Elements of Programming Languages

Lecture 7: Records, variants, and subtyping

James Cheney

University of Edinburgh

October 19, 2023
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.

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

- 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:

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

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

\[
e ::= \cdots \mid C_i(e) \mid \text{case } e \text{ of } \{ C_1(x) \Rightarrow e_1; \ldots \}
\]

\[
v ::= \cdots \mid C_i(v)
\]

\[
\tau ::= \cdots \mid [C_1 : \tau_1, \ldots, C_n : \tau_n]
\]

- 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
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() => ...
    }
  }
}
```
Records, Variants, and Pattern Matching

Type abbreviations and definitions
Subtyping

Type abbreviations

- Obviously, it quickly becomes painful to write "⟨x : int, y : str⟩" over and over.
- **Type abbreviations** introduce a name for a type.

\[
\text{type } T = \tau
\]

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

- Examples:

\[
\begin{align*}
\text{type } Point & = \langle x : \text{dbl}, y : \text{dbl} \rangle \\
\text{type } Point3d & = \langle x : \text{dbl}, y : \text{dbl}, z : \text{dbl} \rangle \\
\text{type } Color & = \langle r : \text{int}, g : \text{int}, b : \text{int} \rangle \\
\text{type } ColoredPoint & = \langle x : \text{dbl}, y : \text{dbl}, c : Color \rangle
\end{align*}
\]
Type definitions

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

  \[
  \text{deftype } T = \tau
  \]

- 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.

  \[
  \text{deftype } \text{IntList} = [\text{Nil} : \text{unit}, \text{Cons} : \text{int} \times \text{IntList}]
  \]
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:

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

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 \( \text{ColoredPoint} \)s also.

But, this doesn’t typecheck (even though it would evaluate correctly):

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

We can introduce a \textit{subtyping} relationship between \( \text{Point} \) and \( \text{ColoredPoint} \) to allow for this.
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 “$\tau$ is a subtype of $\tau'$”, and consider well-typedness to be desirable, then we can translate this to the following subsumption rule:

$$
\Gamma \vdash e : \tau_1 \quad \tau_1 <: \tau_2 \\
\hline
\Gamma \vdash e : \tau_2
$$

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:

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

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

\[
\tau_1 <: \tau'_1 \quad \cdots \quad \tau_n <: \tau'_n
\]

\[
\langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle <: \langle l_1 : \tau'_1, \ldots, l_n : \tau'_n \rangle
\]

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

(We'll abbreviate $P = \text{Point}$, $P3d = \text{Point3d}$, $CP = \text{ColoredPoint}$ 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 \textit{dist} on \texttt{Point3d} or \texttt{ColoredPoint}:

\[
\begin{align*}
  & x : P3d \vdash \text{dist} : P \rightarrow \text{dbl} & x : P3d \vdash x : P3d & P3d <: P \\
\end{align*}
\]

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

- For pairs, subtyping is componentwise

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

- Similarly for binary variants

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

- 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?

\[
\frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2'}{
\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}
\]

So, once ColoredPoint is a subtype of Point, we can change any Point to a ColoredPoint also. That doesn't seem right.
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 \( \Gamma(p) = P \)):

\[
\Gamma \vdash \lambda x.x : CP \to CP \quad \frac{CP <: P \quad CP <: CP}{CP \to CP <: P \to CP}
\]

\[
\Gamma \vdash \lambda x.x : P \to CP \quad \Gamma \vdash p : P \quad \Gamma \vdash (\lambda x.x)p : CP
\]

So, once ColoredPoint is a subtype of Point, we can change any Point to a ColoredPoint also. That doesn't seem right.
Subtyping for functions

- When is \( A_1 \rightarrow B_1 <: A_2 \rightarrow B_2 \)?
- Maybe componentwise, like pairs?

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

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

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

- 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

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

\[
\frac{\tau_2 <: \tau'_2}{\tau_1 \rightarrow \tau_2 <: \tau_1 \rightarrow \tau'_2}
\]
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.
Covariant vs. contravariant

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

\[
\begin{align*}
\tau_2 & \ll \tau_2' \\
\tau_1 \rightarrow \tau_2 & \ll \tau_1 \rightarrow \tau_2'
\end{align*}
\]

- 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:

\[
\begin{align*}
\tau_1' & \ll \tau_1 \\
\tau_1 \rightarrow \tau_2 & \ll \tau_1' \rightarrow \tau_2
\end{align*}
\]
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:

\[
\frac{\tau_1' <: \tau_1}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2}
\]

- 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 \( \tau \) we have `Nothing <: \tau <: Any`. 
**Summary: Subtyping rules**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty &lt;: ( \tau )</td>
<td></td>
</tr>
<tr>
<td>( \tau ) &lt;: any</td>
<td></td>
</tr>
<tr>
<td>( \tau ) &lt;: ( \tau )</td>
<td></td>
</tr>
<tr>
<td>( \tau_1 ) &lt;: ( \tau'_1 ) ( \tau_2 ) &lt;: ( \tau'_2 )</td>
<td>( \tau_1 \times \tau_2 ) &lt;: ( \tau'_1 \times \tau'_2 )</td>
</tr>
<tr>
<td>( \tau_1 ) + ( \tau_2 ) &lt;: ( \tau'_1 + \tau'_2 )</td>
<td></td>
</tr>
<tr>
<td>( \tau'_1 ) &lt;: ( \tau_1 ) ( \tau'_2 ) &lt;: ( \tau_2 )</td>
<td>( \tau'_1 \rightarrow \tau'_2 ) ( \tau'_1 \rightarrow \tau'_2 )</td>
</tr>
</tbody>
</table>

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, $\text{Point3d} <: \text{Point}$ 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 \( \text{Point}' \).
  
  \[
  \text{deftype Point}' = \langle x: \text{dbl}, y: \text{dbl} \rangle \\
  \text{deftype ColoredPoint}' <: \text{Point}' = \langle x: \text{dbl}, y: \text{dbl}, c: \text{Color} \rangle
  \]

- 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:
  - Polymorphism and type inference