CSE 341
Lecture 9

type systems; type safety; defining types
Ullman 6 - 6.2; 5.3

slides created by Marty Stepp
http://www.cs.washington.edu/341/

Types

- **data type**: A classification of a set of values, a set of operations that can be performed on the values, and a description of how the values are stored/represented.

- All languages include a set of pre-defined types.
- Most also allow the programmer to define new types.

Classifications of type systems

- **type checking**: Verifying/enforcing constraints of types.
  - Example: The `length` function must return an `int`.
    Example: a^b only works when a and b are strings.

- **static typing**: Type checking is done at compile time.
  - Java, C, C++, C#, ML, Haskell, Go, Scala

- **dynamic typing**: Type checking is done at run time.
  - Scheme, JavaScript, Ruby, PHP, Perl, Python

Static vs. dynamic typing

- static
  - +: avoids many run-time type errors; verifiable
  - -: code is more rigid to write and compile; less flexible
- dynamic
  - +: more flexible (can generate type/features at runtime)
  - -: can have type errors; errors may not be discovered; code must perform lots of type checks at runtime (slow)
- both can be used by the same language
  - Java type-checks some aspects of object type-casting at runtime (throws a `ClassCastException` on type errors)

Type safety, "strong" vs. "weak"

- **type error**: Erroneous or undesirable program behavior caused by discrepancy between differing data types.

- **type safety**: Degree to which a language prevents type errors.
  - ML is safe; syntactically correct code has no type errors

- **strong typing**, **weak typing**: Whether the language has severe or relaxed restrictions on what operations allow types to mix (what implicit type conversions are allowed).
  - `string + int?  int * real?  string = char?`

  (strong/weak are vaguely defined, outdated terms)

Lack of type safety in C

```
int main(int argc, char** argv) {
    char msg[4] = "abc";           // [97, 98, 99, 0]
    int* num = (int*) msg;

    printf("%s\n",  msg);          // abc
    printf("%d\n", *num);          // 6513249
    (*num)++;                 // (97 + 98*2^8 + 99*2^16 + 0*2^24)

    printf("%d\n", *num);          // 6513250
    printf("%s\n",  msg);          // bbc
}
```

- The code allows a string (`char[]`) to be interpreted as though it were an `int`! This is unsafe.
- C is the poster child for unsafe languages...

More lack of type safety in C

```
int main(int argc, char** argv) {
    int a1[1] = {42};            // two 1-element arrays
    int a2[1] = {78};
    a2[1] = 999;                 // out of bounds!
    printf("%d\n", a1[0]);       // 999 !!
}
```

- C does not check array bounds. If you go past the end of the array, you write into the next piece of memory.
  - In this case, that memory happens to refer to `a1`...
  - can lead to corrupt data elsewhere or crashes ("segfaults")
  - many security bugs, viruses, etc. are from OS/app C code that mistakenly goes past the end of an array on certain input

Parametric polymorphism

- What are the types of hd and `tl`? (and `length`?)

```
- hd;
```
val it = fn : 'a list -> 'a

```
- tl;
```
val it = fn : 'a list -> 'a list

- **parametric polymorphism**: ability of a function to handle values identically without depending on their type
  - language is more expressive; still handles types properly
  - similar to generics in Java (e.g. `ArrayList<String>`)
  - Does parametric polymorphism conflict with type safety?

More about polymorphism

- Some functions have unbounded or generalized types:

```
- fun identity(x) = x;
```

val identity = fn : 'a -> 'a

- Those types can become bounded on a particular call:

```
- fun foo(x) = if x then identity else abs;
```

  - What is the type of `foo`?

- Some operators destroy/reduce a value's polymorphism:
  - yes: `+ - ~ * / div mod < <= >= > andalso orelse`
    `not ^ ord chr real str floor ceil round trunc`
  - no: `:: @ hd tl nil [] = <> #`*`n`*`(`*`tuple`*`)`

Equality types (5.3)

- **equality type**: One where two of its values can be directly tested to see whether they are "equal" to each other.

- in ML, equality types are ones that allow =, <> operators
  - `int`, `bool`, `char`, `string`
  - any tuple, list, or record containing only the above

- the following are *not* equality types:
  - `real`
  - functions
  - any tuple, list, or record containing the above

Generalized equality types

```
- fun identity(x) = x;
```
val identity = fn : 'a -> 'a

```
- fun switch(x, y) = (y, x);
```
*val switch = fn : 'a * 'b -> 'b * 'a*

- ML uses `'`   (e.g. `'a`)   for any general type
- ML uses `''` (e.g. `''a`) for any general *equality* type
  - what is the type of = and <> ?

The 'polyEqual' warning

```
- fun len(lst) =
=       if lst = [] then 0
=       else 1 + len(tl(lst));
```

stdIn:5.19 Warning: calling polyEqual

val len = fn : ''a list -> int

- ML warns us when we use = or <> on a general type.
  - It might be a logic error on our part (though not usually).
  - It is slightly slow for ML to do = or <> on general types, because it must store info about the type at runtime.
    - (Really they should have disabled this warning by default.)

      `sml -Ccontrol.poly-eq-warn=false`

Avoiding polyEqual

```
- fun len([]) = 0
= |   len(first::rest) = 1 + len(rest);
```
val len = fn : 'a list -> int

- Sometimes the = or <> test can be avoided.
  - for lists, the `null(`***`lst`***`)` function tests for [ ] without =

- Sometimes equality tests can't be avoided (it's okay):

```
fun contains([], _) = false
|   contains(first::rest, value) = first = value
                                   orelse contains(rest, value);
```

Defining a type synonym

```
type name = typeExpression;
```

- A named alias for another type or combination of types
- Examples:
  - `type fourInts = int * int * int * int;`
  - `type transcript = real list;`
- Your new type can be used elsewhere in the code:
  - `fun f(x:fourInts) = let (a,b,c,d) = x in ...`

Parameterized type synonym

```
type (params) name = typeExpression;
```

- Your synonym can be generalized to support many types
- Example:
  - `type ('a, 'b) mapping = ('a * 'b) list;`
- Supply the types to use with the parameterized type:
  - ```
    val words = [("the", 25), ("it", 12)]
              : (string, int) mapping;
    ```
    ```

Creating new types of data

```
datatype name = value | value | ... | value;
```

- a new type that contains only a fixed set of values
  - analogous to the enum in Java/C

- Examples:
  - ```
    datatype CardSuit = Clubs | Diamonds
                      | Hearts | Spades;
    ```
  - `datatype Color = Red | Green | Blue;`
  - `datatype Gender = Male | Female;`
    ```

Working with datatypes

- You can process each value of a type using patterns:

```
- fun rgb(Red) = (255, 0, 0)
  |   rgb(Green) = (0, 255, 0)
  |   rgb(Blue) = (0, 0, 255);
```

*val rgb = fn : Color -> int * int * int*

- Patterns here are just *syntactic sugar* for another fundamental ML construct called a *case expression*.

Case expressions

```
case expression of
     pattern1 => expression1
|    pattern2 => expression2
     ...
|    patternN => expressionN
```

- evaluates expression and fits it to one of the patterns
  - the overall `case` evaluates to the *match* for that pattern
- a bit like the `switch` statement in Java, with expressions

Case examples

```
- fun rgb(c) =
      case c of
            Red   => (255, 0, 0)
      |     Green => (0, 255, 0)
      |     Blue  => (0, 0, 255);
```

*val rgb = fn : Color -> int * int * int*

```
fun fib(n) =                         (* inefficient *)
    case n of 1 => 1
           |     2 => 1
           |     x => fib(x-1) + fib(x-2);
```

Equivalent expressions

- `bool` is just a `datatype`:
  - `datatype bool = true | false;`

- `if-then-else` is equivalent to a `case` expression:

```
if a then b else c        case a of
                                true  => b
                          |     false => c
```

Datatype / case exercise

- Define a method `haircutPrice` that accepts an age and gender as parameters and produces the price of a haircut for a person of that age/gender.
  - Kids' (under 10 yrs old) cuts are $10.00 for either gender.
  - For adults, male cuts are $18.25, female cuts are $36.50.

- Solution:

```
fun haircutPrice(age, gend) =
        if age < 10 then 10.00
        else case gend of Male   => 18.25
                        | Female => 36.50;
```

Type constructors

a ***TypeCtor*** is either: `name of typeExpression`

or: `value`

```
datatype name = TypeCtor | TypeCtor ...
              | TypeCtor;
```

- `datatype`s don't have to be just fixed values!
  - they can also be defined via "type constructors" that accept additional information
  - patterns can be matched against each type constructor

Type constructor example

```
(* Coffee : type, caffeinated?
   Wine   : label, year
   Beer   : brewery name
   Water  : needs no parameters *)
datatype Beverage =
    Water
|   Coffee of string * bool
|   Wine of string * int
|   Beer of string;
```

`- val myDrink = Wine("Franzia", 2009);`

val myDrink = Wine ("Franzia",2009) : Beverage

`- val yourDrink = Water;`

val yourDrink = Water : Beverage

Patterns to match type ctors

```
(* Produces cafe's price for the given drink. *)
fun price(Water) = 1.50
|   price(Coffee(type, caf)) = if caf then 3.00
                               else 3.50
|   price(Wine(label, year)) = if year < 2009
                               then 30.0 else 10.0
|   price(Beer(_)) = 4.00;
```

- functions that process datatypes use patterns
  - pattern gives names to each part of the type constructor, so that you can examine each one and respond accordingly