Topics:

- Static vs. dynamic types
- Strong typing
- Inference rules

In this lecture we start looking at types and type systems, which specify the most important kind of semantic analysis performed by compilers.

1 Strong vs. static typing

The C programming language has an expression `*p` whose meaning is to dereference the pointer `p`. However, C gives a lot of flexibility for computing the pointer `p`. In fact, it's possible to construct an *invalid* pointer, and the C compiler will not necessarily stop us or even give a warning. What does the ANSI C specification say happens in this case? In the specification [1], we read the following:

> *If an invalid value has been assigned to the pointer [p], the behavior of the unary operator * is undefined.*

This is actually only one of 88 occurrences of the phrase "the behavior ... is undefined" in the specification. What does it mean? It means that the computer is allowed to do anything it feels like, including bad things like erasing data you care about or sending illegal commands to attached hardware. On desktop computers, it often means in practice that your program crashes, but it also might simply cause the program to run incorrectly in an unpredictable way.

A program is *safe* only it never gets into a state where its behavior is undefined. Languages use *types* to help avoid undefined behavior, and the effectiveness of types at this job has been a real success story of computer science. Types help guarantee that various bad things don't happen, such as:

- operators (such as *) receive the right number and kind of operands.
- user-defined functions received the expected arguments too.
- variables are assigned the expected kinds of values.
- variables are initialized with values before they are used
- private module components are accessed only from within the module (Java security depends crucially on this!)

A *strongly typed* or *type-safe* language is once where undefined behavior can never happen. (This does not rule out *nondeterminism* in which more than one behavior is possible from a specified set.) Clearly, C is not an example of a strongly typed language.

A *statically typed* language is one in which the compiler associates types with program expressions (specifically, terms). C is an example of a statically-typed language. A statically typed language has a *sound type system* if it is also strongly typed. The compile-time type checks ensure safe behavior at run time.

In a *dynamically typed* language, values carry around a representation of their type at run time. Languages like Perl and Python are dynamically typed but not statically typed. The Java language has some support for dynamic typing, allowing it to safely handle type casts and `instanceof` tests.

We can classify languages according to whether they are strongly typed or statically typed:

[1] This is from the C89 spec

	Strongly typed	Not strongly typed
Statically typed	OCaml, Java	Pascal, C, C++, Xi
Not statically typed	Scheme, Python	FORTH, assembly

Given that we can have type safety without static typing, why do we want static typing?

- It helps prevent programmer errors.
- It allows the compiler to avoid run-time overhead in generated code.

Static typing does have some downside: it adds an annotation burden for programmers, it may make certain things programmers want to do impossible. Type systems that don't get in programmers' way tend to be complex and to require a complex, possibly slow type-checking algorithm. And in general, static type system cannot guarantee the absence of every possible kind of run-time error. Errors such as division by zero are hard to check statically, and some run-time checks are unavoidable.

2 Types

Most program expressions denote *types* or *terms*. Terms are ordinary expressions that represent a computation to be carried out at run time, whereas computation associated with types is carried out at compile time, except when dynamic typing features are used.

Types are a compile-time abstraction of run-time program behavior. In a sound type system, this abstraction is conservative, so good types rule out bad program behavior. For example, the type `int` usually indicates that a term with that type computes some integer (or possibly fails to terminate).

Type systems have two components: a set of possible type expressions, and a set of rules for associating terms with types. The set of possible type expressions is generally defined by a grammar, though in some type systems it is an involved process even to define the well-formed types. For a language like Xi, types are easy to define. We use the metavariable t here to denote a type[2].

$$t ::= \texttt{int} \mid \texttt{bool} \mid t[] \mid (t_1, \ldots, t_n)^{\ (n \geq 2)}$$

Here you can see two *base types* (also known as *primitive* or *ground* types), `int` and `bool`, and array and tuple types, which result from the application of a *type constructor*. The grammar for tuple types includes a side condition that the number of elements in the tuple must be at least two.

3 Names and type equality

In many languages, programmers can use names of their choosing to refer to types, so the grammar of types is extended accordingly. We use the metavariable X to refer to a type name.

$$t := \ldots \mid X$$

For example, in OCaml or C we can define another name for an existing type, for example making `num` an alias for the type `int`:

```
type num = int (* OCaml *)
typedef int num; /* C */
```

Here the type expressions `num` and `int` are really the same type, and the compiler needs to be able to do the compile-time computation to decide this equality. The computation involves figuring the true definition of `num` and `int`, by determining that they have the same structure. If we had an OCaml function that expected a `num*num` and we provided it an `int*int`, it would type-check because the types `num*num` and `int*int` are equal. This is an example of a *structural* equality test.

Another example of a named type is a class type, as in Java. Here we declare two classes `C` and `D`, so `C` and `D` can be used as types of objects:

[2]The Greek letter τ (tau) is another popular choice

```
class C { int x; int m(); }
class D { int x; int m(); }
```

In this example, we are binding the names `C` and `D` to types that are structurally identical, but the types are considered unequal simply because they have different names. Therefore type equality in Java is *nominal*, i.e., based purely on names rather than on structure of types.

4 Common type constructors

Different languages have many different kinds of type constructors. However, there are some common underlying patterns, and complex types such as class types can be understood to some extent as combinations of simpler type constructors. In the following we use fairly standard notations for the different kinds of types, though different languages may choose other notations.

4.1 Unit

Another handy kind of primitive type is a type with only a single value, called a *unit* type. It's a little bit like a defective boolean that instead of having `true` and `false`, has only one value. This might sound useless, but the unit type is useful for giving a type to expressions that do not produce an interesting value, such as statements, or procedures that do not return a value. In OCaml this type is written `unit` and has the single value `()`. In type theory, this type is often written as `1`. In C and Java the type `void` is essentially the unit type, and is used only to describe the return type of methods that don't return a value.

4.2 Tuples

A primitive kind of data structure is a *tuple*, which groups together several values of possibly different types. A common notation for a tuple containing values of types $t_1, t_2, \ldots, t_n$ is $t_1 * t_2 * \ldots * t_n$. Values within a tuple are distinguished by their position in the tuple. OCaml is an example of a language with tuples.

4.3 Records

Records are closely related to tuples. The type $\{x_1 : t_1, \ldots, x_n : t_n\}$ is similar to the tuple type $t_1 * \ldots * t_n$, except that the values in the record are accessed by field name x_i rather than by position. For example, accessing the field x_i of a value v might be written as $v\,.\,x_i$.

In some languages the order of the fields in the record is significant; in others, two record types are equal even if their field names appear in a different order. Making field order insignificant has implications if we want to support subtyping, which we will talk about in the next lecture.

4.4 Variants

Records store several named values. Sometimes we want a variable to be able to contain one of several different kinds of values. Variant types are one way to accomplish this. For example, in OCaml if we want a value that can be either an `int` or a `bool`, we can define a named type to accomplish this:

```
type int_or_bool = Integer of int | Boolean of bool
```

Given a value of type `int_or_bool`, we find out which we have by pattern matching:

```
let x: int_or_bool = ... in
match x with
    Integer(i:int) → ... (* use i *)
  | Bool(b:bool) → ... (* use b *)
```

4.5 Sums

The different cases of a variant are distinguished by their names. A more primitive variant-like type is a *sum* type, in which the different cases are distinguished by the order in which they occur. Thus, sums are to variants as tuples are to records. For example, the type `int+bool` represents a value that can be either an `int` or a `bool`. Sum types are useful for studying the theory of types, but real languages usually don't use them because it's too error-prone to distinguish cases by their position.

4.6 Function types

The type $t_1 \to t_2$ describes a function that takes in a value of type t_1 and returns a value of type t_2. To describe functions that expect multiple arguments, we can make t_1 a tuple type.

In languages like OCaml and C, function types are part of the language, because functions are first-class values. In Java, there are no values with function type, because functions occur only as methods inside objects. However, function types are still useful as a way of thinking about the types of methods.

In some languages, the exceptions that can be raised by a function are also part of the function type. Code that uses the function may be required by semantic analysis to handle all exceptions that might be raised.

4.7 Array types

Most languages offer some kind of type constructor for mutable arrays. In Java and C we have the type $t[\,]$; in OCaml we have `T array`. In some languages the array type includes information about the size of the array or the bounds on indices. For example, in the language Pascal, the type `array[1..10] of integer` represents an array of ten integers with 1 as the lowest legal index. In the Cyclone language, a type-safe language based on C, array types may have lengths specified in terms of variables. For example, the declaration `int f(int n, int[n] a)` declares a function `f` with two parameters, where the first (`n`) specifies the length of the second, array argument (`a`). Any attempt within the function to access the elements of `a` must be in a context where the compiler can prove that the index is in the range `0..n-1`.

The Cyclone array type is an example of a *dependent type*: a type that mentions terms. Dependent types are a powerful language feature, but raise a lot of tricky issues. In general they can lead to the compiler needing to prove complex theorems about possible computations. Most languages with some form of dependent types place restrictions on how they are used to avoid this.

4.8 Reference types

Many languages have some form of reference types. A reference type refers to or points to another value, and can be imperatively updated. The OCaml type t `ref` is an example of a reference type. In C we have the pointer type `t *`, which serves a similar function. Pointer types in C also support pointer arithmetic, which is unsafe.

4.9 Parameterized types

Modern languages support *parameterized types*, which are user-defined type constructors. In Java we can declare parameterized classes, e.g.:

```
class Cell⟨T⟩ {
    T contents;
}
```

Here, `Cell` is a type-level entity, but no value has the type `Cell`. Instead, `Cell` is really a function at the type level. If applied to any type `T`, it produces a class which looks like

```
class CellT {
    T contents;
}
```

We can therefore think of `Cell` as having a signature `type`→`type`[3], which is the same signature that we would ascribe to the type constructor for arrays. In OCaml we can similarly define our own type constructors, e.g.:

```
type 't cell = {contents: 't}
```

In either case, `Cell` or `cell`, the type constructor describes a computation that can be carried out entirely at compile time. Languages have been defined in which this type-level computation is Turing-complete, but usually we like to be confident that our compilers will eventually terminate, so this is quite unusual.

[3]Technically, we say that the *kind* of `Cell` is `type`→`type`.