COMP2212
PROGRAMMING LANGUAGE CONCEPTS
LECTURE 10
Julian Rathke and Pawel Sobocinski
STRUCTURED TYPES
THIS LECTURE IS ALL ABOUT SHAPES

• Programs manipulate data.
• Data comes in different shapes and sizes.
• To check that programs are 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?
• 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().. ;;
  • 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:

\[ \vdash () : \text{unit} \]
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 × U. We usually write this as a pairing operation (E₁, E₂)
• The type rule for this is straightforward:

\[
\frac{\vdash E_1 : T \quad \vdash E_2 : U}{\vdash (E_1, E_2) : T \times U}
\]

• It is also straightforward to generalise this to arbitrary tuples:
  • The constructor for general tuples is (E₁,E₂,...,Eₙ)
  • and the type rule is

\[
\frac{\vdash E_1 : T_1 \quad \vdash E_2 : T_2 \quad \ldots \quad \vdash E_n : T_n}{\vdash (E_1, E_2, \ldots, E_n) : T_1 \times T_2 \times \ldots \times T_n}
\]
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 \texttt{fst}. It returns the first component of the pair. The second projection \texttt{snd} returns the second component of the pair.
• There are type rules for these operations:

\[
\frac{}{\vdash E : T \times U} \quad \frac{}{\vdash E : T \times U} \\
\frac{}{\vdash \texttt{fst} \ E : T} \quad \frac{}{\vdash \texttt{snd} \ E : U}
\]

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

\[
\text{match } E \text{ with } (E_1, E_2) \rightarrow \ldots
\]

• 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, \ldots, 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, \ldots, l_n : T_n \}
- The destructors for this type is called **selection** and has a very familiar notation: we write \( R \cdot l_i \) to select the field labelled l_i from record R
- The type rule for records looks like this:

\[
\begin{align*}
\vdash & E_i : T_i \quad \text{for } 1 \leq i \leq n \\
\vdash & \{ l_1 = E_1, l_2 = E_2, \ldots, l_n = E_n \} : \{ l_1 : T_1, l_2 : T_2, \ldots, l_n : T_n \} \\
\vdash & R : \{ l_1 : T_1, l_2 : T_2, \ldots, l_n : T_n \} \\
\vdash & R.l_i : T_i
\end{align*}
\]
SUM TYPES

• A pair types $T \times U$ represent structures in which data of both type $T$ and $U$ is present.

• Sometimes we want a type that represents the possibility of data of either type $T$ or type $U$ being present.

• This is known as a sum type. It is usually written as $T + U$.

• The constructors for this type are called injections and are written inl and inr.

• For example, \textbf{inl 5} could be an element of type $\text{Int} + \text{Bool}$, or \textbf{inr 10} could be an element of type $\text{Int} + \text{Int}$.

• Let’s look at the type rules for injections:

$$
\begin{align*}
\vdash E : T & \quad \vdash E : U \\
\mid \quad \vdash \text{inl } E : T + U & \quad \mid \quad \vdash \text{inr } E : T + U
\end{align*}
$$

• The only destructor for this type is the pattern matching operator.

• Sometimes this is called case rather than the more general match.

• For example, \textbf{case $E$ of inl $x \rightarrow E_1$ | inr $x \rightarrow E_2$}
UNIQUENESS OF SUM TYPES

- You might have noticed that I said above that \texttt{inl 3} could have type \texttt{Int + Bool}
- In fact, it could also have type \texttt{Int + Int}, or indeed \texttt{Int + AnyOtherType}
- If you look at the type rule for injection again you can see why:
  \[
  \frac{\vdash E : T}{\vdash \text{inl } E : T + U}
  \]
- Where does U come from?
- With such a rule, expressions in a language with these sums would fail to have unique types. This can be a complication for type checking - but not a deal breaker.
- One way out of this is to choose the names of the injections to be unique for each different sum type. For example, \texttt{Int + Int} may have different injections to \texttt{Int + Bool}.
- For example, we could choose \texttt{inlft 2} to be of type \texttt{Int + Int}, whereas \texttt{inL 2} would be uniquely of type \texttt{Int + Bool}.
- Of course, the choice of names here is arbitrary. They are just labels \texttt{l1} and \texttt{l2}.
- Indeed, we need not stop at two summands in the type. We could have a type made from \texttt{T1 + T2 + ... + Tn} with injections \texttt{l1}, \texttt{l2}, ..., \texttt{ln} etc
- This is starting to look familiar.
VARIANT TYPES

• Just in the same way that record types are a generalisation of tuples, Variant types are a generalisation of sums.
• The type of variants is written: \(< l_1 : T_1, l_2 : T_2, ..., l_n : T_n >\)
• The constructors for the type are injections named with labels: \(<l_i = E>\)
• The type rules are

\[
\frac{\Gamma \vdash E : T_i}{\Gamma \vdash \langle l_i = E \rangle : \langle l_1 : T_1, l_2 : T_2, ..., l_n : T_n \rangle}
\]

\[
\frac{\Gamma \vdash E : \langle l_1 : T_1, l_2 : T_2, ..., l_n : T_n \rangle \quad \Gamma \vdash x : T_i \vdash E_i : T \quad \text{for } 1 \leq i \leq n}{\Gamma \vdash \text{case } E \text{ of } \langle l_1 = x \rangle \to E_1 \mid ... \mid \langle l_n = x \rangle \to E_n : T}
\]
UNIQUENESS OF VARIANT TYPES

• Consider two different variant types:
  \[ T = \langle \text{Left} : \text{Int}, \text{Right} : \text{Int} \rangle \quad \text{and} \quad U = \langle \text{Wrong} : \text{Int}, \text{Right} : \text{Int} \rangle \]
• What type does the value \( \langle \text{Right} = 0 \rangle \) have?
• It is both of type \( T \) and \( U \)
• But we seem to have lost uniqueness of our types again.
• Where variant types allow arbitrary labels, it is useful to insist that labels are unique among different types.
• What Haskell does in this situation: it doesn’t give an error but simply infers that any value tagged with ‘Right’ to be of the most recently defined variant using that tag.
ENumerations

• Sometimes you just want to write a variant type for its labels
• E.g. a type for days of the week.
• In this case having to use a variant type and give a type to each of these labels would be annoyingly verbose.
• We can easily choose the unit type for this:

\[
\text{Data Day} = \text{Mon of unit} \mid \text{Tue of unit} \mid \ldots \mid \text{Sun of unit}
\]

• which will have values of the form \(< \text{Mon} = () > \) etc.
• An enumerated type is simply a sugar for exactly this structure though.
• Many languages allow us to write

\[
\text{data Day} = \text{Mon} \mid \text{Tue} \mid \text{Wed} \mid \text{Thu} \mid \text{Fri} \mid \text{Sat} \mid \text{Sun}
\]

• The values of the type are simply the labels \text{Mon}, \text{Tue} etc.
• We can also mix and match labels of type unit, suitably sugared, with labels of non-unit type.

• The most prominent example in Haskell is the option type.
  • `data Maybe a = Nothing | Just a`

• This is a variant type with a `Nothing` field (of implicit unit type) and a `Just` field of some other type.

• `Nothing` is a genuine value of this type.
A TYPE RULE FOR MATCH?

- Suppose that we wanted to introduce pattern matching in to our Toy language.
- We could do this with an operator called `match - with` a lot in OCaml.
- What would be the type rule for such an operator?
- It would be used as a general operator for tearing apart data structures.
- The general form of the operator syntax would be

```
match E with p₁ → E₁ | ... | pₙ → Eₙ
```

where the `pᵢ` are patterns.

- The type rule would be something like (very roughly):

```
\[ \frac{E : U \quad pᵢ : U \quad pᵢ^* \vdash Eᵢ : T}{\vdash \text{match } E \text{ with } p₁ \rightarrow E₁ | ... | pₙ \rightarrow Eₙ : T} \]
```

- where `pᵢ^*` embodies assumptions about the variables bound in the pattern match.
- This requires a type system for the language of patterns.
- It gets a bit complicated.
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:

  \[ \frac{E : T \quad ES : T \text{ List}}{E :: ES : T \text{ List}} \quad \frac{ES : T \text{ List}}{\text{hd } ES : T} \quad \frac{ES : T \text{ List}}{\text{tl } ES : T \text{ List}} \]

• 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) :

  \[ \frac{E_i : T \quad \text{for } 1 \leq i \leq n}{[\mid E_1; \ldots; E_n \mid] : T \text{ Array}} \quad \frac{E : T \text{ Array}}{E.(I) : T} \]
A 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 $T \rightarrow U$. 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

\[
\Gamma, x : T \vdash E : U \\
\Gamma \vdash \text{fun} \ (x : T) \rightarrow E : T \rightarrow U
\]

- This rule forms the basis of most functional languages!
NEXT LECTURE: SUBTYPING