ADVANCED PROGRAMMING LANGUAGES
Fall 2024, CM 0632. MSc in Computer Science and Information Technology, Ca' Foscari University of Venice

ADTs in Programming Languages

We complete the discussion on algebraic datatypes with a deeper dive on their implementation in different programming languages

Nominal vs Structural Type Systems

Following a common practice in type theoretic datatypes presentations, so far we have adopted a purely *structural* presentation that introduces new type expressions reflecting the structure of the datatypes the type expressions are associated with.

In programming languages, datatypes are most often introduced with names to stand for long or complex type expressions to improve the readability of examples. e.g.:

$$\begin{array}{rcl} \mathsf{IntPair} & = & \{\mathsf{fst} : \mathsf{Int},\ \mathsf{snd} : \mathsf{Int}\} \\ \mathsf{Address} & = & \{\mathsf{city} : \mathsf{String},\ \mathsf{street} : \mathsf{String},\ \mathsf{number} : \mathsf{Int}\} \end{array}$$

Such definitions are sometimes pure abbreviations: the names are just shorthands for the type expressions and they are interchangeable with the type expressions in any context.

By contrast, programming languages most often adopt a different approach, known as *nominal typing*. In a nominal type system:

- every compound type must be explicitly introduced by a declaration before being used;
- each type name must be declared at most once, and following their declarations, types must always be referred to by their names

Nominal type systems have both advantages and disadvantages over structural presentations. The most significant advantage is probably that the type names in nominal systems are useful not only during type-checking, but at run time as well. Most nominal languages tag each run-time object with a header containing its type name, represented concretely as a pointer to a run-time data structure describing the type. These type tags are handy for a variety of purposes, including run-time type testing (e.g., Java's *instanceOf* tests and *downcasting*), printing, marshaling data structures into binary forms, . . . etc.

A less essential, but pleasant, property of nominal systems is that they offer a natural and intuitive account of recursive types. Such types are ubiquitous in serious programming, being required to describe such common structures as lists and trees. Nominal type systems support them in the most straightforward possible way: referring to List in the body of its own declaration is just as easy as referring to any other type.

Structural systems, in turn, are appealing in that they are somewhat tidier and more elegant. In a structural setting, a type expression is a closed entity: it carries with it all the information that is needed to understand its meaning (by contrast, in a nominal system, we are always working with respect to some global collection of type name-definition pairs). Recursive types can also be handled structurally, by introducing type-level recursive operators (e.g. fix or rec operators on types), though capturing complex schemes of mutually recursive definitions becomes somewhat heavy with such operators.

All in all, the usefulness of type tags and the simple treatment of recursive types are decisive benefits for the practice of programming, and hence it is no surprise to find that nominal type systems are the norm in mainstream programming languages, with few exceptions.

Structural typing in SML. Standard ML (SML) is an early dialect of the ML language that adopts a purely structural type systems. In SML record types are introduced just as we have done above. In fact, SML provides a mechanism for type declarations, but (i) such declarations are not mandatory to introduce a type and (ii) when they are used, they introduce type names as simple *aliases* for the record type they are associated with. Thus, the following are all legal in SML:

```
type IntPair = { fst: Int, snd: Int }
type Address = { city : String, street : String, number : Int }

val p1 : IntPair = { fst = 2, snd = 3 }
val p2 : { snd: Int, fst: Int }  = p1
val a1 : Address = { city = "Boston", street = "Worcester", number = 15 }
val a2 : { city : String, street : String, number : Int } = a1
```

Nominal Typing in Haskell

Record types. Haskell record types are product types with additional syntax to provide a convenient naming (and access) scheme for the product positions. For example the following is the Haskell version of the datatypes defined earlier:

```
data Pair = Pair { fst :: Int, sdn:: Int }
data Address = Addr { city :: String, street :: String, number :: Int }

let p = Pair { fst = 5, snd = 3 }
let a = Addr { city = "Boston", street = "Worcester", number = 15 }
```

Both types are introduced by a `data` declaration, which defines the new type name introduces the data constructor to build the elements of the type's domain: for the type `Pair` the type and data constructors have the same name, whereas for addresses `Address` is the type constructor, while `Addr` is the data constructor.

Types introduced by a `data` declaration are unique and so are the data constructors introduced with them: the labels used to identify the record type fields are also unique, and may not be reused in other (record) types.

Type aliases also exist in Haskell, to create synonyms for existing types names or expressions: they are introduced by the keyword `type` : notice, however, that unlike `data` declarations, `type` declarations do not introduce any data constructor, hence no new data domain, only a new type name for an existing data domain. In that respect, Haskell `type` declarations are much like `typedef` declarations in C.

Variant Types Haskell variant types are defined by a `data` declaration which introduces the new type constructor and one or more domain constructors, each of which may have zero or more arguments.

```
type Radius  = Double
type Side    = Double
data Shape   = Circle Radius | Rectangle Side Side

data IntList = Empty | Cons Int IntList
```

The first two type definitions introduce two name aliases for the type `Double`: their purpose is merely to document the intended meaning of the arguments of the two data constructors `Circle` and `Rectangle` associated with the type `Shape`. As a result of the declaration, the type `Shape` is univocally identified by its name, and circles and rectangles built with the data constructors may only be associated with type `Shape`. This also solves the tagging problem with stuctural typing and variant types we discussed in the last lecture.

`IntList` is a further datatype, which illustrates the use of data constructors with no argument, like `Empty` in a recursive definition of a type `IntList` which represents lists of integers: as anticipated, the use of nominal types makes the definition of recursive types particularly simple and natural. Given the declarations above, shapes and integer lists may be expressed simply as follows:

```
let a_circle : Shape = Circle 3.0
let a_list : IntList = Cons 5 (Cons 3 (Cons 4 Empty))
```

Nominal typing in Scala

Records as (case) classes In Scala, like in other object oriented languages (most notable Java), records and record types are somehow subsumed by objects and classes. There is a fundamental difference between objects and records in that methods of objects have access to the objects' structure (other methods and fields) whereas records simply collect fields without providing any cross-referencing mechanism among them. The key to this dichotomy is the implicit parameter this that object methods can count on as a primitive mechanism to access the hosting object as a self-reference.

Having said that, recent versions of Java have introduced special constructs to provide for a representation of data structures fully coherent with the typing of records found in other functional programming languages. Scala also has a construct that severs the purpose: *case classes*,

Case classes are like regular classes with a few special features that make them effective for modelling algebraic datatypes. We illustrate such features with the Scala implementation of the datatypes discussed above.

```
case class Pair(fst: Int, snd : Double)
case class Address(city : String, street :String, number : Int)
```

The two declarations introduce two cases classes whose name serves both as type and as domain constructor. For each case class, Scala automatically generates a primary data constructor so that case class instances may be constructed using the class name as a constructor (without `new`). As a result, the following two declarations introduce, respectively a `Pair` and an `Address`

```
val p = Pair(5,3)
val a = Address("Boston", "Worcester", 15)
```

Case classes have a variety of useful features that make them different from standard classes and construe them as a faithful proxy of records in functional languages: they are immutable, they come with a neat implementation for equality (instances of case classes are always compared structurally), and with a number of additional useful operators and methods.

Most notably, however, as Odersky, Spoon, and Venners note in their Scala book, “the biggest advantage of case classes is that they support pattern matching.”

Variant Types with Enumerations In Scala, variant datatypes may be defined either with `enum` declarations or with systems of case classes. We look at enumerations first, as they provide for a more convenient and intuitive representation than systems of case classes:

```
enum Shape:
   case Circle(radius : Double)
   case Rectangle(width : Double, height: Double)

enum IntList:
   case Empty
   case Cons(hd : Int, tl: IntList)
```

As we see, Scala enumerations are very much like datatypes definition in Haskell: the `enum` clause introduces a type name (the constructor of the new type) and the subsequent `case` clauses define the data constructors for the type domain. In fact, as we will show in the upcoming lecture, `enum` declarations turn out to be just (rather convenient) syntactic sugar for systems of case classes in which the type introduced in the `enum` clause is the common super-type of the types associated with the cases classes that implement the `case` clauses of the enumeration. The implementation of enumerations with cases classes is almost transparent, except for one detail: to access the data constructors associated with the `enum` declaration, one has to explicitly get them in scope with an explicit *import* declaration.

Given the two enumerations, we may thus use them to build shapes and integer lists as follows:

```
import Shape.{Circle, Rectangle}
import IntList.{Empty, Cons}

val a_circle : Shape = Circle 3.0
let a_list : IntList = Cons(5,Cons(3,Cons(4, Empty)))
```

A detour on Scala classes

Just as in Java, Scala classes are the blueprint (and the type) for objects: once we define a class, we can create objects from the class using the keyword `new`. A class definition can contain field and method declarations: the former (declared as modifiable `var` iable or immutable `val` ues) hold the data of the objects of the class, while methods, introduced with the keyword `def` contain executable code that has access to the class objects' fields.

Here is an example class definition for a person:

```
class Person(val name: String, val age: Int) :
  def appendToName(suffix:String) : Person = new Person(s"$name$suffix", age)

val person = new Person("Giovanni",53)
```

This `Person` class has three members: the fields `name` and `age` and the method `appendToName` . Unlike many other OO languages, the class definition includes the signature `(val name: String, val age: Int)` of the primary constructor (other constructors may be defined in the companion object, see below). The `appendToName` method takes a string argument and returns another `Person`

Parameters without `val` or `var` are private values, and visible only within the class. Class members declared inside the class are public by default. The `private` access modifier to hide them from outside of the class. As a result, to make a field private, it must be declared inside the class rather than via a constructor parameter.

Side remark. Here, and throughout, we insist on functional (stateless / immutable) data structures. Having noted that, we remark that Scala provides full support for stateful object, whose state is defined by `var` tagged parameters for the primary constructor (or equivalently, by `var` declaration of fields) just as in Java.

Companion Objects A *companion object* is an `object` that is declared in the same file as a `class` , and has the same name as the class. Companion objects are special cases of *singleton objects*, the mechanism that Scala provides to account for Java's static methods: whereas Java's static methods are defined within classes, in Scala they are defined as regular methods of singleton objects.

Companion objects come with a variety of interesting features. First, a companion object and its class can access each other's private members (fields and methods). Because of that, companion objects provide the natural scope to define methods (in fact, functions) that operate on objects of the companion class. Among such methods, a companion object may define an `apply` method to enable instances of the companion class to be created without the `new` keyword. What happens is that when you define an `apply` method in a companion object, it has a special meaning to Scala, which in turn activates some useful syntactic sugar that lets you type code such as:

```
val person = Person("Giovanni",53)
```

and during the compilation process (or the REPL) Scala turns that code into this code:

```
val p = Person.apply("Giovanni",53)
```

The `apply` method in the companion object acts as a *factory method*, and Scala's syntactic sugar lets you use the syntax shown, creating new class instances without using the `new` keyword.

To demonstrate how this feature works, here's a companion object for class `Person` along with an `apply` method that makes sure that `Person` s are initialized with their name (if any) capitalized.

```
object Person:
   def  apply(name: String, age:Int) : Person =
           val capsName =
                   if !nameisEmpty then
                           val first = name.charAt(0).toUpper
                           val rest = name.substring(1)
                           s"$first$rest"
                   else "Unknown"
            new Person(capsName, age)
```

The following is the implementation of the `Shape` type by a system of cases classes.

```
sealed trait Shape
case class Circle(radius: Double) extends Shape
case class Rectangle(base: Double, height: Double) extends Shape
```

The first line introduces a *trait*. Scala's traits are similar to Java's interfaces and abstract classes, and serve a similar purpose, namely to define a common interface (possibly with default method implementations) for a hierarchy of types. In this case, the trait defines a new type *Shape* (with empty interface) and is *sealed*, meaning that it cannot be extended outside the scope of the present declaration. Sealing provides is a form of protection for class hierarchies similar to that offered by `final` in Java: in addition, when associated with datatype declarations such as the one above, it helps the type-checker verify the use of pattern matching on the case class instances of the trait.

The subsequent two declarations introduce two cases classes as a concrete implementation of the trait: the subtyping relation introduced by the `estends` clause make `Circle` and `Rectagle` subtypes of `Shape` . As we noted earlier on, the class names serve both as types and as data constructors: this is possible thanks to the specific features that Scala provides for cases classes, as we discuss below.

Primary constructors. First, for each case class, Scala automatically generates a companion object and defines a factory method for the primary constructor so that case class instances may be constructed using the class name as a constructor (hence, without `new`). As a result, the following two declarations introduce, respectively a `Circle` and a `Rectangle` : by subtyping, both are also instances of `Shape` .

```
enum Shape:
   case Circle(radius : Double)
   case Rectangle(width : Double, height: Double)

val circle = Circle(3.0)
val rectangle = Rectangle(5.0, 18.0)
```

Notice also that all arguments in the parameter list are listed without qualification, hence they all get a `val` qualification, which makes the immutable, automatically.

Equality and other utility methods. In addition to the factory method, Scala adds further definitions for methods `toString` , `hasCode` and, perhaps more interestingly, `equals` : the latter, in particular, will compare instances by inspecting the whole structure of the instances. Since Scala always delegates `==` to `equals`, this means that element of cases classes are always compared structurally: in that respect, they can be considered a faithful representation of records.

While all of these features are great benefits to functional programming, as they write in the book, Odersky, Spoon, and Venners note that “the biggest advantage of case classes is that they support pattern matching.” Pattern matching is a major feature of FP languages, and Scala’s case classes provide a simple way to implement pattern matching in match expressions and other areas.

Pattern matching

We illustrate pattern matching in Scala as a representative of a mechanism found in many programming language. Pattern matching in Scala is an extension of the foundational mechanism illustrated in our last lecture, which accommodates a variety of pattern forms. A `match` expression

```
exp match
  case p1 => exp1
  ...
  case pn => expn
```

evaluates the expression `exp` to be matched and then tries each of the patterns `p1` to `pn` in the order they are listed. The first pattern that matches is selected, and the corresponding expression `expi` following the arrow is evaluated: no other match is attempted, (i.e. the execution “falls through” into the next case.) If none of the patterns match, an exception `MatchError` is thrown. If the cases of a match do not cover the type of the expression to be matched, the type checker will issue a warning.

There exist many kinds of patterns in Scala other than the variable and constructor patterns we have discussed.

Matching on constant patterns. Constant patterns are simply a special case of constructor patterns. The following code snippet shows some examples:

```
def describePatterns(x:Any) =
 x match
   case 5 => "five"
   case true => "true"
   case "hi" => "hello"
   case Nil  => "empy list"
```

Matching on case classes Matching on case classes is also a special use case of matching with constructors. In fact, it coincides with the corresponding matching expression for enumerations:

```
def area(s: Shape): Double = s match
  case  Circle(r) => Math.PI * Math.pow(r,2)
  case  Rectangle(b,h) => b*h
```

If the root trait of the class hierarchy is sealed, as for type `Shape`, then the type checker can verify whether or not the cases of a match on root type is exhaustive and issue the warning accordingly (that is because sealing ensures the type checker that no further case classes may extend the root outside the file being checked).

matching with sequence patterns. These are further special cases of constructor patterns over the types `List` and `Array`. As an example, the following code snipped isolates three-element lists that start with a 0:

```
xs match
  case List(0,_,_) => ...
  case _ => ...
```

Alternatively, we can match against a sequence without specifying how long it can be, by introducing `_*` as the last pattern of the sequence pattern:

```
xs match
  case List(0,_*) => ...
  case _ => ...
```

Typed patterns. Typed patterns represent a convenient replacement for type tests and type casts. We show an example below:

```
def perimeter(s: Shape : Double = s match
    case c : Circle => 2 * Math.PI * c.radius
    case r : Rectangle => 2 * (r.base + r.height)
```

The code is similar to the one above, but notice that use of the typed annotations to test the "actual" type of `s` and effectively cast it to that type, once the test has succeeded.

Pattern guards. Patterns in Scala (as ML and Haskell) are *linear*, which means that each variable may occur in a pattern at most once. The only exception is the wildcard, whose occurrences, however, are never related to one another. Pattern guards provide a mechanism to get around this constraint.

```
def isSquare(s: Shape) : Boolean = s match
            case Rectangle(b,h) if b == h => true
            case _ => false
```

Variable binding. In addition to standalone variable patterns, variables may also be added to other patterns as in `x@p` where `x` is the variable and `p` the pattern: the effect is to match the pattern as described above in addition to binding the variable to the whole value being matched.

Patterns in variable definitions. Patterns can be used anywhere a `var` or `val` declaration would be legal. The following example introduces a tuple pattern in a declaration. Assuming `tuple : (Int, String)` the following destructs the tuple and binds the two pattern variables:

```
val (number, string) = tuple
```

Case sequences in function definitions A last interesting use of patterns is in case sequences to be used anywhere a function literal can be used. Essentially, a case sequence *is* a function literal, only with more than one entry point, each specified with a corresponding pattern. To illustrate, the `area` function defined earlier can be defined as follows:

```
def area : Shape => Double =
   case  Circle(r) => Math.PI * Math.pow(r,2)
   case  Rectangle(b,h) => b*h
```

As-patterns in Haskell
Case sequences in function definitions are provided as a common programming practice in Haskell. Sometimes it is convenient to name a pattern for use on the right-hand side of an equation. For example, the `area` function above could be written as follows:

```
data Shape   = Circle Radius | Rectangle Side Side
area (Circle r) = pi * (r**2)
area (Rectangle b h) = b*h
```

This is a special case of the general form of a function definition by cases, which in Haskell may take the form:

$$\begin{array}{l} \mathrm{f}\ \mathrm{pattern}_{11} \ldots \mathrm{pattren}_{1k} = \mathrm{expr}_1 \\ \ldots \\ \mathrm{f}\ \mathrm{pattern}_{n1} \ldots \mathrm{pattern}_{nk} = \mathrm{expr}_n \end{array}$$

and is is semantically equivalent to the following definition using a case expression:

$$\begin{array}{l} \mathrm{f}\ x_1\ x_2 \ldots x_k = \mathrm{case}\ (x_1, \ldots, x_k)\ \mathrm{of} \\ \quad p_{11}, \ldots p_{1k} \to e_1 \\ \quad \ldots \\ \quad p_{n1}, \ldots p_{nk} \to e_n. \end{array}$$

Datatypes vs Objects in Scala

We have already discussed the role of Scala classes in the implementation of records and variants. Below we explore the relationship between (case) classes and objects.

We start by introducing an alternative implementation of the `Shape` type by a system of cases classes.

```
sealed trait Shape
case class Circle(radius: Double) extends Shape
case class Rectangle(base: Double, height: Double) extends Shape
```

The first line introduces a *trait*. Scala's traits are similar to Java's interfaces and abstract classes, and serve a similar purpose, namely to define a common interface (possibly with default method implementations) for a hierarchy of types. In this case, the trait defines a new type *Shape* (with empty interface) and is *sealed*, meaning that it cannot be extended outside the scope of the present declaration. Sealing provides is a form of protection for class hierarchies similar to that offered by `final` in Java: in addition, when associated with datatype declarations such as the one above, it helps the type-checker verify the use of pattern matching on the case class instances of the trait.

The subsequent two declarations introduce two cases classes as a concrete implementation of the trait: the subtyping relation introduced by the `estends` clause make `Circle` and `Rectagle` subtypes of `Shape` . As we noted earlier on, the class names serve both as types and as data constructors: this is possible thanks to the specific features that Scala provides for cases classes, as we discuss below.

The definitions would typically be collected into a Scala package as follows:

```
package FunShapes

sealed trait Shape
case class Circle(radius: Double) extends Shape
case class Rectangle(base: Double, height: Double) extends Shape

def area(s: Shape): Double = s match
  case Circle(r) => Math.PI * Math.pow(r,2)
  case Rectangle(b,h) => b*h

def scale(s: Shape, f: Double) : Shape = s match
  case Circle(r) => Circle(r*f)
  case Rectangle(b,h) => Rectangle(b*f,h*f)
```

The presentation is in purely *functional style*, with the system of trait and case classes serving as data structure definitions packaged with a set of functions for working with `Shape` s.

A corresponding *object-oriented presentation* of the `Shape` datatype may be given as follows:

```
package OOShapes

sealed trait Shape:
  def area : Double
  def scale(f:Double): Shape

case class Circle(radius: Double) extends  Shape:
  def area: Double = Math.PI * Math.pow(radius,2)
  def scale(f:Double) : Shape = Circle(radius*f)

case class Rectangle(base: Double, height: Double) extends  Shape:
  def area: Double = base * height
  def scale(f:Double) : Shape = Rectangle(base*f, height*f)
```

Again, the presentation introduces the system of trait and case classes to serve as data structure definitions. Unlike the functional presentation, however, the `Shape` operations are encoded as methods specified by the trait declaration and implemented by the two case classes implementing `Shape` . We will return to this later on in the course, when we discuss subtyping and Scala's mechanisms for composition and inheritance.

Exercises

1. Compound Datatypes in Rust. Find out how ADTs are implemented in the programming language Rust. Specifically, provide answers to the following questions:

- how are records and variants represented in Rust

- how is pattern matching implemented?
- does Rust adopt a nominal or structural approach to typing?

2. Lambda Calculus representation in Scala We have covered enough material now to start coding some concrete example. In this exercise, you are asked to build a Scala representation for the abstract syntax of the Lambda Calculus as defined by the following productions:

$$
\begin{array}{rcll}
M, N, P & ::= & x, y, z & \text{variable} \\
 & \mid & \lambda x@T.M & \text{abstraction} \\
 & \mid & M\ N & \text{application} \\
 & \mid & n, m, k & \text{integers} \\
 & \mid & \mathsf{true} \mid \mathsf{false} & \text{booleans} \\
 & \mid & M + N \mid M = N & \\
 & \mid & \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P & \text{conditional} \\
 & \mid & \mathsf{fix}@T.M & \text{recursion}
\end{array}
$$

Define a type `Lambda` representing the abstract syntax for the set terms defined by the previous productions. *Hint*: Use an enumeration or a system of cases classes.

3. Lambda Calculus Reduction Write a set of functions to simulate the reduction relation. More precisely, define:

- a function `fv` to compute the free variables of a term;
- a function `subst` that implements the substitution of a variable for with a term in another, given term;
- a function `reduce` that implements the reduction relation, i.e. a function `reduce` such that `reduce(M) = N` whenever $M \to N$ is derived by the rule system below, where `M` and `N` are the Scala representations of the lambda terms M and N respectively.

Numeric Reductions

$$
\frac{(m \text{ and } n \text{ numeric literals})}{n + m \to n +_N m} \qquad \frac{M \to M'}{M + m \to M' + m} \qquad \frac{M \to M'}{m + M \to m + M'}
$$

Boolean Reductions

$$
\frac{(a \text{ and } b \text{ different constants})}{a = b \to \mathsf{false}} \qquad \frac{}{M = M \to \mathsf{true}} \qquad \frac{M \to M'}{M = N \to M' = N} \qquad \frac{M \to M'}{N = M \to N = M'}
$$

$$
\frac{}{\mathsf{if\ true\ then}\ M\ \mathsf{else}\ \ N \to M} \qquad \frac{}{\mathsf{if\ false\ then}\ M\ \mathsf{else}\ \ N \to N}
$$

$$
\frac{M \to M'}{\mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ \ P \to \mathsf{if}\ M'\ \mathsf{then}\ N\ \mathsf{else}\ \ P}
$$

Recursion Reductions

$$
\frac{}{\mathsf{fix}@T.M \to M(\mathsf{fix}@T.M)} \qquad \frac{M \to M'}{\mathsf{fix}@T.M \to \mathsf{fix}@T.M'}
$$

Lambda Reductions

$$\frac{}{(\lambda x.M)N \to [N/x]M} \qquad \frac{M \to M'}{M\,N \to M'\,N} \qquad \frac{N \to N'}{M\,N \to M\,N'} \qquad \frac{M \to M'}{\lambda x.M \to \lambda x.M'}$$

4. Lambda Calculus Typing Write a function that implements the type checking relation, i.e. i.e. a function `typecheck` such that `typecheck(G,M) = T` whenever $\Gamma \vdash M : T$ is derived by the rule system below, where `G`, `M` and `T` are the Scala representations of the typing context Γ, the term M and the type T, respectively.

Lambda Terms

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x : T_1 \vdash M : T_2}{\Gamma \vdash \lambda x@T_1.M : T_1 \to T_2} \qquad \frac{\Gamma \vdash M : S \to T \qquad \Gamma \vdash N : S}{\Gamma \vdash M\,N : T}$$

Recursion

$$\frac{\Gamma \vdash M : T \to T}{\Gamma \vdash \mathsf{fix}@T.M : T}$$

Natural numbers and booleans

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash n : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash M : \mathsf{Nat} \quad \Gamma \vdash N : \mathsf{Nat}}{\Gamma \vdash M + N : \mathsf{Nat}} \qquad \frac{\Gamma \vdash M : T \quad \Gamma \vdash N : T}{\Gamma \vdash M = N : \mathsf{Bool}} \qquad \frac{\Gamma \vdash M : \mathsf{Bool} \quad \Gamma \vdash N : T \quad \Gamma \vdash P : T}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P : T}$$

CREDITS

The discussion on nominal vs structural typing is taken from Chapter 19.3 of Prof. Benjamin's Pierce's book on Types and Programming Languages. MIT Press. 2001.

The discussion on Scala datatypes (and objects) is based on Chapter 4, 13 and 19 of *Programming in Scala (5th Edition)*. M. Odersky, L. Spoon, B. Venners, F. Sommers. Artima Press. 2021. **Please read those chapters** for further information about patterns and pattern matching.