UNIVERSITY OF Southampton

School of Electronics and Computer Science

COMP2212
PROGRAMMING LANGUAGE CONCEPTS
LECTURE 10

Julian Rathke and Pawel Sobocinski

STRUCTURED TYPES

THE SHAPE OF DATA

- Programs manipulate data.
- Data comes in different shapes and sizes.
- To check that programs our doing what we want we can *at least* specify their behaviour in terms of the shape of the data they manipulate.
  - For example, a pair of integers is different shape data to two integers stored in an array.
  - **Structured Types** can make this distinction.
- Goal: catch common programming errors where data being passed to a function or library call is in a different format to what the function or library code is expecting.

- So what are the commonly used shapes of data and how do we express them as types and how do we type check programs that use structured types?

EXAMPLE 1: BOOL

- Easy. The Bool type is simple: it has two distinct elements. Programs can pattern match over each of these. e.g.

```
match b with  true → ... | false → ...
```

THE UNIT TYPE

- This is a surprisingly useful type and is easy to understand.
  - It has exactly one value, usually written as ( ). We'll write the type as **`unit`**
  - Indeed, if you enter **`:type ()`** at the Haskell top-level you get **`()::()`** in response. In Haskell, the unit type is, confusingly, also written as ( )
- This is the type of Java methods that take no parameters: **`int foo( ){ ... }`**
- It can be used to *suspend* the evaluation of an expression until a later point in the computation. For example compare
  - **`let x = print "Hello" in ..x..   ;;`** and
  - **`let x = fun () → print "Hello"  in  ..x()..  ;;`**
  - In a CBV language the former immediately prints hello and goes on with the evaluation.
  - The latter wraps the print statement in to a function for later use.
  - This operation of wrapping an expression with a function of unit type is called **thunking** . We can talk of **unthunking** a thunked value too.
- The type rule for unit values is remarkably easy:

$$\frac{}{\vdash (\,) : \mathbf{unit}}$$

PAIRS AND TUPLES

- Pairs are another very common structured type.
- Given two pieces of data of types T and U then we can form a piece of data of shape $T \times U$. We usually write this as a pairing operation (E_1, E_2)
- The type rule for this is straightforward:

$$\frac{\vdash E_1 \ : \ T \qquad \vdash E_2 \ : \ U}{\vdash (E_1, E_2) \ : \ T \times U}$$

(unit) is a type of: unit

- It is also straightforward to generalise this to arbitrary tuples:
  - The *constructor* for general tuples is $(E_1, E_2, \ldots, E_n)$
  - and the type rule is

$$\frac{\vdash E_1 \ : \ T_1 \qquad \vdash E_2 \ : \ T_2 \qquad \ldots \qquad \vdash E_n \ : \ T_n}{\vdash (E_1, E_2, \ldots, E_n) \ : \ T_1 \times T_2 \times \ldots \times T_n}$$

DESTRUCTORS

- We have referred to the operation (E_1,E_2) as the constructor for the pair datatype.
- There are corresponding operations that we refer to as **destructors** for pairs.
- These are called **projections**.
- The first projection is called `fst`. It returns the first component of the pair. The second projection `snd` returns the second component of the pair.
- There are type rules for these operations:

$$\frac{\vdash E \ : \ T \times U}{\vdash \mathsf{fst}\ E \ : T} \qquad \frac{\vdash E \ : \ T \times U}{\vdash \mathsf{snd}\ E \ : U}$$

- We can also use pattern matching to take apart the structured data

```
match E with  (E1, E2) → ...
```

- Of course, for generalised n-tuples we need n projection functions.

RECORD TYPES

- We can generalise tuples even further. What they are essentially are are collections of n elements of data indexed by integers.
- More generally, we could use labels to index the items. This is what is known as a **record** we use these extensively in C (as struct types) and in Java (as objects !)
- The usual syntax for record constructors is $\{ l_1 = E_1, l_2 = E_2, \dots, l_n = E_n \}$ where the l_i labels are the *fields* of the record and the E_i are the values stored.
- The type of data of this shape is written similarly: $\{ l_1 : T_1, l_2 : T_2, \dots, l_n : T_n \}$
- The destructors for this type is called **selection** and has a very familiar notation: we write $R \,.\, l_i$ to select the field labelled l_i from record R
- The type rule for records looks like this:

$$\frac{\vdash E_i : T_i \qquad \text{for } 1 \leq i \leq n}{\vdash \{l_1 = E_1, l_2 = E_2, \dots, l_n = E_n\} : \{l_1 : T_1, l_2 : T_2, \dots, l_n : T_n\}}$$

$$\frac{\vdash R : \{l_1 : T_1, l_2 : T_2, \dots, l_n : T_n\}}{\vdash R.l_i : T_i}$$

OTHER STRUCTURES? LISTS AND ARRAYS

- List structures are very common across mainstream languages.
- Given type T we can form the type T List
- The constructor for Lists is cons, often written as **: :**
- Destructors for lists are head and tail
- Type rules:

$$\frac{\vdash E : T \qquad \vdash ES : T \ \mathsf{List}}{\vdash E :: ES : T \ \mathsf{List}} \qquad \frac{\vdash ES : T \ \mathsf{List}}{\vdash \mathsf{hd} \ ES : T} \qquad \frac{\vdash ES : T \ \mathsf{List}}{\vdash \mathsf{tl} \ ES : T \ \mathsf{List}}$$

- Arrays feel similar but in fact **aren't** really a structural type. They have no constructor to form elements of the type and we can't pattern match across them.
- However, it still makes sense to have type rules for them (using an array-like syntax) :

$$\frac{\vdash E_i : T \text{ for } 1 \leq i \leq n}{\vdash [E_1, \ldots E_n] : T \ \mathsf{Array}} \qquad \frac{\vdash E : T \ \mathsf{Array} \qquad \vdash I : \ \mathsf{Int}}{\vdash E[I] : T}$$

ANOTHER NON-STRUCTURE TYPE

- A very common type former that we see in one form or another in mainstream languages is that of function types.
- Given a type T and a type U we can form the type T ➞ U. Functions from T to U.
- For *higher-order* functional languages (such as OCaml, Haskell etc) T can be any type at all (including function types).
- For *first-order* languages, such as C, C++, then T is restricted to being a primitive type or a structure built from primitive types.
- Interestingly, the function type is **not** a structure type: we can't pattern match over it.
- The constructor for the type is lambda abstraction

$$\frac{\Gamma, x : T \vdash E : U}{\Gamma \vdash \textsf{fun}\ (x : T)\ \rightarrow E : T \rightarrow U}$$

- Can you see why this is not a structured value?

NEXT LECTURE: SUM TYPES