Comp 311
Functional Programming

Eric Allen, Two Sigma Investments
Robert “Corky” Cartwright, Rice University
Sağnak Taşırlar, Two Sigma Investments

Abstract Datatypes

```
abstract class Shape
case class Circle(radius: Double) extends Shape
case class Square(side: Double) extends Shape
case class Rectangle(height: Double, width: Double) extends Shape
```

Case 1: We Expect Few New Functions But Many New Variants

- This is a case that object-oriented programming handles well
- Classic example domains: GUI Programming, Productivity Apps, Graphics, Games
- Declare an abstract method in our superclass and provide a concrete definition for each sub-class

  a.k.a.,

  The Union Pattern (for the datatype definitions)

  The Template Method Pattern (for the function definitions)

Abstract Datatypes

```
abstract class Shape {
  def area(): Double
}

case class Circle(radius: Double) extends Shape {
  val pi = 3.14
  def area() = pi * radius * radius
}

case class Square(side: Double) extends Shape {
  def area() = side * side
}
```

How Do Abstract Classes Affect Our Type Checking Rules?

- When type checking a class definition, ensure that all abstract methods declared in the superclass are actually defined, with *compatible* method types

- When type checking a collection of class definitions, ensure that there are no cycles in the class hierarchy!

How Do Abstract Classes Affect Our Type Checking Rules?

- If a method is called on a receiver whose static type is an abstract class, extract an arrow type from the declaration (just as with a definition in a concrete class)

```
expr.area() ↦

Shape.area() ↦

() → Double
```

Type Checking Arguments to a Method Call

- The static types of an argument might no longer be an exact match:

```
abstract class Shape {
  def area(): Double

  def makeLikeMe(that: Shape): Shape
}
```

(Let us set aside the concrete definitions of `makeLikeMe` for awhile)

Now Consider a Call to Matcher With Concrete Types

```
Circle(1).makeLikeMe(Circle(2)) ⇒

Circle.makeLikeMe(Circle) ⇒

(Shape → Shape)(Circle)
```

And now we are stuck…

Recall The Substitution Model of Type Checking

- To type check the application of a function to arguments:
  - Reduce the function to an arrow type
  - Reduce the arguments, left to right, to static types
  - If the argument types match the corresponding parameter types, reduce the application to the return type

Relations

- are subsets of tuples

$$R \subseteq A \times B$$

  - reflexive

$$(a, a) \in R \quad \forall_{a \in A}$$

  - symmetric

$$(x, y) \in R \leftrightarrow (y, x) \in R \quad \forall_{x,y \in A}$$

  - anti-symmetric

$$(x, y) \in R \wedge (y, x) \in R \rightarrow x = y \quad \forall_{x,y \in A}$$

  - total

$$(x, y) \in R \vee (y, x) \in R \quad \forall_{x,y \in A}$$

  - transitive

$$(x, y) \in R \wedge (y, z) \in R \rightarrow (x, z) \quad \forall_{x,y,z \in A}$$

Some binary relations

- A total order (total, transitive, anti-symmetric)
- A partial order (reflexive, transitive, anti-symmetric)
- Functions (left covering, right unique)

Hasse Diagram

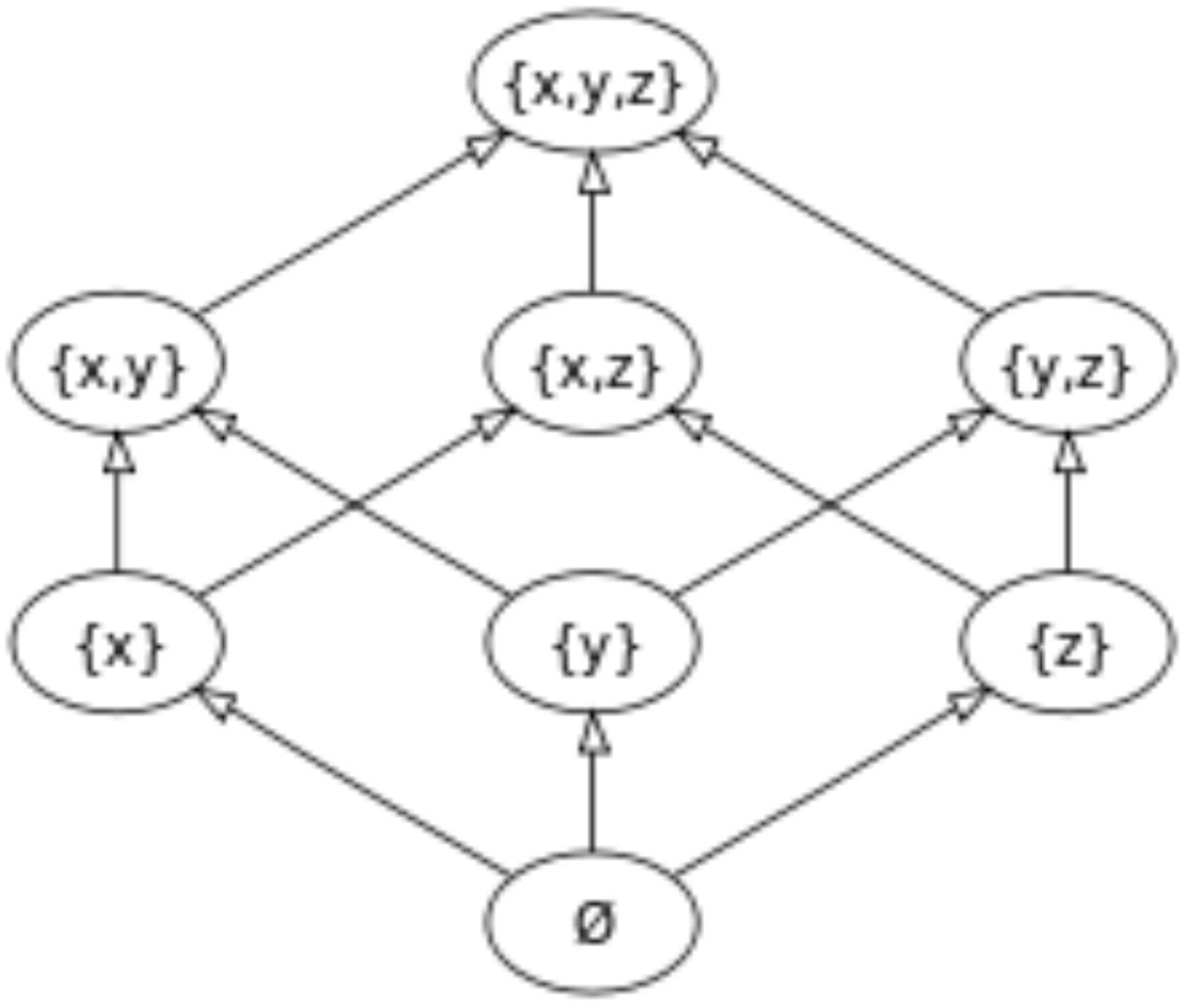


image credit: wiki article on hasse diagrams

Subtyping

- We need to widen our definition of matching a type to include subtyping:

- A class is a subtype of the class it extends

- Subtyping is Reflexive:

`A <: A`

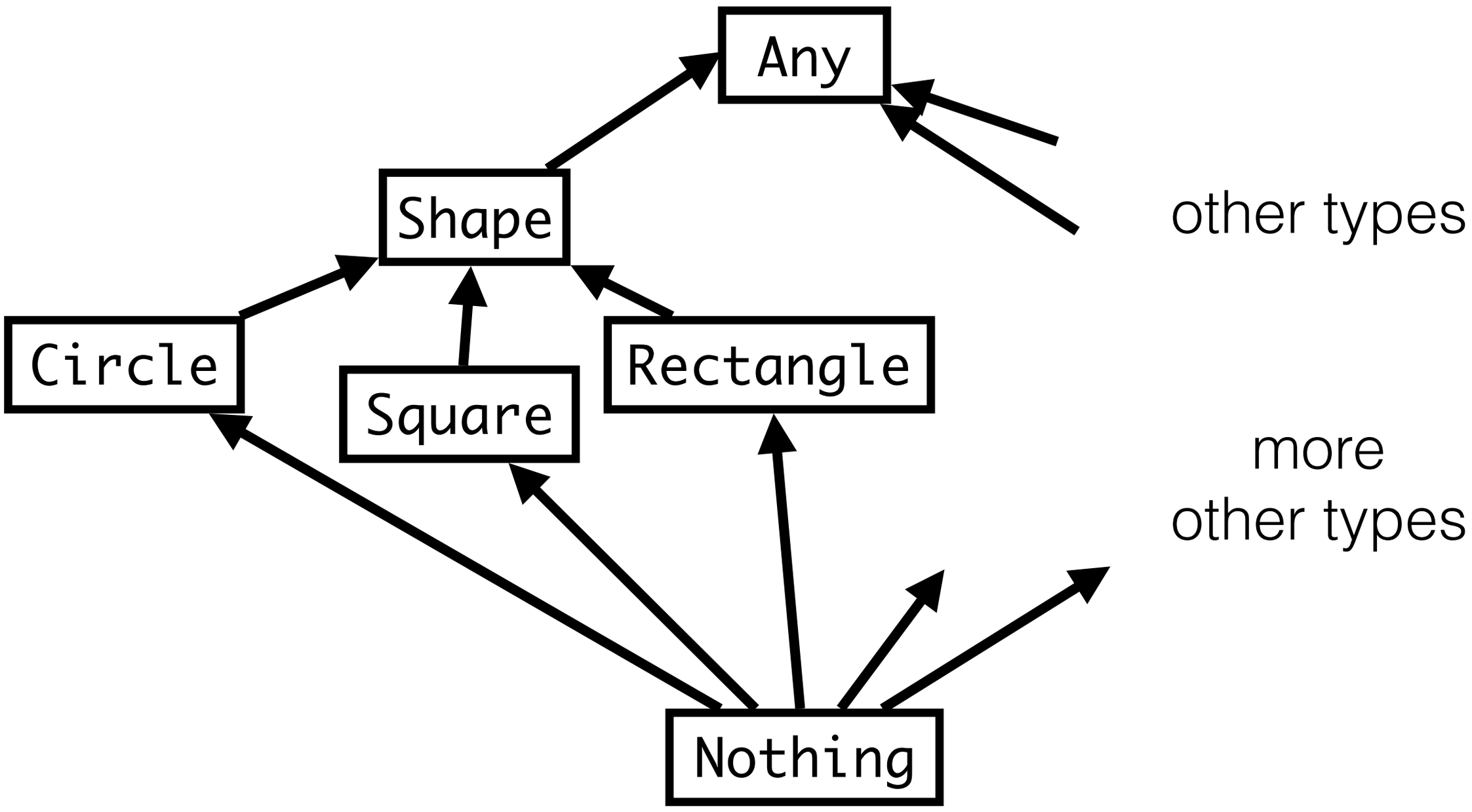
- Subtyping is Transitive:

If `A <: B` and `B <: C` then `A <: C`

Subtyping

- All types are a subtype of type **Any**
- Type `Nothing` is a subtype of all types
  - There is no value with value type `Nothing`

A scala type hierarchy



Recall The Substitution Model of Type Checking

- To type check the application of a function to arguments:
  - Reduce the function to an arrow type
  - Reduce the arguments, left to right, to static types
  - If the argument types **are subtypes of** the corresponding parameter types, reduce the application to the return type

Applying a Class Method Revisited

- To reduce the application of a method:

```
C(v1, …, vk).m(arg1, …, argN)
```

  - Reduce the receiver and arguments, left to right

  - **Find the body of m in C and reduce to that,** replacing constructor parameters with constructor arguments and method parameters with method arguments

The Body of *m*

- To find the body of method `m` in type `C`:
  - Find the definition of m in the body of `C`, if it exists
  - Otherwise, find the body of `m` in the immediate superclass of `C`

Overriding Methods

- Our new rules also handle method overriding!
- Use overriding when:
  - Factoring out a method definition common to several variants
    - Suppose several shapes compute their area in the same way
  - Augmenting the behavior of classes we do not maintain

Overriding Methods

- Scala requires that overriding method definitions include the keyword **`overrides`**

- Why require this extra keyword?

The Fragile Base Class Problem

- Suppose I define a base class `Shape`
- Later a client extends `Shape` with class `Triangle` and defines a private method `position` to record the position of one point of a triangle
- Yet later, I release a new version of my class `Shape` with a private method `position` to record the position of the center of the shape

The Fragile Base Class Problem

- This is an example of *accidental overriding*
- The `overrides` keyword catches the problem when the subclass `Triangle` is recompiled against the new version of `Shape`

Two Occasions to Consider Overriding

- The default equals methods on case classes:

```
Rational(4,6) equals Rational(2,3)
```

Two Occasions to Consider Overriding

- The default toString methods on case classes:

```
Rational(4,6) + Rational(2,3) ↦

Rational(4,3)
```

What is printed during Interactions is determined by toString

Two Occasions to Consider Overriding

- The default toString methods on case classes:

```
Rational(4,6) + Rational(2,3) ↦

4/3
```

What is printed during Interactions is determined by toString

Defining and Overriding Methods

- Recall our rule for abstract methods
  - When type checking a class definition, ensure that all abstract methods declared in the superclass are actually defined, with *compatible* method types
- We need to:
  - Augment our rule to mention overriding (this is easy)
  - Clarify “compatible method types”

Defining and Overriding Methods

- When type checking a class definition, ensure that:
  - All abstract methods declared in the superclass are actually defined, with compatible method types
  - The types of all overriding methods are compatible with the types of the methods they override

Defining and Overriding Methods

- When type checking a class definition, ensure that:
  - All abstract methods declared in the superclass are actually defined, and their types are subtypes of the method types in the corresponding declarations
  - The types of all overriding methods are subtypes of the method types they override

Arrow Types and Subtyping

- How do we define subtyping on arrow types?

- Historically this has been a painful source of bugs in object-oriented languages

Arrow Types and Subtyping

- The substitution principle of arrow typing:
- If a function **f** has type **S→T**

  and **S→T <: U→V**

  then **f** can be safely used in any context requiring a function of type **U→V**

Consider an Example

```
abstract class Shape {
  def area(): Double
  def makeLikeMe(that: Shape): Shape
}
```

- So, `makeLikeMe` has type `Shape → Shape`
- We are required to define it in all subclasses of `Shape`

Consider a Calling Context

```
def matcher(shape1: Shape, shape2: Shape) = {
  shape1.makeLikeMe(shape2).area
}
```

- What are some parameter types we can safely declare for `makeLikeMe` when defining it in class `Circle`?

- What are some return types we could safely declare?

Consider a Calling Context

```
def matcher(shape1: Shape, shape2: Shape) = {
  shape1.makeLikeMe(shape2).area
}
```

- Could class `Circle` define the parameter type of `makeLikeMe` to be `Circle`?

```
// NOT ALLOWED
def makeLikeMe(that: Circle): Shape = that
```

Consider a Calling Context

```
matcher(Circle(1), Square(1)) ↦
```

```
Circle(1).makeLikeMe(Square(1)).area ↦
```

And now we are stuck…

Consider a Calling Context

```
def matcher(shape1: Shape, shape2: Shape) = {
  shape1.makeLikeMe(shape2).area
}
```

- Could class `Circle` define the parameter type of `makeLikeMe` to be `Any`?

```
// This abides by our substitution principle
def makeLikeMe(that: Any): Shape = this
```

Consider a Calling Context

```
matcher(Circle(1), Square(1)) ↦

Circle(1).makeLikeMe(Square(1)).area ↦

Circle(1).area ↦

3.14
```

Consider a Calling Context

```
def matcher(shape1: Shape, shape2: Shape) = {
  shape1.makeLikeMe(shape2).area
}
```

- Could class `Circle` define the return type of `makeLikeMe` to be `Any`?

```
// NOT ALLOWED
def makeLikeMe(that: Any): Any = “what’s up?”
```

Consider a Calling Context

```
matcher(Circle(1), Square(1)) ↦
Circle(1).makeLikeMe(Square(1)).area ↦
“what’s up?”.area ↦
```

And now we are stuck…

Consider a Calling Context

```
def matcher(shape1: Shape, shape2: Shape) = {
  shape1.makeLikeMe(shape2).area
}
```

- Could class `Circle` define the return type of `makeLikeMe` to be `Circle`?

```
// This abides by our substitution principle
def makeLikeMe(that: Any): Circle = this
```

Consider a Calling Context

```
matcher(Circle(1), Square(1)) ↦

Circle(1).makeLikeMe(Square(1)).area ↦

Circle(1).area ↦

3.14
```

Subtyping for Arrow Types

- A type **S→T** is a subtype of **U→V** iff
  - **U** is a subtype of **S**
  - **T** is a subtype of **V**
- We say that arrow types are *contravariant* in their parameter type and *covariant* in their return type

A Limitation on Subtyping of Method Types in Scala

- Parameter types of overriding methods must match exactly in Scala

- This restriction is shared with Java and is a limitation of the JVM

- We will see other uses of arrow types in Scala where this restriction is not in place

Why Methods?

- Remember we are in Case 1: We Expect Few New Functions But Many New Variants

- How do methods help with this case?

  - All functions we support are declared in our abstract class

  - New variants can be added without changing old code:

    - Simply implement all the declared methods

Disadvantages of Methods

- If new functionality is added, every class definition must be modified to include it

Throwing And Catching Exceptions

We Can Throw and Catch Exceptions as in Java

```
def assertConstructorFail(m:Int, n:Int) = {
  try {
    Rational(m,n)
    fail()
  }
  catch {
    case e: IllegalArgumentException => {
    }
  }
}
```

Syntax For Try/Catch

```
try expr
 catch {
   case Pattern => expr
   …
}
```

Syntax For Throw

```
throw expr
```

Static Semantics For Throw

If `e` has static type `T` and

```
T <: Throwable
```

then

```
throw e
```

has static type

```
Nothing
```

Static Semantics For Try/ Catch

- Given an expression `e`:

```
try expr0
 catch {
   case Pattern => expr1
   …
   case Pattern => exprN
}
```

- Where `expr0: T0, expr1: T1, …, exprN: TN,`

- The type of `e` is the least type `T` such that:

```
T0 <: T, T1 <: T,…,TN <: T
```

Static Semantics For Try/ Catch

- The type of **e** is the least type **T** such that:

`T0 <: T, T1 <: T,…,TN <: T`

- Note that we can now use this approach to go back and define better static typing rules for `if-else` and `match` expressions

Dynamic Semantics For Throw

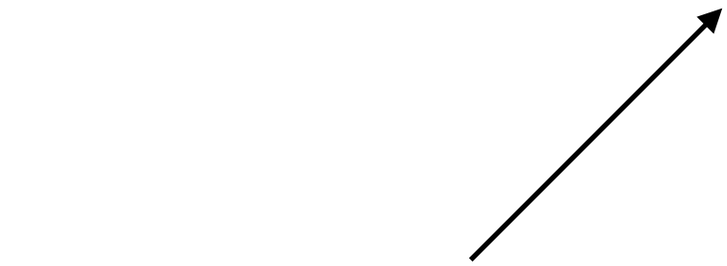
- To explain the semantics of throw, we must introduce new terminology
- Let the *continuation* of an expression **e** refer to all that remains to be done in a computation after **e** is reduced
- We can think of a continuation as an expression with a "hole" in it, corresponding to **e**
- Equivalently, we can think of a continuation as function that takes a parameter, corresponding to the result of evaluating **e**

Example Continuation

```
matcher(Circle(1), Square(1))
```

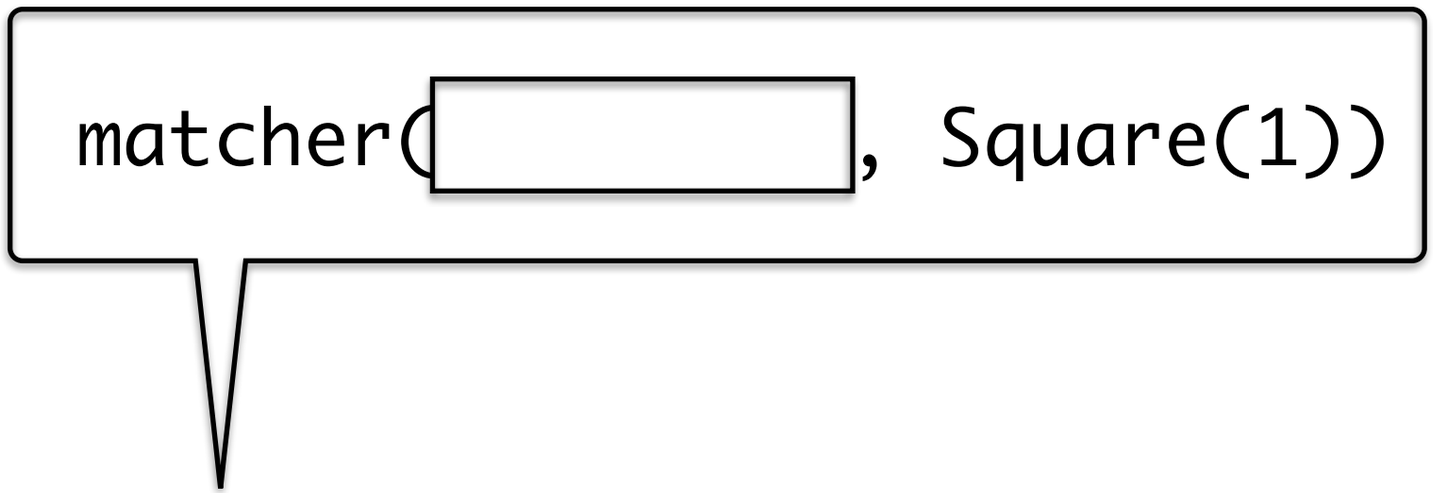
Example Continuation

```
matcher(Circle(1), Square(1))
```



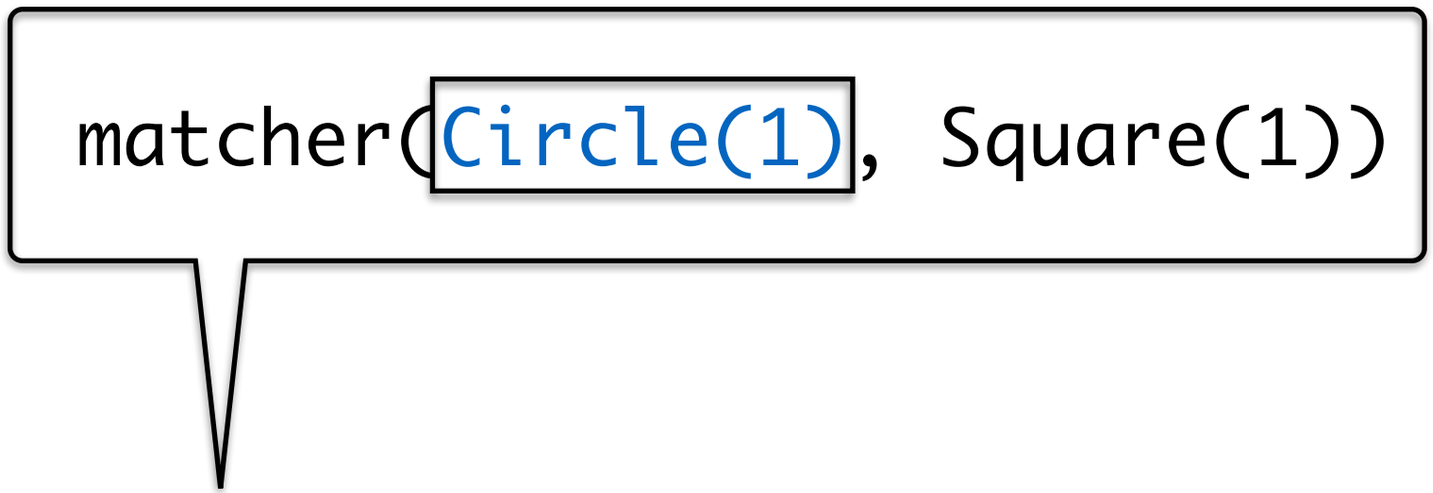
Let this be our expression **e**

Example Continuation



Then this is the continuation of **e**

Example Continuation



Once e is reduce to a value, the box is filled in, and the continuation can be reduced

Reducing a Throw Expression

- To reduce a throw expression:

`throw e`

  - Reduce **`e`** to a value **`v`**
  - Replace the continuation of the throw expression with the special expression **`throw v`**

Reducing a Try/Catch

- To reduce a try/catch expression:

```
try expr0
 catch {
   case Pattern => expr1
    …
   case Pattern => exprN
}
```

Reducing a Try/Catch

- Set aside the continuation `C` of the `try/catch`
- Reduce the body of the `try` in a special continuation `D`
- If `D` reduces to `throw v`:
  - Restore the continuation `C`
  - Try matching `v` against each pattern in the `catch` clause
  - If a match is found, evaluate the body of the matching case
  - Otherwise, reduce to `throw v`
- If `D` reduces to `w`, restore continuation `C` and reduce the `try/catch` to `w`

Consider Our Motivating Test Helper Function

```
def assertConstructorFail(m:Int, n:Int) = {
  try {
    Rational(m,n)
    fail()
  }
  catch {
    case e: IllegalArgumentException => {
    }
  }
}
```

We Call Our Function In An Enclosing Context

```
enclosingProgram (
   assertConstructorFail(1,0)
)
↦
enclosingProgram (
    try {
      {require(0 != 0); Rational(1,0)}
      fail()
    }
    catch {
      case e: IllegalArgumentException => {}
    }
)
↦
```

Continuation C

D

```
enclosingProgram (
     try {
       {require(0 != 0); Rational(1,0)}
       fail()
     }
     catch {
       case e: IllegalArgumentException => {}
     }
)
```

↦

```
{
  {require(0 != 0); Rational(1,0)}
  fail()
}
```

C

↦

```
{
  {require(0 != 0); Rational(1,0)}
  fail()
}
```

C

↦

```
{
  {throw IllegalArgumentException; Rational(1,0)}
  fail()
}
```

C

↦

```
throw IllegalArgumentException
```

C

↦

```
throw IllegalArgumentException
↦
enclosingProgram (
    try {
      throw IllegalArgumentException
    }
    catch {
      case e: IllegalArgumentException => {}
    }
)
↦
enclosingProgram (
    {}
)
↦
enclosingProgram ()
```

C

What If Our Catch Clause Does Not Match?

C

```
throw IllegalArgumentException
↦
enclosingProgram (
    try {
      throw IllegalArgumentException
    }
    catch {
      case e: AssertionError => {}
    }
)
↦
enclosingProgram (
    throw IllegalArgumentException
)
↦
throw IllegalArgumentException
```

Continuations Are A Recurrent Concept in Computer Science

- Distributed computing
- Parallel computing
- Operating systems
- A unified approach to control flow

The Assert Function

```
assert: Boolean → Unit

assert: (Boolean, String) → Unit
```

- Note that the function is overloaded
- Use inside functions to ensure properties hold
- Do not assert unless you actually believe the assertion is true!

Type Checking Overloaded Functions

- For each overloaded declaration of a function f:
  - Provide that declaration with a fresh name, in a manner that respects method overriding

```
abstract class Shape {
  def area(): Double

  def makeLikeMe(that: Int): Shape
  def makeLikeMe(that: Shape): Shape
}
```

Type Checking Overloaded Functions

- For each overloaded declaration of a function f:
  - Provide that declaration with a fresh name, in a manner that respects method overriding

```
abstract class Shape {
  def area(): Double

  def makeLikeMe$Int(that: Int): Shape
  def makeLikeMe$Shape(that: Shape): Shape
}
```

Type Checking Overloaded Functions

- For each overloaded declaration of a function f:
  - Provide that declaration with a fresh name, in a manner that respects method overriding

```
case class Circle(radius: Int) {
  val pi = 3.14
  def area(): Double = pi * r * r

  def makeLikeMe$Int(that: Int): Shape = this
  def makeLikeMe$Shape(that: Shape): Shape = that
}
```

Type Checking an Overloaded Function

- When an overloaded function is called on an argument expression e with type T:
  - If there is a unique matching function definition whose parameter type is:
    - A supertype of T
    - A subtype of all other matching definitions
- Replace the function name with the unambiguous name for that unique function

Reducing an Overloaded Function Definition

- Because of the rewrite during type checking, our reduction rules need no modification!