The story so far

- We’ve now covered the main ingredients of any programming language:
  - Abstract syntax
  - Semantics/interpretation
  - Types
  - Variables and binding
  - Functions and recursion
- but only in the context of a very weak language: there are no “data structures” (records, lists, variants), pointers, side-effects etc.
- Let alone even more advanced features such as classes, interfaces, or generics
- Over the next few lectures we will show how to add them, consolidating understanding of the foundations along the way.

Pairs

- The simplest way to combine data structures: pairing
  $$(1, 2) \quad (\text{true, false}) \quad (1, (\text{true, } \lambda x: \text{int}. x + 2))$$
- If we have a pair, we can extract one of the components:
  $$\text{fst } (1, 2) \rightarrow 1 \quad \text{snd } (\text{true, false}) \rightarrow \text{false}$$
  $$\text{snd } (1, (\text{true, } \lambda x: \text{int}. x + 2)) \rightarrow (\text{true, } \lambda x: \text{int}. x + 2)$$
- Finally, we can often pattern match against a pair, to extract both components at once:
  $$\text{let } (x, y) = (1, 2) \text{ in } (y, x) \rightarrow (2, 1)$$

Pairs in various languages

<table>
<thead>
<tr>
<th></th>
<th>Haskell</th>
<th>Scala</th>
<th>Java</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 2)</td>
<td>(1, 2)</td>
<td>new Pair(1, 2)</td>
<td>(1, 2)</td>
<td></td>
</tr>
<tr>
<td>fst e</td>
<td>e.1</td>
<td>e.getFirst()</td>
<td>e[0]</td>
<td></td>
</tr>
<tr>
<td>snd e</td>
<td>e.2</td>
<td>e.getSecond()</td>
<td>e[1]</td>
<td></td>
</tr>
<tr>
<td>let (x, y) = (1, 2) in (y, x)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

- Functional languages typically have explicit syntax (and types) for pairs
- Java and C-like languages have “record”, “struct” or “class” structures that accommodate multiple, named fields.
  - A pair type can be defined but is not built-in and there is no support for pattern-matching
Syntax and Semantics of Pairs

- Syntax of pair expressions and values:

  \[ e ::= \cdots | (e_1, e_2) | \text{fst } e | \text{snd } e \]
  \[ \triangledown e \triangledown v \]

  (let pair \((x, y) = e_1 \) in \(e_2\) \(\triangledown v\)

Types for Pairs

- Types for pair expressions:

  \[ \tau ::= \cdots | \tau_1 \times \tau_2 \]

Let vs. fst and snd

- The \text{fst} and \text{snd} operations are definable in terms of let pair:

  \[
  \text{fst } e \iff \text{let pair } (x, y) = e_1 \text{ in } x \\
  \text{snd } e \iff \text{let pair } (x, y) = e_1 \text{ in } y
  \]

- Actually, the let pair construct is definable in terms of let, \text{fst}, \text{snd} too:

  \[
  \text{let pair } (x, y) = e_1 \text{ in } e_2 \\
  \iff \text{let } p = e_1 \text{ in } e_2[\text{fst } p/x, \text{snd } p/y]
  \]

- We typically just use the (simpler) \text{fst} and \text{snd} constructs and treat let pair as syntactic sugar.

More generally: tuples and records

- Nothing stops us from adding triples, quadruples, \ldots, \(n\)-tuples.

  \[
  (1, 2, 3) \quad (\text{true}, 2, 3, \lambda x. (x, x))
  \]

- As mentioned earlier, many languages prefer named record syntax:

  \[
  (a : 1, b : 2, c : 3) \quad (b : \text{true}, n_1 : 2, n_2 : 3, f : \lambda x. (x, x))
  \]

  (cf. class fields in Java, structs in C, etc.)

- These are undeniably useful, but are definable using pairs.

- We’ll revisit named record-style constructs when we consider classes and modules.
**Special case: the “unit” type**

- Nothing stops us from adding a type of $0$-tuples: a data structure with no data. This is often called the *unit type*, or unit.

```
\[ e ::= \cdots | () \]
```

```
\[ v ::= \cdots | () \]
```

```
\[ \tau ::= \cdots | \text{unit} \]
```

\[ (\gamma) \Downarrow (\gamma) \quad \Gamma \vdash (): \text{unit} \]

- This may seem a little pointless: why bother to define a type with no (interesting) data and no operations?
- This is analogous to `void` in C/Java; in Haskell and Scala it is called `()`.

**Motivation for variant types**

- Pairs allow us to combine two data structures ($\tau_1$ and a $\tau_2$).
- What if we want a data structure that allows us to *choose* between different options?
- We’ve already seen one example: booleans.
  - A boolean can be one of two values.
  - Given a boolean, we can look at its value and choose among two options, using `if` `then` `else`.
- Can we generalize this idea?

**Another example: null values**

- Sometimes we want to produce *either* a regular value *or* a special "null" value.
- Some languages, including SQL and Java, allow many types to have null values by default.
  - This leads to the need for defensive programming to avoid the dreaded `NullPointerException` in Java, or strange query behavior in SQL
  - Sir Tony Hoare (inventor of Quicksort) introduced null references in Algol in 1965 “simply because it was so easy to implement”!
  - He now calls them “the billion dollar mistake”:
What would be better?

- Consider an option type:
  
  \[
  e ::= \cdots | \text{none} | \text{some}(e)
  \]

  \[
  \tau ::= \cdots | \text{option}[^\tau]
  \]

  \[
  \Gamma \vdash \text{none} : \text{option}[\tau]
  \]

  \[
  \Gamma \vdash \text{some}(e) : \text{option}[\tau]
  \]

- Then we can use none to indicate absence of a value, and some(e) to give the present value.

- Moreover, the type of an expression tells us whether null values are possible.

Error codes

- The option type is useful but still a little limited: we either get a \(\tau\) value, or nothing
- If none means failure, we might want to get some more information about why the failure occurred.
- We would like to be able to return an error code
  
  - In older languages, notably C, special values are often used for errors
  - Example: read reads from a file, and either returns number of bytes read, or -1 representing an error
  - The actual error code is passed via a global variable
  - It’s easy to forget to check this result, and the function’s return value can’t be used to return data.
  - Other languages use exceptions, which we’ll cover much later

The OK-or-error type

- Suppose we want to return either a normal value \(\tau_{\text{ok}}\) or an error value \(\tau_{\text{err}}\).
- Let’s write \(\text{okOrErr}[\tau_{\text{ok}}, \tau_{\text{err}}]\) for this type.

  \[
  e ::= \cdots | \text{ok}(e) | \text{err}(e)
  \]

  \[
  \tau ::= \cdots | \text{okOrErr}[\tau_1, \tau_2]
  \]

- Basic idea:
  - if \(e\) has type \(\tau_{\text{ok}}\), then \(\text{ok}(e)\) has type \(\text{okOrErr}[\tau_{\text{ok}}, \tau_{\text{err}}]\)
  - if \(e\) has type \(\tau_{\text{err}}\), then \(\text{err}(e)\) has type \(\text{okOrErr}[\tau_{\text{ok}}, \tau_{\text{err}}]\)

- How do we use \(\text{okOrErr}[\tau_{\text{ok}}, \tau_{\text{err}}]\)?

  - When we talked about \(\text{option}[\tau]\), we didn’t really say how to use the results.
  - If we have a \(\text{okOrErr}[\tau_{\text{ok}}, \tau_{\text{err}}]\) value \(v\), then we want to be able to branch on its value:
    - If \(v\) is \(\text{ok}(v_{\text{ok}})\), then we probably want to get at \(v_{\text{ok}}\) and use it to proceed with the computation
    - If \(v\) is \(\text{err}(v_{\text{err}})\), then we probably want to get at \(v_{\text{err}}\) to report the error and stop the computation.
  - In other words, we want to perform case analysis on the value, and extract the wrapped value for further processing
Case analysis

- We consider a case analysis construct as follows:

  \[
  \text{case } e \text{ of } \{ \text{ok}(x) \Rightarrow e_{\text{ok}} ; \text{err}(y) \Rightarrow e_{\text{err}} \}
  \]

- This is a generalized conditional: “If \( e \) evaluates to \( \text{ok}(v_{\text{ok}}) \), then evaluate \( e_{\text{ok}} \) with \( v_{\text{ok}} \) replacing \( x \), else it evaluates to \( \text{err}(v_{\text{err}}) \) so evaluate \( e_{\text{err}} \) with \( v_{\text{err}} \) replacing \( y \).”

- Here, \( x \) is bound in \( e_{\text{ok}} \) and \( y \) is bound in \( e_{\text{err}} \)

- This construct should be familiar by now from Scala:

\[
\begin{align*}
\text{e match } \{ & \text{case Ok}(x) \Rightarrow e_1 \\
& \text{case Err}(x) \Rightarrow e_2 \\
& \text{\} // note slightly different syntax}
\end{align*}
\]

Types for variants

- We extend the typing rules as follows:

\[
\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \text{left}(e) : \tau_1 + \tau_2} \quad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \text{right}(e) : \tau_1 + \tau_2}
\]

\[
\frac{\Gamma \vdash e : \tau_1 + \tau_2 \\
\Gamma, x : \tau_1 \vdash e_1 : \tau \\
\Gamma, y : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \text{case } e \text{ of } \{ \text{left}(x) \Rightarrow e_1 ; \text{right}(y) \Rightarrow e_2 \} : \tau}
\]

- Idea: \( \text{left} \) and \( \text{right} \) “wrap” \( \tau_1 \) or \( \tau_2 \) as \( \tau_1 + \tau_2 \)

- Idea: Case is like conditional, only we can use the wrapped value extracted from \( \text{left}(v) \) or \( \text{right}(v) \).

Variant types, more generally

- Notice that the \( \text{ok} \) and \( \text{err} \) cases are completely symmetric.

- Generalizing this type might also be useful for other situations than error handling...

- Therefore, let’s rename and generalize the notation:

\[
\begin{align*}
e & ::= \cdots | \text{left}(e) | \text{right}(e) \\
& \quad | \text{case } e \text{ of } \{ \text{left}(x) \Rightarrow e_1 ; \text{right}(y) \Rightarrow e_2 \}
\end{align*}
\]

\[
\begin{align*}
v & ::= \cdots | \text{left}(v) | \text{right}(v) \\
\tau & ::= \cdots | \tau_1 + \tau_2
\end{align*}
\]

- We will call type \( \tau_1 + \tau_2 \) a variant type (sometimes also called sum or disjoint union)

Semantics of variants

- We extend the evaluation rules as follows:

\[
\frac{e \Downarrow v}{e \Downarrow v}
\]

\[
\begin{align*}
\frac{e \Downarrow v}{\text{left}(e) \Downarrow \text{left}(v)} & \quad \frac{e \Downarrow v}{\text{right}(e) \Downarrow \text{right}(v)} \\
\frac{e_1[v_1/x] \Downarrow v}{\text{case } e \text{ of } \{ \text{left}(x) \Rightarrow e_1 ; \text{right}(y) \Rightarrow e_2 \} \Downarrow v} \\
\frac{e_2[v_2/y] \Downarrow v}{\text{case } e \text{ of } \{ \text{left}(x) \Rightarrow e_1 ; \text{right}(y) \Rightarrow e_2 \} \Downarrow v}
\end{align*}
\]

- Creating a \( \tau_1 + \tau_2 \) value is straightforward.

- Case analysis branches on the \( \tau_1 + \tau_2 \) value
Defining Booleans and option types

- The Boolean type bool can be defined as \( \text{unit} + \text{unit} \)
  
  \[
  \text{true} \iff \text{left}(()) \quad \text{false} \iff \text{right}()
  \]

- Conditional is then defined as case analysis, ignoring the variables
  
  \[
  \text{if e then } e_1 \text{ else } e_2 
  \iff \text{case } e \text{ of } \{ \text{left}(x) \Rightarrow e_1 \; ; \; \text{right}(y) \Rightarrow e_2 \}
  \]

- Likewise, the option type is definable as \( \tau + \text{unit} \):

  \[
  \text{some}(e) \iff \text{left}(e) \quad \text{none} \iff \text{right}()
  \]

Datatypes: named variants and case classes

- Programming directly with binary variants is awkward
- As for pairs, the \( \tau_1 + \tau_2 \) type can be generalized to \( n \)-ary choices or \textit{named variants}
- As we saw in Lecture 1 with abstract syntax trees, variants can be represented in different ways
  - Haskell supports “datatypes” which give constructor names to the cases
  - In Java, can use classes and inheritance to simulate this, verbosely (Python similar)
  - Scala does not directly support named variant types, but provides “case classes” and pattern matching
- We’ll revisit case classes and variants later in discussion of object-oriented programming.

The empty type

- We can also consider the 0-ary variant type

  \[
  \tau ::= \cdots | \text{empty}
  \]

  with \textit{no} associated expressions or values

- Scala provides Nothing as a built-in type; most languages do not
  - [Perhaps confusingly, this is not the same thing at all as the void or unit type!]

- We will talk about Nothing again when we cover \textit{subtyping}
  - (Insert \textit{Seinfeld} joke here, if anyone is old enough to remember that.)

Summary

- Today we’ve covered two primitive types for structured data:
  - Pairs, which combine two or more data structures
  - Variants, which represent alternative choices among data structures
  - Special cases (unit, empty) and generalizations (records, datatypes)

- This is a pattern we’ll see over and over:
  - Define a type and expressions for creating and using its elements
  - Define typing rules and evaluation rules

- Next time:
  - Named records and variants
  - Subtyping