Today's Agenda

• Testing
• Lists, Let-Expression (Review)
• Options
• Type synonyms
• Type generality
• Equality types
• Syntactic sugar

Reminder: Check out the CSE341 style guide as you work on HW! Also check out the style guides in section 1 slide!

Section Learning Objectives

• Review building/accessing new types (e.g. datatypes)
• Recognize type synonyms as “convenient” feature
• Be able to generalize specific types with polymorphism (e.g. int list into ‘a list) and equality types
• Practice using pattern-matching with case expressions

Lists

• Lots of new types: For any type t, the type t list describes lists where all elements have type t
  o Examples: int list, bool list, (int * int) list, (int list * int) list
• So [] can have type t list for any type t
  o SML uses ‘a list to indicate this (“tick a” or “alpha”)
• For e1::e2 to type-check, we need a t such that e1 has type t and e2 has type t list. Then the result type is t list
  o null: ‘a list -> bool
  o hd: ‘a list -> ‘a
  o tl: ‘a list -> ‘a list

Let-Expression

• Syntax: let b1 b2 ... bn in e end
  o Each bi is any binding and e is any expression
  o Type-checking: Type-check each bi and e in a static environment that includes the previous bindings.
  o Type of whole let-expression is the type of e.
  o Evaluation: Evaluate each bi and e in a dynamic environment that includes the previous bindings.

Result of whole let-expression is result of evaluating e.
Options

t option is a type for any type t
  • (much like t list, but a different type, not a list)

Building:
  • NONE has type 'a option (much like [] has type 'a list)
  • SOME e has type 'a option if e has type t (much like e::[])

Accessing:
  • isSome has type 'a option -> bool
  • valOf has type 'a option -> 'a (exception if given NONE)

Type Synonyms

• What does \( \texttt{int} \times \texttt{int} \times \texttt{int} \) represent?

• In HW1 we called it a date

• Wouldn't it be nice to reflect this representation in the source code itself?

\[
\text{type date} = \texttt{int} \times \texttt{int} \times \texttt{int}
\]

Datatypes

• What if we want something unique? A new type?

• We can't just use type synonyms because they can only be built from existing types.

• Datatypes give us the ability to define custom types.

\[
\text{datatype foo} = \texttt{bar} | \texttt{baz} \text{ of int} | \texttt{qux} \text{ of bool}
\]

type vs datatype

• \texttt{datatype} introduces a new type name, distinct from all existing types

\[
\begin{align*}
\text{datatype suit} &= \texttt{Club} | \texttt{Diamond} | \texttt{Heart} | \texttt{Spade} \\
\text{datatype rank} &= \texttt{Jack} | \texttt{Queen} | \texttt{King} | \texttt{Ace} \\
&\quad | \text{Num of int}
\end{align*}
\]

• \texttt{type} is just another name

\[
\text{type card} = \texttt{suit} \times \texttt{rank}
\]

Type Synonyms

Why?
  • For now, just for convenience
  • It doesn't let us do anything new

Later in the course we will see another use related to modularity.

Type Generality

Write a function that appends two string lists...
Type Generality

- We would expect
  \[\text{string list} \times \text{string list} \rightarrow \text{string list}\]
- But the type checker found
  \[\text{'a list} \times \text{'a list} \rightarrow \text{'a list}\]
- 'a are called Polymorphic Types
- Why is this OK?

More General Types

- The type
  \[\text{'a list} \times \text{'a list} \rightarrow \text{'a list}\]
  is more general than the type
  \[\text{string list} \times \text{string list} \rightarrow \text{string list}\]
  and “can be used” as any less general type, such as
  \[\text{int list} \times \text{int list} \rightarrow \text{int list}\]
- But it is not more general than the type
  \[\text{int list} \times \text{string list} \rightarrow \text{int list}\]

The Type Generality Rule

The “more general” rule

A type \(t_1\) is more general than the type \(t_2\) if you can take \(t_1\), replace its type variables \(\text{consistently}\), and get \(t_2\)

What does \(\text{consistently}\) mean?

Equality Types

Write a list “contains” function...

Equality Types

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

More Syntactic Sugar

- Tuples are just records
- If-then-else is implemented as syntactic sugar for a case statement
**If-then-else**

- We've just covered case statements
- How could we implement if-then-else

```haskell
case x of
    true => "apple"
  | false => "banana"

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

**val-Pattern Matching**

Remember our unit test?

```
(* Neat trick for creating hard-fail tests: *)
val true = ((4 div 4) = 1);
```

Just a pattern match!

"Match the left hand side against the value 'template' true, binding any variables (there aren't any!)"

**Adventures in pattern matching**

- Shape example
- Function-pattern syntax if we get to it