CS 206

Lecture 15 – Types and Type Systems

Types in Programming Languages

- Discussion

What are Data Types?

- A (data) type is a homogeneous collection of values, effectively presented, equipped with a set of operations which manipulate these values
- Various perspectives:
  - collection of values from a "domain" (the denotational 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)

Why do you need types?

- Provide implicit context
  - a + b
  - Can be integer or floating point
- Help with program organization and documentation
  - Separate types for separate concepts
    - Addition for numbers, concatenation for characters etc.
  - Document intended use of declared identifiers
    - Types can be checked, unlike program comments
- Identify and prevent errors
  - Compile-time or run-time checking can prevent meaningless computations such as 3 + True
- Support implementation and optimization
  - Example: short integers require fewer bits – Access components of structures by known offset

A Type System

- A type system consists of
  - The set of predefined types of the language.
    - Int, float, bool etc.
  - The mechanisms which permit the definition of new types
  - The mechanisms for the control of types, which include:
    - **Equivalence rules** which specify when two formally different types correspond to the same type.
      - when are the types of two values the same?
    - **Compatibility rules** specifying when a value of a one type can be used in a context in which a different type would be required.
    - Rules and techniques for **type inference** which specify how the language assigns a type to a complex expression based on information about its components.
  - The specification as to whether (or which) constraints are statically or dynamically checked.

Type Safety

- A language is type safe when no program can violate the distinctions between types defined in its type system
- In other words, a type system is safe when no program, during its execution, can generate an unsignalled type error
- Also: if code accesses data, it is handled with the type associated with the creation and previous manipulation of that data

Languages : Safe and not Safe

- Not safe: C and C++
  - Casts, pointer arithmetic
- Almost safe: Algol family, Pascal, Ada
  - Dangling pointers
  - Allocate a pointer p to an integer, deallocate the memory referenced by p, then later use the value pointed to by p.
  - ***No language with explicit deallocation of memory is fully typesafe***.
- Safe or Strongly Typed: Lisp, Smalltalk, ML, Haskell, Java, JavaScript
  - Dynamically typed: Lisp, Smalltalk, JavaScript
  - Statically typed: ML, Haskell, Java

Type Checking

- Before any operation is performed, its operands must be type-checked to prevent a type error. Example-
- mod operation:
  - check that both operands are integers
- and operation:
  - check that both operands are booleans
- indexing operation (array)
  - check that the left operand is an array, and that the right operand is a value of the array’s index type

Static and Dynamic Typing

- In a statically typed language
  - all variables and expressions have fixed types (either stated by the programmer or inferred by the compiler)
  - all operands are type-checked at compile-time
- Most PLs are statically typed, including Ada, C, C++, Java, Haskell

Static and Dynamic Typing

- In a dynamically typed language
  - values have fixed types, but variables and expressions do not
  - operands must be type-checked when they are computed at run-time.

- Some PLs and many scripting languages are dynamically typed, including Smalltalk, Lisp, Prolog, Perl, Python.

Example: Ada static typing

- Ada function definition:

```
function is_even (n: Integer)
     return Boolean is
begin
   return (n mod 2 = 0);
end;
```

The compiler doesn't know the value of n. But, knowing that n's type is Integer, it infers that the type of "n **mod** 2 = 0" will be Boolean.

- Function Call:

```
p: Integer;
...
if is_even(p+1) ...
```

The compiler doesn't know the value of p. But, knowing that p's type is Integer, it infers that the type of "p+1" will be Integer.

- Even without knowing the values of variables and parameters, the Ada compiler can guarantee that no type errors will happen at run-time.

Example: Python Dynamic Typing

- Python function definition:

```
def even (n):
   return (n % 2 == 0)
```

The type of n is unknown. So the "%" (*mod*) operation must be protected by a run-time type check.

- The types of variables and parameters are not declared, and cannot be inferred by the Python compiler. So run-time type checks are needed to detect type errors

Static vs dynamic typing

- Pros and cons of static and dynamic typing:
- Static typing is more efficient. Dynamic typing requires runtime type checks (which make the program run slower), and forces all values to be tagged (to make the type checks possible). Static typing requires only compile-time type checks, and does not force values to be tagged.
- Static typing is more secure: the compiler can guarantee that the object program contains no type errors. Dynamic typing provides no such security.
- Dynamic typing is more flexible: This is needed by some applications where the types of the data are not known in advance.
  - JavaScript array: elements can have different types
  - Haskell list: all elements must have same type