example works because the arguments to `foo` are unambiguous; 1 is not a double and 3.14 is not an int.

```
[MSFT] /tmp javac Foo.java
[MSFT] /tmp java Foo
Adding two integers...
Adding two doubles...
[MSFT] /tmp
```

Typeclasses (Lab 8)

<u>Definition:</u> A **typeclass** represents a family of types (the instances of the class) together with an associated set of functions defined for each instance of the class.

For some given typeclass C and type a, the predicate function application `C a` (in Haskell syntax) represents the assertion that `a` is an instance of C. Treating typeclasses as qualified types, this will be our $\pi(t)$ predicate.

(Note: instances of a typeclass are types! This is different than the OOP terminology, where instances of an OO class are objects.)

Defining a typeclass...

```
-- JSON: JavaScript Object Notation
-- toJSON will consume a value of some
-- particular type and convert it to the string
-- representation of that type in JSON.
```

Defining a typeclass...

```
-- JSON: JavaScript Object Notation
-- toJSON will consume a value of some
-- particular type and convert it to the string
-- representation of that type in JSON.
class JSON a where
```

`class JSON a` introduces a name `JSON` for the class, and indicates that the type variable `a` will be used to represent an arbitrary instance of the class

Defining a typeclass...

```
-- JSON: JavaScript Object Notation
-- toJSON will consume a value of some
-- particular type and convert it to the string
-- representation of that type in JSON.
class JSON a where
```

The type system-level predicate `JSON a` is also defined, that returns true for all type that are instances of the JSON class.

Defining a typeclass...

```
*Main> :t toJSON
toJSON :: JSON a => a -> String
*Main>
```

What can we say about the typeclass function we defined?

`toJSON`'s polymorphism is **qualified** for all types `a` that satisfy `JSON a`

Defining a typeclass...

```
*Main> :t toJSON
toJSON :: JSON a => a -> String
*Main>
```

Finally! We can understand the double arrow vs single arrow; the double arrow is **implication:** "If `JSON a` holds for this `a`, then it is a function `a -> string`"

Implementing toJSON on a sum type via pattern matching...

```haskell
-- JSON: JavaScript Object Notation
-- toJSON will consume a value of some
-- particular type and convert it to the string
-- representation of that type in JSON.
class JSON a where
    toJSON :: a -> String

-- Bool
instance JSON Bool where
    toJSON True = "true"
    toJSON False = "false"
```

Implementing toJSON on a type by deferring to the behaviour of another typeclass...

```haskell
-- JSON: JavaScript Object Notation
-- toJSON will consume a value of some
-- particular type and convert it to the string
-- representation of that type in JSON.
class JSON a where
    toJSON :: a -> String

-- Bool
instance JSON Bool where
    toJSON True = "true"
    toJSON False = "false"

-- Integer
instance JSON Integer where
    toJSON = show

-- String
instance JSON String where
    toJSON = show

```

Implementing toJSON on an algebraic datatype

```haskell
-- JSON: JavaScript Object Notation
-- toJSON will consume a value of some
-- particular type and convert it to the string
-- representation of that type in JSON.
class JSON a where
    toJSON :: a -> String

-- Bool
instance JSON Bool where
    toJSON True = "true"
    toJSON False = "false"

-- Integer
instance JSON Integer where
    toJSON = show

-- String
instance JSON String where
    toJSON = show

-- List a
instance {-# OVERLAPPABLE #-} JSON a => JSON [a] where
    toJSON l = "[" ++ (concat (intersperse ", " (map toJSON l))) ++ "]"
```

Implementing toJSON on an algebraic datatype

```haskell
-- JSON: JavaScript Object Notation
-- toJSON will consume a value of some
-- particular type and convert it to the string
-- representation of that type in JSON.
class JSON a where
    toJSON :: a -> String

-- Bool
instance JSON Bool where
    toJSON True = "true"
    toJSON False = "false"

-- Integer
instance JSON Integer where
    toJSON = show

-- String
instance JSON String where
    toJSON = show

-- List a
instance {-# OVERLAPPABLE #-} JSON a => JSON [a] where
    toJSON l = "[" ++ (concat (intersperse ", " (map toJSON l))) ++ "]"

```

The definition for JSON [a] depends on the definition of JSON a; if a is an instance of JSON, so too is [a]

:i (short for :info) will enumerate all instances of a typeclass.

```
*Main> :i JSON
class JSON a where
  toJSON :: a -> String
        -- Defined at /tmp/Lecture16.hs:37:1
instance (JSON k, JSON v) => JSON (Map.Map k v)
  -- Defined at /tmp/Lecture16.hs:58:10
instance [overlappable] JSON a => JSON [a]
  -- Defined at /tmp/Lecture16.hs:54:31
instance JSON String -- Defined at /tmp/Lecture16.hs:50:10
instance JSON Integer -- Defined at /tmp/Lecture16.hs:46:10
instance JSON Bool -- Defined at /tmp/Lecture16.hs:41:10
*Main>
```

"Are typeclasses exhaustive?"

What happens if we try to transform a type that isn't a member of a typeclass to its JSON representation?

```haskell
data Day = Monday
         | Tuesday
         | Wednesday
         | Thursday
         | Friday
         | Saturday
         | Sunday
         deriving (Show)

```

The dreaded "No instance of … arising from a use of …" error!

```
*Main> toJSON Monday

<interactive>:96:1:
    No instance for (JSON Day) arising from a use of ‘toJSON’
    In the expression: toJSON Monday
    In an equation for ‘it’: it = toJSON Monday
*Main>
```

"Are typeclasses unique?"

Recall that the type of a string is a list of `Char`s (no surprise there).

```
*Main> :t "Hello"
"Hello" :: [Char]
*Main>
```

But, we have a JSON instance for lists of `Char`s but also polymorphic lists… `[Char]` could just as easily use either!

```haskell
-- String (aka [Char])
instance JSON String where
    toJSON = show

-- List a
instance JSON a => JSON [a] where
    toJSON l = "[" ++
               (concat (intersperse ", " (map toJSON l))) ++
               "]"
```

"Are typeclasses unique?"

```
*Main> toJSON "Hello"

<interactive>:108:1:
    Overlapping instances for JSON [Char]
      arising from a use of ‘toJSON’
    Matching instances:
      instance JSON a => JSON [a] -- Defined at /tmp/Lecture16.hs:57:10
      instance JSON String -- Defined at /tmp/Lecture16.hs:53:10
    In the expression: toJSON "Hello"
    In an equation for ‘it’: it = toJSON "Hello"
*Main>
```

"Are typeclasses unique?"

"GHC [the Glasgow Haskell Compiler] requires that it be *unambiguous* which instance declaration should be used to resolve a type-class constraint."

```
*Main> :t "Hello"
"Hello" :: [Char]
*Main>
```

```haskell
-- String (aka [Char])
instance JSON String where
    toJSON = show

-- List a
instance JSON a => JSON [a] where
    toJSON l = "[" ++
                  (concat (intersperse ", " (map toJSON l))) ++
                  "]"

```

"Are typeclasses unique?"

We saw how the OVERLAPPABLE language extension loosens this restriction… but which does it pick?

```
*Main> :t "Hello"
"Hello" :: [Char]
*Main>
```

```haskell
-- String (aka [Char])
instance JSON String where
    toJSON = show

-- List a
instance {-# OVERLAPPABLE #-} JSON a => JSON [a] where
    toJSON l = "[" ++
                  (concat (intersperse ", " (map toJSON l))) ++
                  "]"

```

If you were implementing Haskell, which overlapping instance would <u>you</u> choose?

"Are typeclasses unique?"

"If I were choosing between overlapping typeclass instances I would simply choose the most appropriate one"

```
*Main> :t "Hello"
"Hello" :: [Char]
*Main>
```

```haskell
-- String (aka [Char])
instance JSON String where
    toJSON = show

-- List a
instance {-# OVERLAPPABLE #-} JSON a => JSON [a] where
    toJSON l = "[" ++
                (concat (intersperse ", " (map toJSON l))) ++
                "]"

```

"Are typeclasses unique?"

OK, but concretely, is that…

- The first instance that Haskell comes across that satisfies the constraint?
- The *last* instance that Haskell comes across that satisfies the constraint?

That seems dicey, typeclasses are **open** so it's hard to know in advance what those would be!

```
*Main> :t "Hello"
"Hello" :: [Char]
*Main>
```

```haskell
-- String (aka [Char])
instance JSON String where
    toJSON = show

-- List a
instance {-# OVERLAPPABLE #-} JSON a => JSON [a] where
    toJSON l = "[" ++
                   (concat (intersperse ", " (map toJSON l))) ++
                   "]"

```

"Are typeclasses unique?"

What about some metric for "the most precise instance"?

```
*Main> :t "Hello"
"Hello" :: [Char]
*Main> 
```

```haskell
-- String (aka [Char])
instance JSON String where
    toJSON = show

-- List a
instance {-# OVERLAPPABLE #-} JSON a => JSON [a] where
    toJSON l = "[" ++
                (concat (intersperse ", " (map toJSON l))) ++
                "]"

```

Formalising "the most specific instance"

Given two instances

```
instance Q1 => P1 where ...
instance Q2 => P2 where ...
```

Definition: We say that P1 and P2 *overlap* if they unify!

Formalising "the most specific instance"

Given two instances

```
instance Q1 => P1 where ...
instance Q2 => P2 where ...
```

Definition: We say that P1 and P2 *overlap* if they unify!

```
> (unify '(JSON Bool) '(JSON Integer) (hash))
'failed
```

Formalising "the most specific instance"

Given two instances

```
instance Q1 => P1 where ...
instance Q2 => P2 where ...
```

Definition: We say that P1 and P2 *overlap* if they unify!

```
> (unify '(Eq Integer) '(Ord Integer) (hash))
'failed
```

Formalising "the most specific instance"

Given two instances

```
instance Q1 => P1 where ...
instance Q2 => P2 where ...
```

Definition: We say that P1 and P2 *overlap* if they unify!

```
> (unify '(JSON (listof Char)) '(JSON (listof ?a)) (hash))
'#hash((?a . Char))
```

Formalising "the most specific instance"

Given two instances

```
instance Q1 => P1 where ...
instance Q2 => P2 where ...
```

Definition: We say that P1 and P2 *overlap* if they unify!

```
> (unify '(JSON (listof Char)) '(JSON (listof ?a)) (hash))
'#hash((?a . Char))
```

So given some concrete type T that overlaps with P1 and P2, how do we decide which we should choose?

Revisiting unification

Remember that `unify` on Asn2 returns a mapping of substitutions, such that if the substitutions are applied to both sides, both sides would be identical:

```
(unify '(The ?pet of Jacqueline is named (the chairman))
       '(The  cat of Jacqueline is named ?name))

=> (hash '(?pet . cat) '(?name . (the chairman)))
```

Revisiting unification

(aside: since this has come up on Piazza: don't forget that variables can be unified with other variables: make sure your solution handles this!)

```
(unify '(?x ?x) '(?y 42) (hash))

=> (hash '?x '?y '?y '42)
```

Formalising "the most specific instance"

Given two instances

```
instance Q1 => P1 where ...
instance Q2 => P2 where ...
```

Definition: We say P1 is *more precise* than P2 with respect to T if the number of substitutions needed when unifying T and P1 is smaller than T and P2.

Typeclass hierarchy

We have seen how typeclasses can have a **constraint** that further qualifies the typeclass instance.

In this example, the constraint relates the same typeclass to itself...

```haskell
-- JSON: JavaScript Object Notation
-- toJSON will consume a value of some
-- particular type and convert it to the string
-- representation of that type in JSON.
class JSON a where
    toJSON :: a -> String

-- Bool
instance JSON Bool where
    toJSON True = "true"
    toJSON False = "false"

-- Integer
instance JSON Integer where
    toJSON = show

-- String
instance JSON String where
    toJSON = show

-- List a
instance {-# OVERLAPPABLE #-} JSON a => JSON [a] where
    toJSON l = "[" ++ (concat (intersperse ", " (map toJSON l))) ++ "]"
```

The definition for JSON [a] depends on the definition of JSON a; if a is an instance of JSON, so too is [a]

Typeclass hierarchy

...but they need not be! Here, we see Haskell's built-in numeric typeclasses.

The most general Num class implements basic arithmetic operations, and operations requiring specific kinds of numbers are implemented in more specific typeclasses that *depend* on other typeclasses.

```
class  (Eq a, Show a) => Num a  where
    (+), (-), (*)  :: a -> a -> a
    negate         :: a -> a
    abs, signum    :: a -> a
    fromInteger    :: Integer -> a

class  (Num a, Ord a) => Real a  where
    toRational ::  a -> Rational

class  (Real a, Enum a) => Integral a  where
    quot, rem, div, mod :: a -> a -> a
    quotRem, divMod     :: a -> a -> (a,a)
    toInteger           :: a -> Integer

class  (Num a) => Fractional a  where
    (/)          :: a -> a -> a
    recip        :: a -> a
    fromRational :: Rational -> a

class  (Fractional a) => Floating a  where
    pi                  :: a
    exp, log, sqrt      :: a -> a
    (**), logBase       :: a -> a -> a
    sin, cos, tan       :: a -> a
    asin, acos, atan    :: a -> a
    sinh, cosh, tanh    :: a -> a
    asinh, acosh, atanh :: a -> a

class  (Real a, Fractional a) => RealFrac a  where
    properFraction   :: (Integral b) => a -> (b,a)
    truncate, round  :: (Integral b) => a -> b
    ceiling, floor   :: (Integral b) => a -> b
```

Figure 6

Standard Numeric Classes and Related Operations, Part 1

https://www.haskell.org/onlinereport/basic.html

Typeclass hierarchy

"For a type a to be an instance of the Real typeclass, it needs to also be an instance of the Num and Ord (orderable) typeclasses"

```
class  (Eq a, Show a) => Num a  where
    (+), (-), (*)    :: a -> a -> a
    negate           :: a -> a
    abs, signum      :: a -> a
    fromInteger      :: Integer -> a

class  (Num a, Ord a) => Real a  where
    toRational ::  a -> Rational

class  (Real a, Enum a) => Integral a  where
    quot, rem, div, mod :: a -> a -> a
    quotRem, divMod     :: a -> a -> (a,a)
    toInteger           :: a -> Integer

class  (Num a) => Fractional a  where
    (/)          :: a -> a -> a
    recip        :: a -> a
    fromRational :: Rational -> a

class  (Fractional a) => Floating a  where
    pi                  :: a
    exp, log, sqrt      :: a -> a
    (**), logBase       :: a -> a -> a
    sin, cos, tan       :: a -> a
    asin, acos, atan    :: a -> a
    sinh, cosh, tanh    :: a -> a
    asinh, acosh, atanh :: a -> a

class  (Real a, Fractional a) => RealFrac a  where
    properFraction   :: (Integral b) => a -> (b,a)
    truncate, round  :: (Integral b) => a -> b
    ceiling, floor   :: (Integral b) => a -> b
```

Figure 6

Standard Numeric Classes and Related Operations, Part 1

https://www.haskell.org/onlinereport/basic.html

Typeclass hierarchy

The "is-a" relationship in the class hierarchy looks a lot like inheritance in the OOP model!

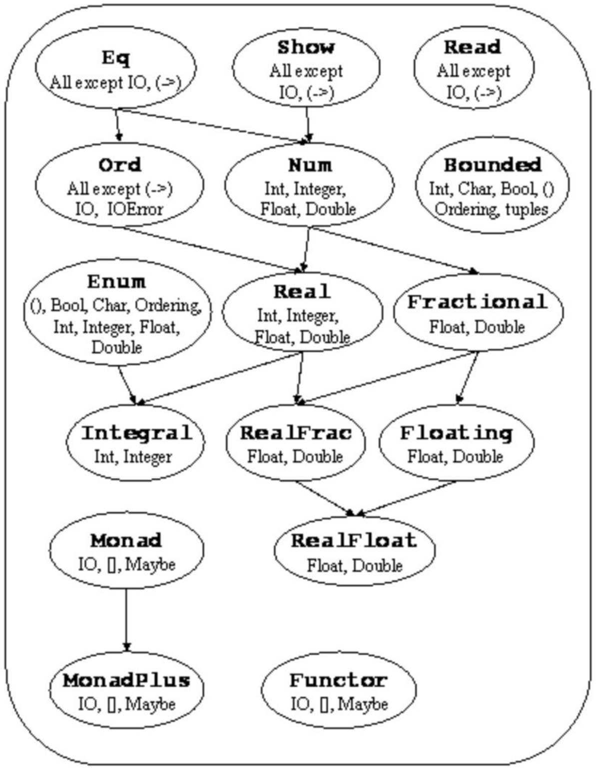


Figure 5

Standard Haskell Classes

https://www.haskell.org/onlinereport/basic.html