Overview

- Last time:
  - Polymorphism: abstracting over types
- Today we explore another form of abstraction
  - Records (generalizing pairs), variants (generalizing sums) and pattern matching
  - Subtyping

Records

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

\[
\begin{align*}
e & ::= \cdots \mid \langle l_1 = e_1, \ldots, l_n = e_n \rangle \mid e.1 \\
v & ::= \cdots \mid \langle l_1 = v_1, \ldots, l_n = v_n \rangle \\
\tau & ::= \cdots \mid \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle
\end{align*}
\]

- Examples:
  - \( \langle \text{fst}=1, \text{snd}="\text{forty-two}" \rangle. \text{snd} \mapsto "\text{forty-two}" \)
  - \( \langle x=3.0, y=4.0, \text{length}=5.0 \rangle \)

- Record fields can be (first-class) functions too:
  - \( \langle x=3.0, y=4.0, \text{length}=\lambda(x, y). \sqrt{x \times x + y \times y} \rangle \)

Named variants

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

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

- 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 Int | 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 and earlier designs such as HOPE (Burstall).

Type abbreviations

- Obviously, it quickly becomes painful to write "\(x : \text{int}, y : \text{str}\)" 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:dbl,y:dbl\rangle
  type Point3d = \langle x:dbl,y:dbl,z:dbl\rangle
  type Color = \langle r:int,g:int,b:int\rangle
  type ColoredPoint = \langle x:dbl,y:dbl,c: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 : unit, Cons : int \times IntList]
  ```
Patterns 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
  
  ```
  x match { case (1, (true, "abcd")) => ... }
  ```

- Patterns in Scala, Haskell, ML can also be nested: that is, they can match more than one constructor
  
  ```
  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
result match {
  case _ => println("Release the hounds!")
  case OK => println("All is well")
}

Expanding nested pattern matching

- Nested pattern matching can be expanded out:
  
  ```
  x match {
    case Cons(1,Cons(y,Nil())) => ...
  }
  ```

  expands to
  
  ```
  x match {
    case Cons(h,t) => y match {
      case 1 => z match {
        case Cons(y,t2) => t2 match {
          case Nil() => ...
        }
      }
    }
  }
  ```
Suppose we have a function:

\[ \text{dist} = \lambda x: \text{Point}. \, \text{sqrt}((p.x)^2 + (p.y)^2) \]

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 ColoredPoints also.
- But, this doesn't typecheck:

\[ \text{dist}(\langle x=8.0, y=12.0, c=\text{purple} \rangle) = 13.0 \]

We can introduce a subtyping relationship between Point and ColoredPoint to allow for this.

Record subtyping: width and depth

- There are several different ways to define subtyping for records.
- **Width subtyping**: subtype has all fields of supertype (with identical types)

\[ \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle <: \langle l_1' : \tau_1', \ldots, l_n' : \tau_n' \rangle \]

- **Depth subtyping**: subtype's fields are pointwise subtypes of supertype

\[ \tau_1 <: \tau_1' \quad \cdots \quad \tau_n <: \tau_n' \]

\[ \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle <: \langle l_1 : \tau_1', \ldots, l_n : \tau_n' \rangle \]

These rules can be combined. Optionally, field reordering can also be allowed (but is harder to implement).

Liskov proposed a guideline for subtyping:

**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:

\[ \frac{}{\Gamma \vdash e : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2} \]

This says: if \( e \) has type \( \tau_1 \) and \( \tau_1 <: \tau_2 \), then we can proceed by pretending it has type \( \tau_2 \).

Examples

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

- So we have:

\[ P3d = \langle x: \text{dbl}, y: \text{dbl}, z: \text{dbl} \rangle <: \langle x: \text{dbl}, y: \text{dbl} \rangle = P \]

\[ CP = \langle x: \text{dbl}, y: \text{dbl}, c: \text{Color} \rangle <: \langle x: \text{dbl}, y: \text{dbl} \rangle = P \]

but no other subtyping relationships hold
- So, we can call \( \text{dist} \) on Point3d or ColoredPoint:

\[ x : P3d \vdash \text{dist} : P3d \rightarrow \text{dbl} \]

\[ x : P3d \vdash \text{dist}(x) : \text{dbl} \]
Subtyping for pairs and variants

- For pairs, subtyping is componentwise

\[
\tau_1 <: \tau'_1 \quad \tau_2 <: \tau'_2 \\
\tau_1 \times \tau_2 <: \tau'_1 \times \tau'_2
\]

- Similarly for binary variants

\[
\tau_1 <: \tau'_1 \quad \tau_2 <: \tau'_2 \\
\tau_1 + \tau_2 <: \tau'_1 + \tau'_2
\]

- 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?

\[
\tau_1 <: \tau'_1 \quad \tau_2 <: \tau'_2 \\
\tau_1 \rightarrow \tau_2 <: \tau'_1 \rightarrow \tau'_2
\]

- But then we can do this (where \( \Gamma(p) = P \)):

\[
\begin{align*}
\Gamma \vdash \lambda x : CP \rightarrow CP \\
CP <: P \quad CP <: CP \\
\Gamma \vdash \lambda x : CP \rightarrow CP \\
P : CP \\
\Gamma \vdash \lambda (\lambda x : CP) p : CP
\end{align*}
\]

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

Covariant vs. contravariant

- We say that subtyping is \textit{covariant} for the result type of a function (and for pairs and other data structures)

\[
\begin{align*}
\tau_2 \lln \tau'_2 \\
\tau_1 \rightarrow \tau_2 \lln \tau_1 \rightarrow \tau'_2
\end{align*}
\]

- For the \textit{argument} type of a function, we say that subtyping is \textit{contravariant}

\[
\begin{align*}
\tau'_1 \lln \tau_1 \\
\tau_1 \rightarrow \tau_2 \lln \tau'_1 \rightarrow \tau_2
\end{align*}
\]

The “top” and “bottom” types

- \textit{any}: a type that is a supertype of all types.

\[
\begin{align*}
\text{Such a type describes the common interface of all its subtypes (e.g. hashing, equality in Java)} \\
\text{In Scala, this is called \textit{Any}}
\end{align*}
\]

- \textit{empty}: a type that is a subtype of all types.

\[
\begin{align*}
\text{Usually, such a type is considered to be \textit{empty}: there cannot actually be any values of this type.} \\
\text{We’ve actually encountered this before, as the degenerate case of a choice type where there are zero choices} \\
\text{In Scala, this type is called \textit{Nothing}. So for any Scala type \( \tau \) we have \textit{Nothing} \lln \tau \lln \textit{Any}.}
\end{align*}
\]
Summary: Subtyping rules

\[
egin{align*}
\tau_1 &<: \tau_2 \\
\emptyset &<: \tau \\
\tau &<: \text{any} \\
\tau &<: \tau \\
\tau_1 &<: \tau_2 \\
\tau_1 \times \tau_2 &<: \tau_1 \times \tau_2 \\
\tau_1 + \tau_2 &<: \tau_1 + \tau_2 \\
\tau_1' &<: \tau_1 \\
\tau_2 &<: \tau_2' \\
\tau_1 \rightarrow \tau_2 &<: \tau_1' \rightarrow \tau_2'
\end{align*}
\]

Structural vs. Nominal subtyping

- 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.

Structural vs. Nominal subtyping

- 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!

Summary

- Today we covered:
  - Records, variants, and pattern matching
  - Type abbreviations and definitions
  - Subtyping
- Next time:
  - Imperative programming and control structures