Types I

Hwansoo Han
Data Types

- Intuitive notion of what types are:
  - Denotational point of view
    - A set of values from a "domain"
  - Constructive point of view
    - Either a small collection of built-in types (integer, character, boolean, real, etc.) or
    - A composite type created by constructor (record, array, set, etc.)
  - Abstraction-based point of view
    - Collection of well-defined operations that can be applied to objects of that type
What Are Types Good For?

- Provide implicit context
  - Make sure that certain meaningless operations do not occur
    - $a + b$ : use integer addition, if types of both are integer

- Limit the set of operations
  - Prevent programmers from using semantically invalid operations (e.g., add a character to a record)

- Type checking cannot prevent all meaningless operations
  - It catches enough of them to be useful
Type System

Type system consists of
1. A mechanism to define types and their language constructs
2. A set of rules for type equivalence, compatibility, inference

Notions in type system
- Type equivalence
  - When are the types of two values the same?
- Type compatibility
  - When can a value of type A be used in a context that expects type B?
- Type inference
  - What is the type of an expression, given the types of the operands?
Type Checking – Strong vs. Weak

- **Strong typing**
  - A popular buzz-word like *structured programming*
  - Informally, prevents you from applying an inappropriate operation to data
  - **Strongly typed languages**
    - Ada, Java

- **Weak typing**
  - **Weak typed languages**
    - C, C++
    - C89 is more strongly typed than its predecessor dialects, but less strongly typed than Pascal
Type Checking – Static vs. Dynamic

Static typing
- Means that all the type checking can be done at compile time
- Statically typed languages
  - e.g., Ada, Pascal
  - In practice, most type-checking can be done at compile time,
    - But not 100% can be done at compile time – needs dynamic checking
      (e.g., array index range check)

Dynamic typing
- Dynamic type checking
  - Lisp, Smalltalk, and most scripting languages (e.g., Python and Ruby) perform type checking at run time (but strongly typed)
  - Languages with dynamic scoping are generally dynamically typed
Classification of Types

- **Numeric types**
  - integer, real
    - multiple precision (bit length): short/int/long, single/double-precision
    - signed, unsigned, decimal (base-10), fixed-point (real number with two integers)

- **Enumeration types**
  - type weekday = {sun, mon, tue, wed, thu, fri, sat};

- **Subrange types**
  - type score = 0..100; workday = mon..fri;

- **Composite types (constructed types)**
  - Records (structures), variant records (unions), arrays, strings, sets, pointers (often implement recursive data types), lists, files

- **Discrete types – countable (integer, enumeration, subrange)**
Orthogonality in Type System Design

- A useful goal in the design of languages (and types)
  - A collection of features is orthogonal if there are no restrictions on the ways in which the features can be combined

- Example: vector type
  - C and Pascal are more orthogonal than Fortran in arrays
    - C and Pascal allows any type as an element
    - Fortran allows only scalar type as an element
Type Checking - Compatibility

- Types of an object in a context are constrained
  - An object can be used in a context, if its type is type-equivalent to the type that is expected in that context

- Type compatibility, a looser relation than equivalence
  - Type of an object and type of a context are compatible, even when their types are not equivalent, but allowed to be used

  **Type checking is to check the type compatibility**
Structural vs. Name Equivalence

Two major approaches in equivalence
- Structural type system – structural equivalence
- Nominative type system – name equivalence

Structural equivalence
- Based on implementation-oriented view (Algol-68, Modula-3, C)
  - Cannot distinguish two types that have the same structure by coincidence

Name equivalence
- Based on type names in declarations (Java, C#, Pascal, Ada)

```plaintext
struct XY {
  int x;
  int y;
}
 = ?

struct coordinate {
  int x;
  int y;
}
 = ?

struct reverseXY {
  int y;
  int x;
}
```
Type Conversion

- Static typing expects a specific type for many contexts
  - \( a = \text{expr} \) --- \( \text{expr} \) should be the same type of \( a \)
  - \( a + b \) --- + requires both \( a \) and \( b \) are integers or reals
  - \( \text{foo}(a, b) \) --- \( a \) and \( b \) should be the same types of \( \text{foo} \)'s formal parameters

- Cases in type conversion
  - Structurally equivalent
    - No conversion code is needed
  - Two types have common values (subrange, signed/unsigned)
    - Dynamic check is needed to avoid semantic errors
  - Structurally different
    - Convert may result in loss of precision, overflow
      - Conversions among int, unsigned, float, double
    - Many processors provide conversion instructions
Type 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

  ```c
  int a; float b, c;
  ...
  c = a + b;
  ```

- If languages allow different types than the expected one, implicit type conversion (called *type coercion* in this case) should take place.
Many languages coerce an expression to be of the proper type
  Coercion can be based just on types of operands, or expected type from surrounding context as well

Fortran
  All based on operand type

C
  all floats in expressions become doubles
  short int and char become int in expressions
  if necessary, precision is removed when assigning to l-value
Type Coercion (cont’d)

- Coercion rules are a relaxation of type checking
- Modern languages advocate static typing, away from type coercion
  - Modula-2 and Ada do not permit coercions
  - C++, however, provides extremely rich set of rules for type coercions
Type Conversion – Misc.

- Understand the differences
  - Type conversion -- general term for all
  - Type coercion -- implicit type conversion required by languages
  - Type cast -- used for explicit type conversion by programmers

- Non-converting type cast
  - No conversion code is used at all
  - The bits of underlying implementation of a type is interpreted as another type
Records and Variants

- Records (= structures)
  - Usually laid out contiguously
  - Possible holes for alignment reasons

- Smart compilers may rearrange fields to minimize holes
  - But C compilers promise not to rearrange
- Kernel programs sometimes assume a particular layout
  - To handle memory-mapped control registers for a device
Records and Variants

- Variant records (= unions)
  - Overlay space
  - Cause problems for type checking

- Main usage patterns
  - Same bytes interpreted in different ways at different times
  - Alternative sets of fields within a record
    - Common fields + various other fields
Structure Memory Layout

- Holes due to alignment
  - If not aligned, multiple instructions needed to read a field
  - For speed, aligning fields is needed

```
struct Atom {
  short  name;
  int    atomic_number;
  double atomic_weight;
  char   metallic;
}
```
Structure Memory Layout (cont’d)

- Packed record
  - Allows the compiler to optimize for the space
    - May require multiple instructions to read non-aligned field
- Reordered (sorted) fields
  - Can minimize the space due to holes
Summary

- Type checking
  - Strong vs. weak
  - Static vs. dynamic

- Type
  - Equivalence
  - Compatibility
  - Conversion
  - Coercion

- Structures and variants
  - Layout, alignment and holes