UNIVERSITY OF Southampton
School of Electronics and Computer Science

COMP2212 PROGRAMMING LANGUAGE CONCEPTS LECTURE 10

Julian Rathke and Pawel Sobocinski

SUM TYPES

SUM TYPES

- A pair types T x 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, `inl 5` could be an element of type Int + Bool, or `inr 10` could be an element of type Int + Int
- Let's look at the type rules for injections:

$$\frac{\vdash E : T}{\vdash \text{inl } E : T + U} \qquad \frac{\vdash E : U}{\vdash \text{inr } E : T + U}$$

- The only destructor for this type is the pattern matching operator.
- Sometimes this is called `case` rather than the more general `match`
- For example, **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 `inl 3` could have type Int + Bool
- In fact, it could also have type Int + Int, or indeed Int + AnyOtherType
- If you look at the type rule for injection again you can see why: $$\frac{\vdash E : T}{\vdash \textsf{inl}\ E : T + U}$$
- 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, Int + Int may have different injections to Int + Bool.
- For example, we could choose inlft 2 to be of type Int + Int, whereas inL 2 would be uniquely of type Int + Bool.
- Of course, the choice of names here is arbitrary. They are just labels l_1 and l_2 .
- Indeed, we need not stop at two summands in the type. We could have a type made from $T_1 + T_2 + ... + T_n$ with injections l_1 , l_2 , ... , l_n 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 : $< l_1 : T_1,\ l_2 : T_2 , \ldots ,\ l_n : T_n >$
- The constructors for the values are injections named with labels: $<l_i = E>$
- The type rules are

$$\frac{\vdash E : T_i}{\vdash \langle l_i = E\rangle : \langle l_1 : T_1, l_2 : T_2, \ldots, l_n : T_n\rangle}$$

$$\frac{\vdash E : \langle l_1 : T_1, l_2 : T_2, \ldots, l_n : T_n\rangle \qquad x : T_i \vdash E_i : T \qquad \text{for } 1 \leq i \leq n}{\vdash \mathsf{case}\ E\ \mathsf{of}\ \langle l_1 = x\rangle \to E_1 \mid \ldots \mid \langle l_n = x\rangle \to E_n : T}$$

UNIQUENESS OF VARIANT TYPES

- Consider two different variant types :
- $T = < \text{Left} : \text{Int} , \text{Right} : \text{Int} >$ and $U = < \text{Wrong} : \text{Int} , \text{Right} : \text{Int} >$
- What type does the value $< \text{Right} = 0 >$ 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:

```
data Day = Mon of unit | Tue of unit | ... | Sun of unit
```

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

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

- The values of the type are simply the labels **`Mon`**, **`Tue`** etc.

OPTION TYPES

- 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.
- Similar to the operator `case - of - →` in Haskell.
- 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 something like

`match E with p₁ → E₁ | ... | pₙ → Eₙ` where the p_i are patterns.

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

$$\frac{\vdash E : U \qquad \vdash p_i : U \qquad p_i^* \vdash E_i : T}{\vdash \textsf{match}\ E\ \textsf{with}\ p_1 \to E_1 \mid \ldots p_n \to E_n : T}$$

- where p_i* 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.

NEXT LECTURE: SUBTYPING