Outline

Types and Datatypes

Type Generality

Equality

Syntactic Sugar

Pattern Matching
Types

- What does int * int * int mean?
Types

- What does int * int * int mean?
- In HW1, we use it as a **date** type
Types

- What does int * int * int mean?
- In HW1, we use it as a date type
- Can we make the semantics of this type more explicit?
Types

- What does \texttt{int * int * int} mean?
- In HW1, we use it as a \texttt{date} type
- Can we make the semantics of this type more explicit?

\begin{verbatim}
  type date = int * int * int
\end{verbatim}
Types vs. DataTypes

A **datatype** introduces a new type distinct from all existing types

datatype suit = Club | Diamond | Heart | Spade
datatype rank = Jack | Queen | King | Ace
| Num of int
Types vs. DataTypes

A **datatype** introduces a new type distinct from all existing types

```haskell
datatype suit = Club | Diamond | Heart | Spade
datatype rank = Jack | Queen | King | Ace
    | Num of int
```

A **type** (aka *type synonym*) is just another name

```haskell
type card = suit * rank
```
Types vs. DataTypes

A **datatype** introduces a new type distinct from all existing types

datatype suit = Club | Diamond | Heart | Spade
datatype rank = Jack | Queen | King | Ace
   | Num of int

A **type** (aka *type synonym*) is just another name

type card = suit * rank

datatypes have constructors but type synonyms don’t!
Types vs. DataTypes

A **datatype** introduces a new type distinct from all existing types.

```haskell
datatype suit = Club | Diamond | Heart | Spade
datatype rank = Jack | Queen | King | Ace
             | Num of int
```

A **type** (aka *type synonym*) is just another name.

```haskell
type card = suit * rank
```

datatypes have constructors but type synonyms don’t!

Why use type synonyms?

- For now, just for convenience
- Later we will see another use for type synonyms in the **modules** unit
Type Generality

Write a function that appends two string lists
Type Generality

Write a function that appends two string lists

What happened?
Write a function that appends two string lists

What happened?

We thought we needed

\[
\text{string list} \times \text{string list} \rightarrow \text{string list}
\]

But the type checker found and used

\[
\text{'a list} \times \text{'a list} \rightarrow \text{'a list}
\]

Why is this OK?
Type Generality

More General Types

The type

\[ \text{'a list} \times \text{'a list} \to \text{'a list} \]

is more general than the type

\[ \text{string list} \times \text{string list} \to \text{string list} \]

More general types can be used in place of less general types, for example

\[ \text{int list} \times \text{int list} \to \text{int list} \]
The type

'a list * 'a list -> 'a list

is more general than the type

string list * string list -> string list

More general types can be used in place of less general types, for example

int list * int list -> int list

Is 'a list * 'a list -> 'a list more general than
int list * string list -> int list?
Type Generality
The ”Type Generality Rule”

A type $t_1$ is more general than the type $t_2$ if you can take $t_1$, replace its type variables consistently, and get $t_2$

What does consistently mean?
Equality

Write a "list contains" function
Equality

Write a "list contains" function

Equality Types

- The double quoted variable arises from use of the = operator
- We can use = on most types like int, bool, string, tuples (that contain only "equality types")
- Generality rules work the same, except substitution must be some type which can be compared with =
Equality

Write a “list contains” function

Equality Types
- The double quoted variable arises from use of the = operator
- We can use = on most types like int, bool, string, tuples (that contain only “equality types”)
- Generality rules work the same, except substitution must be some type which can be compared with =
- Functions and real are not ”equality types”
- You can ignore warnings about “calling polyEqual”
if-then-else

if-then-else is *syntactic sugar* for a case expression:
if-then-else is *syntactic sugar* for a case expression:

```plaintext
if x then "apple" else "banana"
```

can be written as

```plaintext
case x of true => "apple" | false => "banana"
```
Syntactic Sugar

Logical Operators

andalso and orelse are also forms of *syntactic sugar*!
and also and orelse are also forms of *syntactic sugar*!

Given

```scala
val x = true
val y = false
```

Logical "and" can be written as

```scala
val x_and_y = x andalso y
```

or this

```scala
val x_and_y = case x of true => y | false => false
```
Adventures in pattern matching!

**SML code we write / look at together will be available on the course website**