Elements of Programming Languages

Lecture 7: Records, variants, and subtyping

James Cheney

University of Edinburgh

October 14, 2024

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.

$$\begin{array}{rcl} e & ::= & \cdots \mid \langle l_1 = e_1, \ldots, l_n = e_n \rangle \mid e.l \\ v & ::= & \cdots \mid \langle l_1 = v_1, \ldots, l_n = v_n \rangle \\ \tau & ::= & \cdots \mid \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle \end{array}$$

- Examples:

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

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

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

Named variants

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

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

- 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

```
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*

```
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

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

- and cases can define named fields:

```
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

```
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

```
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:

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

expands to

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

Type abbreviations

- Obviously, it quickly becomes painful to write "$\langle x : \texttt{int}, y : \texttt{str}\rangle$" over and over.
- **Type abbreviations** introduce a name for a type.

$$\texttt{type}\ T = \tau$$

  An abbreviation name T treated the same as its expansion τ
  - (much like `let`-bound variables)
- Examples:

$$\begin{array}{l}\texttt{type}\ \mathit{Point} = \langle x{:}\texttt{dbl}, y{:}\texttt{dbl}\rangle \\ \texttt{type}\ \mathit{Point3d} = \langle x{:}\texttt{dbl}, y{:}\texttt{dbl}, z{:}\texttt{dbl}\rangle \\ \texttt{type}\ \mathit{Color} = \langle r{:}\texttt{int}, g{:}\texttt{int}, b{:}\texttt{int}\rangle \\ \texttt{type}\ \mathit{ColoredPoint} = \langle x{:}\texttt{dbl}, y{:}\texttt{dbl}, c{:}\mathit{Color}\rangle\end{array}$$

Type definitions

- Instead, can also consider *defining new (named) types*

$$\texttt{deftype}\ T = \tau$$

- 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; recursive type definitions are often allowed.

$$\texttt{deftype}\ \mathit{IntList} = [\mathit{Nil} : \texttt{unit}, \mathit{Cons} : \texttt{int} \times \mathit{IntList}]$$

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*s also.
- But, this doesn't typecheck (even though it would evaluate correctly):

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

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

Subtyping

- Liskov (Turing award 2008) 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 "τ is a subtype of τ'", 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 τ_1 and $\tau_1 <: \tau_2$, then we can proceed by pretending it has type τ_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:

$$\frac{}{\langle l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots, l_{n+k} : \tau_{n+k} \rangle <: \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle}$$

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

$$\frac{\tau_1 <: \tau_1' \quad \cdots \quad \tau_n <: \tau_n'}{\langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle <: \langle l_1 : \tau_1', \ldots, l_n : \tau_n' \rangle}$$

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

Examples

- (We'll abbreviate $P = Point$, $P3d = Point3d$, $CP = ColoredPoint$ to save space...)
- So we have:

$$P3d = \langle x{:}\mathtt{dbl}, y{:}\mathtt{dbl}, z{:}\mathtt{dbl}\rangle <: \langle x{:}\mathtt{dbl}, y{:}\mathtt{dbl}\rangle = P$$

$$CP = \langle x{:}\mathtt{dbl}, y{:}\mathtt{dbl}, c{:}Color\rangle <: \langle x{:}\mathtt{dbl}, y{:}\mathtt{dbl}\rangle = P$$

but no other subtyping relationships hold

- So, we can call *dist* on *Point3d* or *ColoredPoint*:

$$\dfrac{\dfrac{\vdots}{x : P3d \vdash dist : P \to \mathtt{dbl}} \quad \dfrac{x : P3d \vdash x : P3d \quad P3d <: P}{x : P3d \vdash x : P}}{x : P3d \vdash dist(x) : \mathtt{dbl}}$$

Subtyping for pairs and variants

- For pairs, subtyping is componentwise

$$\frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2'}{\tau_1 \times \tau_2 <: \tau_1' \times \tau_2'}$$

- Similarly for binary variants

$$\frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2'}{\tau_1 + \tau_2 <: \tau_1' + \tau_2'}$$

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

Subtyping for functions

- When is $A_1 \to B_1 <: A_2 \to B_2$?
- Maybe componentwise, like pairs?

$$\frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}$$

Subtyping for functions

- When is $A_1 \to B_1 <: A_2 \to B_2$?
- Maybe componentwise, like pairs?

$$\frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}$$

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

$$\frac{\dfrac{\Gamma \vdash \lambda x.x : CP \to CP \quad \dfrac{CP <: P \quad CP <: CP}{CP \to CP <: P \to CP}}{\Gamma \vdash \lambda x.x : P \to CP} \quad \Gamma \vdash p : P}{\Gamma \vdash (\lambda x.x)p : CP}$$

Subtyping for functions

- When is $A_1 \to B_1 <: A_2 \to B_2$?
- Maybe componentwise, like pairs?

$$\frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}$$

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

$$\frac{\dfrac{\Gamma \vdash \lambda x.x : CP \to CP \quad \dfrac{CP <: P \quad CP <: CP}{CP \to CP <: P \to CP}}{\Gamma \vdash \lambda x.x : P \to CP} \quad \Gamma \vdash p : P}{\Gamma \vdash (\lambda x.x)p : CP}$$

- 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:

$$\frac{\tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1 \to \tau_2'}$$

Covariant vs. contravariant

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

$$\frac{\tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1 \to \tau_2'}$$

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

Covariant vs. contravariant

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

$$\frac{\tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1 \to \tau_2'}$$

- 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:

$$\frac{\tau_1' <: \tau_1}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2}$$

Covariant vs. contravariant

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

$$\frac{\tau_2 <: \tau_2'}{\tau_1 \rightarrow \tau_2 <: \tau_1 \rightarrow \tau_2'}$$

- 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:

$$\frac{\tau_1' <: \tau_1}{\tau_1 \rightarrow \tau_2 <: \tau_1' \rightarrow \tau_2}$$

- 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`

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 τ we have $Nothing <: \tau <: Any$.

Summary: Subtyping rules

$\tau_1 <: \tau_2$

$$\frac{}{\texttt{empty} <: \tau} \qquad \frac{}{\tau <: \texttt{any}} \qquad \frac{}{\tau <: \tau} \qquad \frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

$$\frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2'}{\tau_1 \times \tau_2 <: \tau_1' \times \tau_2'} \qquad \frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2'}{\tau_1 + \tau_2 <: \tau_1' + \tau_2'}$$

$$\frac{\tau_1' <: \tau_1 \quad \tau_2 <: \tau_2'}{\tau_1 \rightarrow \tau_2 <: \tau_1' \rightarrow \tau_2'}$$

Notice that we combine the covariant and contravariant rules for functions into a single rule.

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, $Point3d <: Point$ because $Point3d$ has all of the fields of $Point$ (and more).
- Then $dist(p)$ also runs on $p : Point3d$ (and gives a nonsense answer!)
- So far, a defined type has no subtypes (other than itself).
- By default, definitions $ColoredPoint$, $Point$ and $Point3d$ are unrelated.

Structural vs. Nominal subtyping

- If we defined new types *Point′* and *Point3d′*, rather than treating them as abbreviations, then we have more control over subtyping
- Then we can *declare* *ColoredPoint′* to be a subtype of *Point′*

$$\texttt{deftype}\ \mathit{Point}' = \langle x{:}\texttt{dbl}, y{:}\texttt{dbl}\rangle$$
$$\texttt{deftype}\ \mathit{ColoredPoint}' <: \mathit{Point}' = \langle x{:}\texttt{dbl}, y{:}\texttt{dbl}, c{:}\mathit{Color}\rangle$$

- However, we could choose not to assert *Point3d′* to be a subtype of *Point′*, preventing (mis)use of subtyping to view *Point3d′*s as *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:
  - Polymorphism and type inference