

Structure of Programming Languages – Lecture 11

CS 636 / 536

June 22, 2009

- 1 Types
 - Primitive types

- 2 Strong Typing
 - Type Conversion
 - Type Identity and Type Checking
 - Type Coercion

Built-in Data Types

Languages, old and new, vary greatly in the set of primitive types that are supported. The issues are: variety, size, encoding, specificity, efficiency, precision, and semantics.

- Are booleans supported? Characters? If so, what kind? What about strings?
- Number vs. integer and floating point.
- One size of integer vs. many? Real vs. float, double, and long double?
- Unsigned, fixed point, rational, and complex?
- Hardware-based types vs. indefinite precision types.
- Read-only type modifiers?
- What type compatibilities and conversions are supported? Are they automatic?

What have you seen in languages you know?

Ancient History

The idea of type (an abstraction, separate from an object) developed slowly and late.

- Machine code: a type is a bit-level representation together with machine instructions that operate on it.
- Mathematics: A type is a set of objects.
- FORTRAN: integer and floating point.
- COBOL: Characters: DISPLAY
Numbers: COMPUTATIONAL (binary fixed point) , COMP-2 (packed bcd), COMP-3 (floating point)
- Simula (1967): a structural definition was grouped with operations on that structure.

Aggregate Types

Aggregate types were needed, with the ability to define new types.

- FORTRAN: Arrays, based on use of hardware index registers.
- COBOL: Tables (arrays) and records. An object could have named fields. To use a field, only the field name was needed.
- LISP: Atoms, cells and linked structures. Type predicates for atom and list.
- APL: Objects were typed, variables were not. Type tags were present at run time.
- PL/1 (1966): Records, as in COBOL, but “LIKE” was supported.

History

Type became a definable abstraction in the late 60's

- Strachey and Standish: A type is a set of constructors, selectors, and a predicate.
- Algol-68: type declarations and a type calculus (to deduce the result types of functions)
- C (1971): Used hardware-based types but added structure definitions. Mapped multiple types with distinct semantics onto the same representation. No checking of parameter types of user-defined functions.
- Pascal (1972): Keep it simple; Implemented Algol-68 type system without the complex system of type conversions.
- Smalltalk(about 1981): Introduced classes.

Casts, Conversions, Coercions.

The C **type cast** hides a rat's nest of problems. C++ got it straight.

- Static cast (type conversion, changes size or encoding, keeps semantics.)
- Reinterpret cast (pointer cast, changes semantics, keeps representation.)
- Const cast (removes or adds a restriction to a pointer)
- Dynamic cast (movement up or down a type hierarchy. Downward movement is an run time operation and may fail.)

A **type coercion** is a cast that is applied by the compiler.

C++ allows the programmer to define casts and will use them for coercion.

Type Checking Supports Semantic Validity.

A language is **strongly typed** if the type of each actual argument in a function call must match the type of the corresponding formal parameter.

- Does each object belong to exactly one type?
- Is the type declared or is it deduced from the structure?
- Is type coercion allowed? Or must the match be exact?
- Does the type describe the representation or the semantics or both?
- Are the types checked before applying every function?
- Are they used to restrict or eliminate semantically meaningless operations?

Strong Typing and Stronger Typing.

Both Pascal and Ada supported strong typing, but Ada's rules were stronger. For example, both supported **variant record** types.

- A variant record represented two or more alternative structures.
- The alternatives had a common beginning, followed by a type tag, and different endings.
- Modern languages achieve the same thing with class derivation and polymorphism.
- In Pascal, the type tag and the following parts could be changed (assigned) independently.
- In Ada, the type tag and the parts it described could only be assigned as a unit.

Can the Type of an Object be Deduced from its Structure?

- The first “functional” language was LISP, which had two types: atom and list. The LISP system could tell whether an object was an atom or a list, and a programmer could test this property.
- The functional languages of the 90’s perpetuated this idea: types were deduced, not declared. Functions could have multiple methods, and a different method might be dispatched depending on the type of the object.
- If control over semantics was desired, the programmer could add a type indicator at the head of each object. (This is more work than declaring a type for a variable.)

Miranda	C or C++
b = [box, 36, 24, 36]	box b = {36, 24, 36}
b = [combo, 36, 44, 39]	combo c = {36, 44, 39}

Some coercions will create nonsense.

Some languages will **coerce** argument types when they do not match the corresponding parameter types.

- Coercions are supposed to be applied **ONLY** when they preserve semantics.
- Lengthening a short value to a longer type preserves semantics.
- Shortening might or might not preserve the semantics, so it should not be used freely.
- Some changes of representation preserve semantics (int to double).
- Some generally do not (double to short int).

Coercion is, therefore, a questionable practice.

PL/1 was liberal with coercion.

In PL/1, any argument would be coerced to any parameter type if the compiler could find a chain of conversions to get from one to the other. In the process, garbage often happened. Consider this PL/1 statement: `IF (a<=b<=c) THEN x = 1; ELSE t = 1`

- The result of `a<=b` is a single bit truth value, 1 or 0.
- The 1 or 0 is cast to the underlying type – bitstring, length 1.
- The bitstring is promoted to an integer to match type of `c`.
- The integer is lengthened to the length of `c`.
- Now it is compared to `c`, and the answer is always true if `c` is greater than 0!

Analysis: Casting to an underlying representation type removes the original semantics. Then casting to a higher-level represented type ADDS semantics, which may be inappropriate. The final step in this evaluation compared apples (painted orange) to oranges.

Ada threw it all out.

Semantic validity was a primary goal in the design of Ada.

- The coercion messes made by PL/1 were to be avoided.
- So all coercion was thrown out in the design of Ada.
- If you wanted to add 1 to PI, you could define a method for operator+ that converted the 1 to a floating representation, then did the addition and returned a real.
- Ada programmers learned to be careful of the types of their constants.

Analysis: The semantic nonsense problem was solved at the cost of convenience.

C implemented coercion between numeric types.

The idea in C is that numbers can retain all or most of their meaning when converted to a different representation.

- Lengthening a representation (`float --> double`) is “safe” and is used freely for coercion.
- Shortening a representation is not safe, and is only used if necessary to carry out an assignment or call-by-value. It should trigger a warning.
- Converting to a floating representation (`int --> double` or `float`) is usually “safe” and is used freely for coercion.
- Converting to an integer representation (`double` or `float --> int`) usually loses precision. It is considered “unsafe” and used only for assignment. It should trigger a warning.

Analysis: This set of conventions works and is convenient.

Warnings are never given if the cast is explicit.

C++ adds new coercions, under programmer control.

C++ adopted the type coercion rules of C and added the ability to define conversions for new types.

- A class constructor can be used to convert from any type to the class type.
- The cast operator can be used to convert from the class type to any other type.
- Programmer-defined casts will be used for coercion, if necessary.
- Overuse of this facility can make code very hard to penetrate.
- Casts up a class hierarchy (toward the base class) are used for coercion.
- Downward casts must be explicit and may bomb at runtime.

Analysis: C++ preserves object semantics by doing a run-time test for the validity of a downward cast on a derivation hierarchy.