Notes for CSC 2/454, 26 and 28 Oct. 2020
==========
Type Systems

We have all developed an intuitive notion of what types are.

What's behind the intuition -- what ***is*** a type?
- collection of values from a "domain"
  (the mathematical/denotational approach)
- equivalence class of objects (the implementor's approach)
- internal structure of a bunch of data, described down to the level
  of a small set of fundamental types (the structural approach)
- collection of well-defined operations that can be applied to objects
  of that type (the abstraction approach)

What are types good for?
  implicit context (resolution of polymorphism and overloading)
  checking -- make sure that certain meaningless operations do not
    occur. Type checking cannot prevent all meaningless operations,
    but it catches enough of them to be useful.

Strong typing means, informally, that the language prevents you from applying an operation to data on which it is not appropriate.

Static typing means that the compiler can do all the checking at compile time. Lisp dialects are strongly typed, but not statically typed. Ada is statically typed. ML dialects are statically typed with inference. C is statically but not strongly typed. Java is strongly typed, with a non-trivial mix of things that can be checked statically and things that have to be checked dynamically.

With the proliferation of scripting languages, static v. dynamic typing has become a controversial topic. The dynamic camp argues that static type declarations add too much noise and confusion to programs, making it harder to express what you want quickly. The static camp argues that you ought to catch as much as you can ahead of time.

A ***type system*** has rules for
- type ***equivalence*** (when are the types of two values the same? --
  that is, what exactly ***are*** the types in the program?)
- type ***compatibility*** (when can a value of type A be used in a
  context that expects type B?) Note that this is directional. One
  might, for example, be allowed to use an integer everywhere a
  real is expected, but not vice versa.
- type ***inference*** (what is the type of an expression, given the types of
  the operands [and maybe the surrounding context]?)

Type compatibility / type equivalence

Compatibility is the more useful concept, because it tells you what you can ***do***.
The terms are often (incorrectly, but I do it too) used interchangeably.
Most languages say type A is compatible with (can be used in a context that expects) type B if it is equivalent or if it can be ***coerced*** to it.

Two major approaches to equivalence: ***structural*** equivalence and ***name*** equivalence. Name equivalence is based on declarations. Structural equivalence is based on some notion of meaning behind those declarations. Name equivalence is more fashionable these days, but not universal.

Structural equivalence depends on recursive comparison of type descriptions
  Substitute out all names; expand all the way to built-in types.
  Original types are equivalent if the expanded type descriptions
  are the same.

  (Pointers complicate matters, but the Algol folks figured out how to handle it in the late 1960's. The correct approach is to apply a "set of subsets" algorithm to the graph of types that point to each other, the same way one turns a non-deterministic FSM into an equivalent deterministic FSM.)

Name equivalence depends on actual occurrences of declarations in the source code.

Example:

```
struct person {
   string name;
   string address;
}
```

```
struct school {
   string name;
   string address;
}
```

These are structurally equivalent but not name equivalent. Depending on your language, the following might also be structurally equivalent to the above:

```
struct part {
   string manufacturer;
   string description;
}
```

Depending on your language, the following might or might not be name equivalent:

```
type fahrenheit = integer;
type celsius = integer;
```

We probably don't want those to be, but maybe `integer` and `score` should be equivalent -- the word "score" might just be for documentation purposes.

This is ***strict*** v. ***loose*** name equivalence. Ada lets you choose:

```
type score is integer;

type fahrenheit is new integer;
type celsius is new integer;
```

Algol-68 used structural equivalence, as did many early Pascal implementations (the ISO standard uses name equivalence). Java uses name equivalence. ML-family languages are more-or-less structural (see below). C uses a hybrid (structural except, ironically, for structs).

Both forms of type equivalence have nontrivial implementation issues for separate compilation.
  timestamp header files?
  checksum header files?
    avoid comments, format?
    how handle compatible upgrades?
  finer grain?
"name mangling" -- enforce with standard linker

Coercion
  When an expression of one type is used in a context where a different type is expected, one normally gets a type error. But what about

```
var a : integer; b, c : real;
...
c := a + b;
```

  Many languages allow things like this, and ***coerce*** an expression to be of the proper type. Coercion can be based just on types of operands, or can take into account expected type from surrounding context as well.

  Fortran and C have lots of coercion, all based on operand type.
  Here's an abbreviated version of the C rules:
    if either operand is `long double`, the other is converted if
      necessary, and the result is `long double`
    else similarly if either is `double`
    else similarly if either is `float`
    else both are integral:
      if they're the same, the result matches
      else if both are signed or both unsigned, the one with lower
        "rank" is converted to the one with higher rank, and the
        result matches
      else one is signed and the other is unsigned:
        if the unsigned has greater or equal rank, the signed
          one is converted and the result has the unsigned type
        else if the signed type can hold all possible values of
          the unsigned type, the unsigned one is converted and
          the result has the signed type
        else both are converted to the unsigned type corresponding
          to the signed type, and that's also the type of the result
  if necessary, precision is removed when assigning into LHS

  In effect, coercion is a relaxation of type checking.
  Some languages (e.g. Modula-2 and Ada) forbid it.

  C++, by contrast, goes hog-wild with coercion.
  It's one of several parts of the language that many programmers find difficult to understand.

Make sure you understand the difference between
  type ***conversions*** (explicit)
  type ***coercions*** (implicit)
  non-converting type ***casts*** (breaking the typing rules)

  Sometimes the word 'cast' is used for conversions,
  which is unfortunate. C is guilty here.

  Some authors also vary the meanings of "conversion" and "coercion" -- e.g., to distinguish between cases that do or do not entail run-time code. I think that's a bad idea: I use the terms to indicate semantics; implementation is orthogonal.

Type inference and polymorphism

simple case: local-only. Esp. useful for declarations.

```
var pi = 3.14;     // C#
auto pi = 3.14;    // C++11
```
or
```
auto o = new very_long_type_name<X, Y, Z>(args);
```

similarly
```
var/val/def  in Scala
var/let      in Swift
var/:=       in Go
```

complicated case: ML (OCaml), Miranda, Haskell

```
1  -- fib :: int -> int
2  let fib n =
3    let rec helper f1 f2 i =
4     if i = n then f2
5     else helper f2 (f1 + f2) (i + 1) in
6    helper 0 1 0;;
```

`i` is `int`, because it is added to 1 at line 5
n is `int`, because it is compared to `i` at line 4
all three args at line 6 are `int` consts, and that's the only use of
  `helper` (given scope of `let`), so `f1` and `f2` are `int`
also, the 3rd argument is consistent with the known `int` type of `i` (good!)
and the types of the arguments to the recursive call at line 5 are
  similarly consistent
since `helper` returns `f2` (known to be `int`) at line 4, the result of
  the call at line 6 will be `int`
Since `fib` immediately returns this result as its own result,
  the return type of `fib` is `int`

(Note that the limited scope of the `let` construct allows the compiler to use the types of `helper`'s actual parameters to deduce `helper`'s own types -- something it can't do at the global level.)

`fib` itself is of type `int -> int`
`helper` is of type `int -> int -> int -> int`

Polymorphism results when the compiler finds it doesn't need to know certain things. For example:

```
let compare x p q =
  if x = p then if x = q then "both" else "first"
  else if x = q then "second" else "neither";;
  (* NB: I've used structural equality comparison here,
       not physical identity *)
```

`compare` has type `'a -> 'a -> 'a -> string`
`'a` is a ***type variable***, so `compare` is polymorphic.

Any time the ML or Haskell compiler determines that A and B have to have the same type, it tries to ***unify*** them. For example, in the expression

```
if x then e1 else e2
```

x has to be of type `bool`, and e1 and e2 have to be of the same type.
If e1 is (so far) known to be of type `'a * int` (a 2-element tuple)
and e2 is known (so far) to be of type `char list * 'b`, then `'a` is `char list`
and `'b` is `int`, and the expression as a whole is of type `char list * int`.

Like Lisp, ML-family languages make heavy use of lists, but ML's lists are homogeneous -- all elements have to have the same type. Ex:

```
let append l1 l2 =
    match l1 with
    [] -> l2
    h::t -> h :: append t l2;;
```

`::` is a ***constructor*** -- used for piecing together values of composite types (like `cons` in Lisp).
There are many other such polymorphic functions.
Note that `hd` and `tl` in ML (like `car` and `cdr` in Lisp) are bad style; you should almost always use `match`, as in the example above.

Unification, by the way, is a powerful technique, used for a variety of purposes in programming languages. It's the basis of computation in Prolog, which tries to unify RHS's of rules with LHS's of things that might imply them.
  In Prolog, unification assigns values to variables
  In ML, it assigns types to type variables

Unification is also used to type-check C++ templates.

Advanced Topics

Types can be the subject of a whole class on their own.
A certain amount of advanced material gets covered in 255.

Here are a couple examples.

Type classes and ***Higher-level types (kinds)***

  Type classes are sort of like interfaces in an object-oriented language, but built into the compiler.

  In Haskell, for example, a type that supports equality and inequality operators (`==` and `/=`) is of ***class*** `Eq`.
  A type that supports `<`, `>`, `<=`, and `>=` is of class `Ord`.
  `Ord` is a subclass of `Eq`: you can use an `Ord` type anywhere an `Eq` type is expected.

  Like OCaml, Haskell provides ML-family type inference. But where OCaml defines ordering operations on every type for simplicity, Haskell ***infers*** that any values to which you apply < or > operators must be of a type in class `Ord`.

  You can define your own type classes.

  There is also a notion of type ***kinds***, which impose structure on type constructors like tuples, records, variants, and functions.

Typestate

  A few languages capture, in the compiler, the notion that objects of a class can be in any of several ***states***, that certain methods apply only when the object is in a certain state, and the certain methods ***transform*** the object from one state to another. This allows certain kinds of errors to be caught by the compiler.

  An object of type `file`, for example, might only support `read` and `write` operations after it has been opened. A typestate compiler might catch "file has not been opened" errors at compile time.

  You can think of ***definite assignment*** in Java and C# as a very limited form of typestate.

Lifetime analysis

  Rust incorporates the notion of ***lifetime*** into types, to avoid dangling references and storage leaks without run-time garbage collection.

  By default, a dynamically allocated mutable (non-constant) object in Rust can be accessed through only one variable at a time. When desired, the programmer can create multiple read-only references to a variable. The compiler can always tell when the last reference to a variable goes away, and can generate code to reclaim its space.

  The rules are complicated, however, and have not yet been successfully formalized. Moreover many standard container classes have to break the rules in order to produce code that is both fast and fully functional. One has to trust that this "unsafe" code is correct -- or perhaps some day prove it.