### Overview

- **Last time:**
  - Simple data structures: pairing (product types), choice (sum types)

- **Today:**
  - Records (generalizing products), variants (generalizing sums) and pattern matching
  - Subtyping

### Records

- **Records** generalize pairs to \(n\)-tuples with *named* fields.

\[
\begin{align*}
e &::= \cdots | \langle l_1 = e_1, \ldots, l_n = e_n > e.l \\
v &::= \cdots | \langle l_1 = v_1, \ldots, l_n = v_n \\
\tau &::= \cdots | \langle l_1 : \tau_1, \ldots, l_n : \tau_n >
\end{align*}
\]

- **Examples:**

\[
\langle \text{fst}=1, \text{snd}="forty-two" \rangle . \text{snd} \mapsto "forty-two"
\]
\[
\langle x=3.0, y=4.0, \text{length}=5.0 \rangle
\]

- Record fields can be (first-class) functions too:

\[
\langle x=3.0, y=4.0, \text{length}=\lambda(x,y). \text{sqrt}(x*x + y*y) \rangle
\]

### Named variants

- As mentioned earlier, *named variants* generalize binary variants just as records generalize pairs.

\[
\begin{align*}
e &::= \cdots | C_i(e) | \text{case } e \text{ of } \{ C_1(x) \Rightarrow e_1; \ldots \} \\
v &::= \cdots | C_i(v) \\
\tau &::= \cdots | [C_1 : \tau_1, \ldots, C_n : \tau_n]
\end{align*}
\]

- Basic idea: allow a choice of \(n\) cases, each with a name

- To construct a named variant, use the constructor name on a value of the appropriate type, e.g. \(C_i(e_i)\) where \(e_i : \tau_i\)

- The case construct generalizes to named variants also
Named variants in Scala: case classes

- We have already seen (and used) Scala’s case class mechanism

```scala
abstract class IntList
case class Nil() extends IntList
case class Cons(head: Int, tail: IntList) extends IntList
```

- Note: IntList, Nil, Cons are newly defined types, different from any others.
- Case classes support pattern matching

```scala
def foo(x: IntList) = x match {
  case Nil() => ... 
  case Cons(head,tail) => ...
}
```

Aside: Records and Variants in Haskell

- In Haskell, data defines a recursive, named variant type

```haskell
data IntList = Nil | Cons Int IntList
```

- and cases can define named fields:

```haskell
data Point = Point {x :: Double, y :: Double}
```

- In both cases the newly defined type is different from any other type seen so far, and the named constructor(s) can be used in pattern matching
- This approach dates to the ML programming language (Milner et al.) and earlier designs such as HOPE (Burstall et al.).
  - (Both developed in Edinburgh)

Pattern matching

- Datatypes and case classes support pattern matching
  - We have seen a simple form of pattern matching for sum types.
  - This generalizes to named variants
  - But still is very limited: we only consider one “level” at a time
- Patterns typically also include constants and pairs/records

```scala
x match { case (1, (true, "abcd")) => ...}
```

- Patterns in Scala, Haskell, ML can also be nested: that is, they can match more than one constructor

```scala
x match { case Cons(1,Cons(y,Nil())) => ...}
```

More pattern matching

- Variables cannot be repeated, instead, explicit equality tests need to be used.
- The special pattern _ matches anything
- Patterns can overlap, and usually they are tried in order

```scala
result match {
  case OK => println("All is well")
  case _ => println("Release the hounds!")
}
```

// not the same as

```scala
result match {
  case _ => println("Release the hounds!")
  case OK => println("All is well")
}
```
Expanding nested pattern matching

- Nested pattern matching can be expanded out:

```scala
l match {
  case Cons(x,Cons(y,Nil())) => ... 
}
```

expands to

```scala
l match {
  case Cons(x,t1) => t1 match {
    case Cons(y,t2) => t2 match {
      case Nil() => ...
    }
  }
}
```

Type abbreviations and definitions

- Obviously, it quickly becomes painful to write "\( \langle x : \text{int}, y : \text{str} \rangle \)" over and over.
- **Type abbreviations** introduce a name for a type.
  
  ```scala
  type T = \( \tau \)
  ```

  An abbreviation name \( T \) treated the same as its expansion \( \tau \)
  - (much like let-bound variables)

- Examples:
  ```scala
  type Point = \( \langle x : \text{dbl}, y : \text{dbl} \rangle \)
  type Point3d = \( \langle x : \text{dbl}, y : \text{dbl}, z : \text{dbl} \rangle \)
  type Color = \( \langle r : \text{int}, g : \text{int}, b : \text{int} \rangle \)
  type ColoredPoint = \( \langle x : \text{dbl}, y : \text{dbl}, c : \text{Color} \rangle \)
  ```

Type definitions

- Instead, can also consider defining new (named) types
  ```scala
  deftype T = \( \tau \)
  ```

- The term **generative** is sometimes used to refer to definitions that create a new entity rather than introducing an abbreviation

- Type abbreviations are usually not allowed to be recursive; type definitions can be.
  ```scala
  deftype IntList = \[ Nil : \text{unit}, Cons : \text{int} \times \text{IntList} \]
  ```

Type definitions vs. abbreviations in practice

- In Haskell, type abbreviations are introduced by type, while new types can be defined by data or newtype declarations.
- In Java, there is no explicit notation for type abbreviations; the only way to define a new type is to define a class or interface
- In Scala, type abbreviations are introduced by type, while the class, object and trait constructs define new types
Subtyping

- Suppose we have a function:
  \[ dist = \lambda p: Point. \sqrt{(p.x)^2 + (p.y)^2} \]
  for computing the distance to the origin.
- Only the x and y fields are needed for this, so we'd like to be able to use this on ColoredPoint also.
- But, this doesn't typecheck:
  \[ dist(\langle x=8.0, y=12.0, c=purple \rangle) = 13.0 \]
- We can introduce a subtyping relationship between Point and ColoredPoint to allow for this.

Liskov Substitution Principle

If S is a subtype of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of the program.

- If we use \( \tau <: \tau' \) to mean "\( \tau \) is a subtype of \( \tau' \)" and consider well-typedness to be desirable, then we can translate this to the following subsumption rule:
  \[
  \Gamma \vdash e : \tau_1 \\
  \tau_1 <: \tau_2 \\
  \Gamma \vdash e : \tau_2
  \]
  This says: if \( e \) has type \( \tau_1 \) and \( \tau_1 <: \tau_2 \), then we can proceed by pretending it has type \( \tau_2 \).

Record subtyping: width and depth

- There are several different ways to define subtyping for records.
  - **Width subtyping**: subtype has same fields as supertype (with identical types), and may have additional fields at the end:
    \[
    \langle l_1 : \tau_1, \ldots, l_n : \tau_n, l_{n+k} : \tau_{n+k} \rangle <: \langle l_1 : \tau'_1, \ldots, l_n : \tau'_n \rangle
    \]
  - **Depth subtyping**: subtype's fields are pointwise subtypes of supertype
    \[
    \tau_1 <: \tau'_1 \quad \ldots \quad \tau_n <: \tau'_n \\
    \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle <: \langle l_1 : \tau'_1, \ldots, l_n : \tau'_n \rangle
    \]
  - These rules can be combined. Optionally, field reordering can also be allowed (but is harder to implement).

Examples

(We'll abbreviate \( P = Point \), \( P3 = Point3d \), \( CP = ColoredPoint \) to save space...)

- So we have:
  \[ P3d = \langle x: db1, y: db1, z: db1 \rangle <: \langle x: db1, y: db1 \rangle = P \]

  \[ CP = \langle x: db1, y: db1, c: Color \rangle <: \langle x: db1, y: db1 \rangle = P \]

  but no other subtyping relationships hold

- So, we can call \( dist \) on \( Point3d \) or \( ColoredPoint \):

  \[ \begin{align*}
  x : P3d & \vdash x : P3d <: P \\
  x : P3d & \vdash x : P \\
  x : P3d & \vdash dist : P \rightarrow db1 \\
  x : P3d & \vdash dist(x) : db1
  \end{align*} \]
**Subtyping for pairs and variants**

- For pairs, subtyping is componentwise
  
  \[
  \tau_1 <: \tau'_1 \quad \tau_2 <: \tau'_2 \\
  \frac{}{\tau_1 \times \tau_2 <: \tau'_1 \times \tau'_2}
  \]

- Similarly for binary variants
  
  \[
  \tau_1 <: \tau'_1 \quad \tau_2 <: \tau'_2 \\
  \frac{}{\tau_1 + \tau_2 <: \tau'_1 + \tau'_2}
  \]

- For named variants, can have additional subtyping rules (but this is rare)

**Subtyping for functions**

- When is \(A_1 \rightarrow B_1 <: A_2 \rightarrow B_2?\)
- Maybe componentwise, like pairs?
  
  \[
  \tau_1 <: \tau'_1 \quad \tau_2 <: \tau'_2 \\
  \frac{}{\tau_1 \rightarrow \tau_2 <: \tau'_1 \rightarrow \tau'_2}
  \]

- But then we can do this (where \(\Gamma(p) = P)\):
  
  \[
  \tau_1 <: \tau'_1 \quad \tau_2 <: \tau'_2 \\
  \frac{}{\tau_1 \rightarrow \tau_2 <: \tau'_1 \rightarrow \tau'_2}
  \]

- So, once \(ColoredPoint\) is a subtype of \(Point\), we can change any \(Point\) to a \(ColoredPoint\) also. That doesn’t seem right.

**Covariant vs. contravariant**

- For the result type of a function (and for pairs and other data structures), the direction of subtyping is preserved:

  \[
  \frac{}{\tau_1 \rightarrow \tau_2 <: \tau'_1 \rightarrow \tau'_2}
  \]

- Subtyping of function results, pairs, etc., where order is preserved, is **covariant**.

- For the *argument* type of a function, the direction of subtyping is flipped:

  \[
  \frac{}{\tau'_1 <: \tau_1} \Rightarrow \frac{}{\tau_1 \rightarrow \tau_2 <: \tau'_1 \rightarrow \tau'_2}
  \]

- Subtyping of function arguments, where order is reversed, is called **contravariant**.

**The “top” and “bottom” types**

- any: a type that is a supertype of all types.
  
  - Such a type describes the common interface of all its subtypes (e.g. hashing, equality in Java)
  - In Scala, this is called Any

- empty: a type that is a subtype of all types.
  
  - Usually, such a type is considered to be empty: there cannot actually be any values of this type.
  - We’ve actually encountered this before, as the degenerate case of a choice type where there are zero choices
  - In Scala, this type is called Nothing. So for any Scala type \(\tau\) we have \(Nothing <: \tau <: Any\).
Notice that we combine the covariant and contravariant rules for functions into a single rule.

The approach to subtyping considered so far is called structural.

The names we use for type abbreviations don’t matter, only their structure. For example, \( \text{Point3d} <: \text{Point} \) because \( \text{Point3d} \) has all of the fields of \( \text{Point} \) (and more).

Then \( \text{dist}(p) \) also runs on \( p : \text{Point3d} \) (and gives a nonsense answer!)

So far, a defined type has no subtypes (other than itself).

By default, definitions \( \text{ColoredPoint}, \text{Point} \) and \( \text{Point3d} \) are unrelated.

If we defined new types \( \text{Point'} \) and \( \text{Point3d'} \), rather than treating them as abbreviations, then we have more control over subtyping.

Then we can declare \( \text{ColoredPoint'} \) to be a subtype of \( \text{Point'} \)

\[
\text{deftype} \ \text{Point'} = \langle x : \text{dbl}, y : \text{dbl} \rangle \\
\text{deftype} \ \text{ColoredPoint'} <: \text{Point'} = \langle x : \text{dbl}, y : \text{dbl}, c : \text{Color} \rangle
\]

However, we could choose not to assert \( \text{Point3d'} \) to be a subtype of \( \text{Point'} \), preventing (mis)use of subtyping to view \( \text{Point3d's} \) as \( \text{Point's} \).

This nominal subtyping is used in Java and Scala

- A defined type can only be a subtype of another if it is declared as such
- More on this later!