Elements of Programming Languages

Lecture 6: Data structures

James Cheney

University of Edinburgh

October 11, 2016

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

$$(1, 2) \qquad (\mathtt{true}, \mathtt{false}) \qquad (1, (\mathtt{true}, \lambda x{:}\mathtt{int}.x + 2))$$

- If we have a pair, we can *extract* one of the components:

$$\mathtt{fst}\ (1, 2) \leadsto 1 \qquad \mathtt{snd}\ (\mathtt{true}, \mathtt{false}) \leadsto \mathtt{false}$$

$$\mathtt{snd}\ (1, (\mathtt{true}, \lambda x{:}\mathtt{int}.x + 2)) \leadsto (\mathtt{true}, \lambda x{:}\mathtt{int}.x + 2)$$

- Finally, we can often *pattern match* against a pair, to extract both components at once:

$$\mathtt{let\ pair}\ (x, y) = (1, 2)\ \mathtt{in}\ (y, x) \leadsto (2, 1)$$

Pairs in various languages

Haskell	Scala	Java	Python
`(1,2)`	`(1,2)`	`new Pair(1,2)`	`(1,2)`
`fst e`	`e._1`	`e.getFirst()`	`e[0]`
`snd e`	`e._2`	`e.getSecond()`	`e[1]`
`let (x,y) =`	`val (x,y) =`	N/A	N/A

- 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:

$$\begin{array}{rcl} e & ::= & \cdots \mid (e_1, e_2) \mid \texttt{fst}\ e \mid \texttt{snd}\ e \\ & \mid & \texttt{let pair}\ (x, y) = e_1\ \texttt{in}\ e_2 \\ v & ::= & \cdots \mid (v_1, v_2) \end{array}$$

$\boxed{e \Downarrow v}$ for pairs

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(e_1, e_2) \Downarrow (v_1, v_2)} \qquad \frac{e \Downarrow (v_1, v_2)}{\texttt{fst}\ e \Downarrow v_1} \qquad \frac{e \Downarrow (v_1, v_2)}{\texttt{snd}\ e \Downarrow v_2}$$

$$\frac{e_1 \Downarrow (v_1, v_2) \quad e_2[v_1/x, v_2/y] \Downarrow v}{\texttt{let pair}\ (x, y) = e_1\ \texttt{in}\ e_2 \Downarrow v}$$

Types for Pairs

- Types for pair expressions:

$$\tau \quad ::= \quad \cdots \mid \tau_1 \times \tau_2$$

$\boxed{\Gamma \vdash e : \tau}$ for pairs

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst}\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{snd}\ e : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \times \tau_2 \quad \Gamma, x : \tau_1, y : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{let pair}\ (x, y) = e_1\ \texttt{in}\ e_2 : \tau}$$

`let` vs. `fst` and `snd`

- The `fst` and `snd` operations are definable in terms of `let pair`:

$$\begin{array}{rcl} \texttt{fst}\ e & \iff & \texttt{let pair}\ (x, y) = e\ \texttt{in}\ x \\ \texttt{snd}\ e & \iff & \texttt{let pair}\ (x, y) = e\ \texttt{in}\ y \end{array}$$

- Actually, the `let pair` construct is definable in terms of `let`, `fst`, `snd` too:

$$\begin{array}{l} \texttt{let pair}\ (x, y) = e\ \texttt{in}\ e_2 \\ \quad \iff \texttt{let}\ p = e\ \texttt{in}\ e_2[\texttt{fst}\ p/x, \texttt{snd}\ p/y] \end{array}$$

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

More generally: tuples and records

- Nothing stops us from adding triples, quadruples, ..., n-tuples.

$$(1, 2, 3) \qquad (\texttt{true}, 2, 3, \lambda x.(x, x))$$

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

$$(a : 1, b : 2, c : 3) \qquad (b : \texttt{true}, n_1 : 2, n_2 : 3, f : \lambda x.(x, x))$$

- (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`.

$$
\begin{array}{rcl}
e & ::= & \cdots \mid () \\
v & ::= & \cdots \mid () \\
\tau & ::= & \cdots \mid \texttt{unit}
\end{array}
$$

$$
\frac{}{() \Downarrow ()} \qquad \frac{}{\Gamma \vdash () : \texttt{unit}}
$$

- 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 `()`.

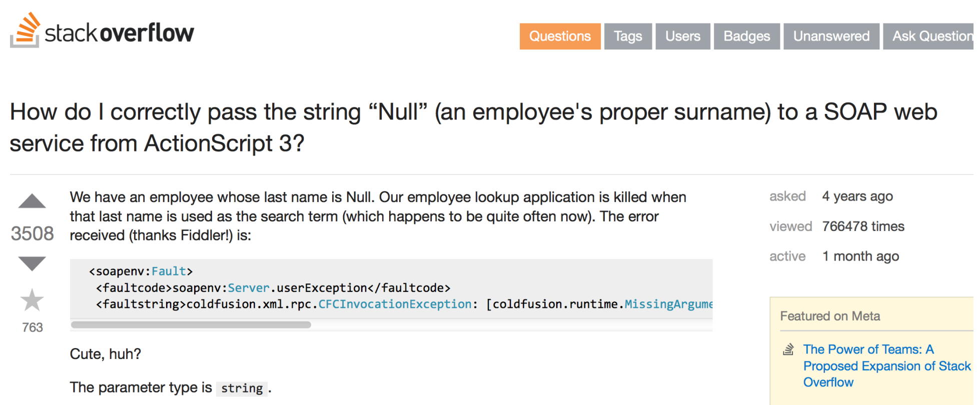
Motivation for variant types

- Pairs allow us to combine two data structures (a τ_1 and a τ_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": `http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare`

Another problem with `Null`



What would be better?

- Consider an *option type*:

$$e ::= \cdots \mid \texttt{none} \mid \texttt{some}(e)$$
$$\tau ::= \cdots \mid \texttt{option}[\tau]$$

$$\frac{}{\Gamma \vdash \texttt{none} : \texttt{option}[\tau]} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{some}(e) : \texttt{option}[\tau]}$$

- Then we can use `none` to indicate absence of a value, and $\texttt{some}(e)$ to give the present value.
- Morover, 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 τ 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 τ_{ok} *or* an error value τ_{err}.
- Let's write $\texttt{okOrErr}[\tau_{ok}, \tau_{err}]$ for this type.

$$e ::= \cdots \mid \texttt{ok}(e) \mid \texttt{err}(e)$$
$$\tau ::= \cdots \mid \texttt{okOrErr}[\tau_1, \tau_2]$$

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

How do we use $\texttt{okOrErr}[\tau_{ok}, \tau_{err}]$?

- When we talked about $\texttt{option}[\tau]$, we didn't really say how to *use* the results.
- If we have a $\texttt{okOrErr}[\tau_{ok}, \tau_{err}]$ value v, then we want to be able to *branch* on its value:
  - If v is $\texttt{ok}(v_{ok})$, then we probably want to get at v_{ok} and use it to proceed with the computation
  - If v is $\texttt{err}(v_{err})$, then we probably want to get at v_{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:

$$\texttt{case}\ e\ \texttt{of}\ \{\texttt{ok}(x) \Rightarrow e_{ok}\ ;\ \texttt{err}(y) \Rightarrow e_{err}\}$$

- This is a generalized conditional: "If e evaluates to $\texttt{ok}(v_{ok})$, then evaluate e_{ok} with v_{ok} replacing x, else it evaluates to $\texttt{err}(v_{err})$ so evaluate e_{err} with v_{err} replacing y."
- Here, x is bound in e_{ok} and y is bound in e_{err}
- This construct should be familiar by now from Scala:

```
e match { case Ok(x) => e1
          case Err(x) => e2
} // note slightly different syntax
```


Variant types, more generally

- Notice that the `ok` and `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:

$$\begin{array}{lcl} e & ::= & \cdots \mid \texttt{left}(e) \mid \texttt{right}(e) \\ & \mid & \texttt{case}\ e\ \texttt{of}\ \{\texttt{left}(x) \Rightarrow e_1\ ;\ \texttt{right}(y) \Rightarrow e_2\} \\ v & ::= & \cdots \mid \texttt{left}(v) \mid \texttt{right}(v) \\ \tau & ::= & \cdots \mid \tau_1 + \tau_2 \end{array}$$

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

Types for variants

- We extend the typing rules as follows:

$\boxed{\Gamma \vdash \tau}$ for variant types

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{left}(e) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{right}(e) : \tau_1 + \tau_2}$$

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

- Idea: `left` and `right` "wrap" τ_1 or τ_2 as $\tau_1 + \tau_2$
- Idea: Case is like conditional, only we can use the wrapped value extracted from $\texttt{left}(v)$ or $\texttt{right}(v)$.

Semantics of variants

- We extend the evaluation rules as follows:

$\boxed{e \Downarrow v}$ for variant types

$$\frac{e \Downarrow v}{\texttt{left}(e) \Downarrow \texttt{left}(v)} \qquad \frac{e \Downarrow v}{\texttt{right}(e) \Downarrow \texttt{right}(v)}$$

$$\frac{e \Downarrow \texttt{left}(v_1) \quad e_1[v_1/x] \Downarrow v}{\texttt{case}\ e\ \texttt{of}\ \{\texttt{left}(x) \Rightarrow e_1\ ;\ \texttt{right}(y) \Rightarrow e_2\} \Downarrow v}$$

$$\frac{e \Downarrow \texttt{right}(v_2) \quad e_2[v_2/y] \Downarrow v}{\texttt{case}\ e\ \texttt{of}\ \{\texttt{left}(x) \Rightarrow e_1\ ;\ \texttt{right}(y) \Rightarrow e_2\} \Downarrow v}$$

- 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 $\texttt{unit} + \texttt{unit}$

$$\texttt{true} \iff \texttt{left}() \qquad \texttt{false} \iff \texttt{right}()$$

- Conditional is then defined as case analysis, ignoring the variables

$$\begin{array}{l}\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \\ \quad \iff \texttt{case } e \texttt{ of } \{\texttt{left}(x) \Rightarrow e_1 \; ; \; \texttt{right}(y) \Rightarrow e_2\}\end{array}$$

- Likewise, the option type is definable as $\tau + \texttt{unit}$:

$$\texttt{some}(e) \iff \texttt{left}(e) \qquad \texttt{none} \iff \texttt{right}()$$

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 *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

$$\tau \ ::= \ \cdots \mid \texttt{empty}$$

  with *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 *subtyping*
  - (Insert *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