COEN 175

Lecture 12: More Type Expressions

Review

- A **type expression** or **type signature** denotes the type of an expression.
- Any built-in or "atomic" type in the language is a legal type expression.
- If S and T are type expressions, then:
  - $S \rightarrow T$ denotes a mapping from type S to type T
  - $S \times T$ denotes a (Cartesian) product of type S and type T
  - pointer(T) denotes a pointer to type T
  - array(T, length) denotes an array of type T

Example: Addition

- What are the type expressions for + in Simple C?
  - `int` × `int` → `int`
  - `double` × `double` → `double`
  - pointer(α) × `int` → pointer(α)
  - `int` × pointer(α) → pointer(α)
- The first two expressions are for addition.
- The last two are for pointer arithmetic.
  - We have two expressions since addition is commutative.

Example: Subtraction

- What are the type expressions for - in Simple C?
  - `int` × `int` → `int`
  - `double` × `double` → `double`
  - pointer(α) × `int` → pointer(α)
  - pointer(α) × pointer(α) → `int`
- The first two expressions are for subtraction.
- The last two are for pointer arithmetic.
  - If adding an offset to a pointer yields a new pointer, then we should be able to subtract the two pointers to get the offset.

Example: Address

- What is the type expression for & in Simple C?
  - $\alpha \rightarrow \text{pointer}(\alpha)$
- Given an object of some type, the result is a pointer to that object.
- What was the type expression for dereference?
  - $\text{pointer}(\alpha) \rightarrow \alpha$
- We see that the address and dereference operators are **inverses**.
  - For example, `*&x` is the same as just writing `x`!

Example: Indexing

- What is the type expression for `[]` in Simple C?
  - pointer(α) $\times$ `int` $\rightarrow \alpha$
- Why do we use a pointer and not an array?
  - All arrays are promoted to pointers.
- The true semantics in C are more interesting.
  - E_1`[`E_2`]` is defined by the C standard as `*(`E_1 `+` E_2`)`.
  - By that definition, `a[i]` is equivalent to `*(a + i)`, which is equivalent to `*(i + a)`, which is equivalent to `i[a]`!

Example: Logical Or

- What are the type expressions for `||` in Simple C?
  - `int` × `int` → `int`
  - `int` × `double` → `int`
  - `int` × pointer(α) → `int`
  - `double` × `int` → `int`
  - `double` × `double` → `int`
  - `double` × pointer(α) → `int`
  - pointer(α) × `int` → `int`
  - pointer(α) × `double` → `int`
  - pointer(α) × pointer(β) → `int`
- Why so many? Short-circuit evaluation.

Are These Errors?

- Assume that `x` is declared as type `int`.
- The statement `x = 1` is certainly legal, as `1` also has type `int`.
- Is the statement `1 = x` legal? Why or why not?
- Assume that `p` is declared as a pointer to an `int`.
- The statement `p = &x` is again certainly legal.
- Is the statement `&x = p` legal? Why or why not?

Lvalues vs. Rvalues

- The statement `x = y` says to take the value of `y` and place it into the location denoted by `x`.
- The problem we had earlier is that both `1` and `&x` do not have locations.
- An expression that denotes a location is an **lvalue**.
  - It is so called because it can be used on the left-hand side of an assignment statement.
- An expression that only denotes a value is an **rvalue**.
  - Both `1` and `&x` are not lvalues; they are rvalues.

Lvalues

- Not every identifier denotes an lvalue.
  - Scalar variables are lvalues.
  - Functions and arrays are **not** lvalues (in Simple C).
  - You could consider them to be **constant** lvalues in C.
- Most expressions do not yield lvalues, but some do.
  - A dereference does: `*p = 1`.
  - An array index does: `a[i] = 1`.
- Some expressions require lvalues.
  - Assignment does: `x = 1` is legal, but `1 = x` is not.
  - Address does: `p = &x` is legal, but `p = &1` is not.

Imperative Languages

- Formally including lvalues in our specification would require some new notation.
- In an imperative language such as C:
  - Names are bound to declarations;
  - Declarations are mapped to locations;
  - Locations store values.
- In functional languages, declarations are mapped directly to values.
- A graduate level class in formal semantics would cover these topics in a lot more depth.

Summary

- A type system is a set of rules that assign types to expressions.
- Type expressions or signatures are a formal and compact way of specifying those rules.
- C is a statically typed language, albeit weakly typed.
  - C++ is somewhat stronger, but still not much.
  - Ada is a strongly, statically typed language.
- Overloading, polymorphism, coercion, and casting are all necessary evils of type systems.