Notes for CSC 254, 1 and 6 Nov. 2023

==========

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.
readability
performance optimization

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 or most of 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.

To a large (but not exclusive!) extent, static checking adds complexity to the programmer's task in return for better performance and earlier error detection.

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 type equivalence, subtyping, and type compatibility

Compatibility is the practical concept: it tells you what you can ***do***.
Most languages say type A is compatible with (can be used in a context that expects) type B if A is equivalent to B, A is a subtype of B, or if A can be ***coerced*** to B.

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 if we declare

```
type score = integer;
```

then 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`

Side note: named types in OCaml introduce new types, even if their internal structure matches. So the language is *mostly* structural but with a bit of name equivalence as well. If two types have constructors with the same name, the inference engine can get confused (in OCaml, in the case of ambiguity, it arbitrarily guesses the most recently declared type with a matching constructor). You can resolve the confusion/ambiguity with explicit type declarations (e.g., `let foo:t = Bar a b c;;`)

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. Ownership in Rust is a more complicated subcase.

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.

==========

Polymorphism and Generics

Recall from chapter 3:

ad hoc polymorphism: fancy name for overloading

subtype polymorphism in OO languages allows code to do the "right thing" when a ref of parent type refers to an object of child type

implemented with vtables (to be discussed in chapter 10)

parametric polymorphism

type is a parameter of the code, implicitly or explicitly

implicit (true)

language implementation figures out what code requires of object

at compile-time, as in ML or Haskell

at run-time, as in Lisp, Smalltalk, or Python

lets you apply operation to object only if object has everything the code requires

explicit (***generics***)

programmer specifies the type parameters explcitly

mostly what I want to talk about today

Generics are found in Clu, Ada, Modula-3, C++, Eiffel, Java 5, C# 2.0, ...

C++ calls its generics ***templates***.

They allow you, for example, to create a single stack abstraction, and instantiate it for stacks of integers, stacks of strings, stacks of employee records, ...

```
template <class T>
class stack {
   T[100] contents;
   int tos = 0;    // first unused location
public:
   T pop();
```

```
   void push(T);
   ...
}
...
stack<double> S;
```

I could, of course, do

```
class stack {
   void*[100] contents;
   int tos = 0;
public:
   void* pop();
   void push(void *);
   ...
}
```

But then I have to use type casts all over the place. Inconvenient and, in C++, unsafe.

Lots of containers (stacks, queues, sets, strings, mappings, ...) in the C++ standard library.

Similarly rich libraries exist for Java and C#. (Also for Python and Ruby, but those use implicit parametric polymorphism with run-time checking.)

Some languages (e.g. Ada and C++) allow things other than types to be passed as template arguments:

```
template <class T, int N>
class stack {
   T[N] contents;
   int tos = 0;
public:
   T pop();
   void push(T);
   ...
}
...
stack<double, 100> S;
```

Implementation

C# generics do run-time instantiation (***reification***). When you say `stack<foo>`, the run-time system invokes the JIT compiler and generates the appropriate code. Doesn't box native types if it doesn't need to—more efficient.

Java doesn't do run-time instantiation. Internally everything is `stack<Object>`. You avoid the casts in the source code, but you have to pay for boxing of native types. And since the designers were unwilling (for backward compatibility reasons) to modify the VM, you're stuck with the casts in the generated code (automatically inserted by the compiler) -- even though the compiler knows they're going to succeed—because the JVM won't accept the byte code otherwise: it will think it's unsafe. Also, because everything is an `Object` internally, reflection doesn't work.

The Java implementation strategy is known as ***erasure*** -- the type parameters are simply erased by the compiler. One more disadvantage: you can't say `new T()`, where T is generic parameter, because Java doesn't know what to create.

C++ does compile-time instantiation; more below.
C# may do compile-time instantiation when it can, as an optimization.

Constraints

The problem:

> If I'm writing a sorting routine, how do I insist that the elements in the to-be-sorted list support a `less_than()` method or < operator?
>
> If I'm writing a hash table, how do I insist that the keys support a `hash()` method?
>
> If I'm writing output formatting code, how do I insist that objects support a `to_string()` method?

Related question:

> Do I (can I) type-check the generic code, independent of any particular instantiation, or do I type-check the instantiations individually?

Tradeoffs are nicely illustrated by comparing Java, C#, and C++.

C++ is very flexible: every instantiation is individually type-checked. Constraints are implicit: if we try to instantiate a template for a type that doesn't support needed operations, the instance won't type-check. This has led, historically, to ***really*** messy error messages. Also introduces complications for separate compilation: suppose the code is in module A but the use that requires instantiation is in module B? In practice, most templates are put in header files, and the implementation elides duplicate instantiations at link time. It's also possible to manually instantiate particular versions in the module that has the code, but that's arguably a violation of abstraction.

Most other languages type-check the generic itself, so you don't get any instantiation-specific error messages. To support this, they require that the operations supported by generic parameter types be explicitly listed. Java and C# leverage interfaces for this purpose. C++ has used "named requirements" in the standard library for a while now; C++20 adds ***concepts***, which provide a more general (but optional, for backward compatibility) superset of the functionality found in Java and C#. They mostly (but not completely) supersede named requirements.

Java example:

```
public static <T implements Comparable<T>> void sort(T A[]) {
   ...
   if (A[i].compareTo(A[j]) >= 0) ...
   ...
}
...
Integer[] myArray = new Integer[50];
...
sort(myArray);
```

Note that Java puts the type parameters right in front of the return type of the function, rather than in a preceding "template" clause. `Comparable` is a standard library interface that includes the `compareTo()` method. Wrapper class `Integer` implements `Comparable<Integer>`.

C# syntax is similar:

```
static void sort<T>(T[] A) where T : IComparable {
   ...
   if (A[i].compareTo(A[j]) >= 0) ...
   ...
}
...
int[] myArray = new int[50];
...
sort(myArray);
```

C# puts the type parameter between the function name and the parameter list and the constraints after the parameter list. Java won't let you use `int` as a generic parameter, but C# is happy to; it creates a custom version of sort for `int`s.

(pre-20) C++ doesn't require that constraints be explicit.

```
template <class T>
void sort(T A[], int A_size) {...
```

(C++ can't figure out the size of an array, so you have to pass it in.
Alternatively you could make it another generic parameter.)

As noted in the book, bad things happen if a parameter "accidentally" supports a needed operation, in the "wrong way". If we instantiate sort on an array of C strings (`char*` s), for example, we get sorting by location in memory, not lexicographic order (C++ string objects compare lexicographically).

Constraints have historically been specified explicitly in C++ only by convention:

- make a function parameter inherit from a standard base class
  - e.g., `sort<foo>(SortableVector<foo> A)`, where `SortableVector<T>` inherits from both `Vector<T>` and `Comparator<T>`
- provide required operations as generic parameters
  - e.g. `sort<foo, comparator<foo>>(foo* A, int len)`
- provide required operations as ordinary parameters
  - e.g. `sort<foo>(foo* A, int len, bool (*less_than<foo>)())`
- make sort the `operator()` of a class for which `less_than()` is a constructor argument
  - e.g. `sort = new Sorter(bool (*less_than<foo>)())`,
  - where `Sorter` has an `operator()`

Concepts change this:

```
template <typename T>
concept Comparable = requires(T a, T b) { a < b; };

template <Comparable T>
void sort(T A[], int A_size) {...
```

Implicit Instantiation

Several languages, including C++, Java, and C#, will instantiate generic functions (not classes) as you need them, using roughly the same resolution mechanism used for overloading. (Actually, in C++ it requires unification, because of the generality of generic parameters, including nested templates and specialization.)

Interaction with Subtype Polymorphism

These two play nicely together. If I derive queue from `list`, I want subclasses. But I may also want generics: derive queue<T> from `list`<T>.

The subtle part is ***conformance*** of argument and return types. Suppose I want to be able to sort things in Java that don't implement `Comparable` themselves. I could make the comparator be a constructor argument instead of a generic argument (the 4th by-convention option in C++ above):

```
interface Comparator<T> {
   public Boolean ordered(T a, T b);
}

class Sorter<T> {
   Comparator<T> comp;

   public Sorter(Comparator<T> c) {   // constructor
      comp = c;
   }

   public void sort(T A[]) {
      ...
      if (comp.ordered(A[i], A[j])) ...
      ...
   }
```

```
}

class IntComp implements Comparator<Integer> {
   public Boolean ordered(Integer a, Integer b) {
      return a < b;
   }
}

Sorter<Integer> s = new Sorter<Integer>(new IntComp());
s.sort(myArray);
```

This works fine, but it breaks if I try

```
class ObjComp implements Comparator<Object> {
   public Boolean ordered(Object a, Object b) {
      return a.toString().compareTo(b.toString()) < 0;
   }
}
Sorter<Integer> s = new Sorter<Integer>(new ObjComp());
s.sort(myArray);
```

The call to new causes the compiler to generate a type clash message, because we're passing a `Comparator<Object>` rather than a `Comparator<Integer>`. This is fixed in Java using ***type wildcards***:

```
class Sorter<T> {
   Comparator<? super T> comp;
   public Sorter(Comparator<? super T> c) {
      comp = c;
   }
```

In general, you use `<? super T>` when you expect to pass objects into a method that might be willing to take something more general (and you never expect to accept such objects in return).

There's also `<? extends T>` syntax, for when you expect something back out of a context that might in fact give you something more specific (and you never expect to pass such objects in).

In effect, these `super` and `extends` keywords serve to control type compatibility for generics. Given a generic Foo<T>, we must ask: if C is derived from P (and thus C can be used in any context that expects a P), can Foo<C> be used in any context that expects Foo<P>? If so, we say Foo<T> is ***covariant*** in T.

Covariance typically happens in the case where T objects are returned

from Foo methods, but never passed into them as parameters.

For example, I can probably pass a `Generator<C>` object to anybody who expects a `Generator<P>` object:

- They expect to use the generator to conjure up P objects.
- If the generator gives them a C instead, they're happy.

Conversely (and more commonly), there are times when a `Foo<P>` object can be used in a context that expects `Foo<C>`.
When this happens, we say `Foo<T>` is ***contravariant*** in T.

Contravariance typically happens when T objects are passed to Foo methods, but never returned from them.

For example, I can probably pass a `Printer<P>` object to anybody who expects a `Printer<C>`.

- They're only going to give it C objects.
- Since the printer is willing to take a P object, it's happy.

C# makes covariance and contravariance a property of the generic itself. You specify `<in T>` and `<out T>` in the declaration of the generic to indicate contravariance and covariance, respectively. This restricts what can be done with T inside the generic.

Java gives you more flexibility (and arguably more confusion) by allowing you to annotate ***uses*** of the generic -- in effect saying "I promise (and the compiler should verify) that I never call methods that pass objects the wrong way)."

If you're always going to pass objects in, you're using contravariance, and you say `<? super T>`

If you're always going to accept objects as return values, you're using covariance, and you say `<? extends T>`.

covariance:

C is a P --> `Foo<C>` is a `Foo<P>` C# `out` Java `extends`

contravariance:

C is a P --> `Foo<P>` is a `Foo<C>` C# `in` Java `super`

invariance:

C is a P but `Foo<C>` and `Foo<P>` are incomparable

More on this on the PLP companion, section 7.3.2.
The Wikipedia page on covariance & contravariance has even more detail.