Type Systems
Composite Types

OPL – 2/66

Outline

- Data Type
- Type checking

Data Types

- Data types serve several important purposes:
  - provide implicit context for operations: In C, a+b
    - will use integer addition if a and b are of integer
    - will use floating-point addition if a and b are of floating-point
  - limit the set of operations.
    - prevent the programmer from adding a character and a record
    - good type systems catch enough mistakes to be highly valuable in practice
  - make the program easier to read and understand.
  - If types are known at compile time, they can be used to drive important performance optimizations.

Data Types

- Three of the most popular **meaning of “Type**” are:
  - **Denotational point of view:** a type is simply a set of value
  - **Structural point of view:** a type is either one of a small collection of built-in/primitive/predefined types (integer, character, Boolean, real, etc.), or a composite type (record, array, set, etc.)
  - **Abstraction-based point of view:** a type is an interface consisting of a set of operations with well-defined and mutually consistent semantics.

Type Systems

Common terms:

- discrete types – countable
  - integer
  - boolean
  - char
  - enumeration
  - subrange
- Scalar types - one-dimensional
  - discrete
  - real

Type Systems

Composite types:

- records (or structures)
- arrays
- strings
- sets
- pointers
- lists: usually defined recursively
- files

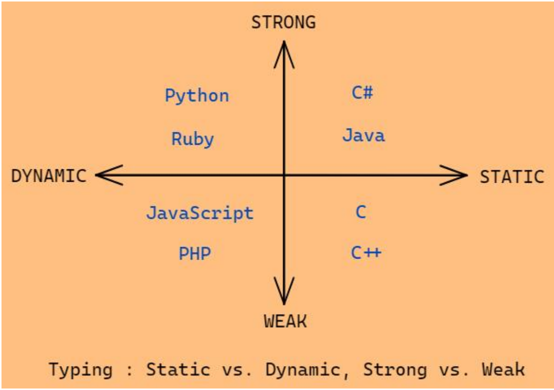
Type Systems

- Orthogonality
  - A collection of features is orthogonal if there are no restrictions on the ways in which the features can be combined
  - C++ overloaded << and >> operators are non-orthogonal: they can mean bit shifting or output/input depending on the context
- Orthogonality is nice primarily because it makes a language easy to understand, easy to use, and easy to reason about
- External link: https://en.wikipedia.org/wiki/Orthogonality_(programming)

Type Checking

- Type Checking
  - The process of ensuring that a program obeys the language's
  - A violation of the rules is known as a type clash
- Strongly typed
  - The language implementation can enforce, the application of any operation to any object that is not intended to support that operation.
- Statically typed
  - A language is said to be statically typed if it is strongly typed and type checking can be performed at compile time.
- Weakly typed
  - A weakly typed language has looser typing rules and may produce unpredictable or even erroneous results or may perform implicit type conversion at runtime

Type system Diagram



https://stackoverflow.com/users/9279260/debjbhatt

Type Checking

- Dynamic (run-time) type checking can be seen as a form of late binding, and tends to be found in languages that delay other issues until run time as well.
- Static typing -> intended for performance;
- dynamic typing -> intended for ease of programming.
- Lisp and Smalltalk are dynamically (though strongly) typed. Most scripting languages are also dynamically typed; some (e.g., Python and Ruby) are strongly typed.
- Languages with dynamic scoping are generally dynamically typed (or not typed at all): if the compiler can't identify the object to which a name refers, it usually can't determine the type of the object either

Type Checking

- **A type system has rules for**
  - **type equivalence** (when are the types of two values the same?)
  - **type compatibility** (is the one of most concern to programmers)
    - It determines when an object of a certain type can be used in a certain context.
    - the object can be used if its type and the type expected by the context are equivalent (i.e., the same).
  - **type inference** (what is the type of an expression, given the types of the operands?)

Objects → constant, variable, subroutine, etc.

Type Checking

- Type compatibility / type equivalence
  - –Compatibility is the more useful concept, because it tells you what you can DO
  - –The terms are often used interchangeably.
  - –If 2 types are equivalent, then they are pretty much automatically compatible, but compatible types do NOT need to be equivalent.

Type Checking

- Certainly format does not matter:

```
struct { int a, b; }
```

is the same as

```
struct {
     int a, b;
}
```

We also want them to be the same as

```
struct {
     int a;
     int b;
}
```


Structural Equivalence

- But should:

```
struct {
     int a;
     int b;
}
```

- Be the same as:

```
struct {
     int b;
     int a;
}
```

- Most languages say no, but some (like ML) say yes.

Type Checking

- Two major approaches: structural equivalence and name equivalence
  - Name equivalence is based on declarations
  - Structural equivalence is based on some notion of meaning behind those declarations
  - Name equivalence is more fashionable these days

Name Equivalence

- Should aliased types be considered the same?
- Example:

```
TYPE cel_temp = REAL;
     faren_temp = REAL;
 VAR c : cel_temp;
     f : faren_temp;
...
f := c;    (* should this raise an error? *)
```

- With strict name equivalence, the above raises an error. Loose name equivalence would allow it.

Structural Equivalence

- Structural equivalence depends on simple comparison of type descriptions substitute out all names
  - expand all the way to built-in types
- Original types are equivalent if the expanded type descriptions are the same

Type Casting

- Most programming languages allow some form of type conversion (or casting), where the programmer can change one type of variable to another.
- We saw a lot of this in Python:
  - Every print statement implicitly cast variables to be strings. We even coded this in our own classes.
  - Example from Point class:

```
def __init__(self):
    return ‘<‘ + str(self._x) + ‘,’ + str(self._y) + ‘>’
```


Type Casting

- 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

```
var a : integer; b, c : real;
      ...
c := a + b;
```


Type Casting

- Coercion
  - Many languages allow things like this, and COERCE an expression to be of the proper type
  - Coercion can be based just on types of operands, or can take into account expected type from surrounding context as well
  - Fortran has lots of coercion, all based on operand type

Type Coercion

- C has lots of coercion, too, but with simpler rules:
  - all `float`s in expressions become `double`s
  - short `int` and `char` become `int` in expressions
  - if necessary, precision is removed when assigning into LHS

Type Checking

- There are some hidden issues in type checking that we usually ignore.
- For example, consider x + y. If one of them is a float and one an int, what happens?
  - Cast to float. What does this mean for performance?
- What type of runtime checks need to be performed? Can they be done at compile time? Is precision lost?
- In some ways, there are good reasons to prohibit coercion, since it allows the programmer to completely ignore these issues.

Type Checking

- Make sure you understand the difference between
  - type conversions (explicit)
  - type coercions (implicit)
  - sometimes the word 'cast' is used for conversions (C is guilty here)

Records (or structures)

- Records
  - usually laid out contiguously
  - possible holes for alignment reasons
  - smart compilers may re-arrange fields to minimize holes (C compilers promise not to)
  - implementation problems are caused by records containing dynamic arrays

Records (or structures)

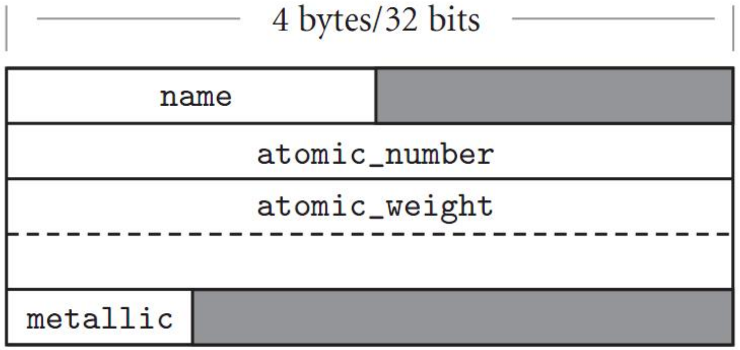


Figure 7.1 Likely layout in memory for objects of type element on a 32-bit machine. Alignment restrictions lead to the shaded "holes."

Records (Structures)

- Memory layout and its impact (structures)

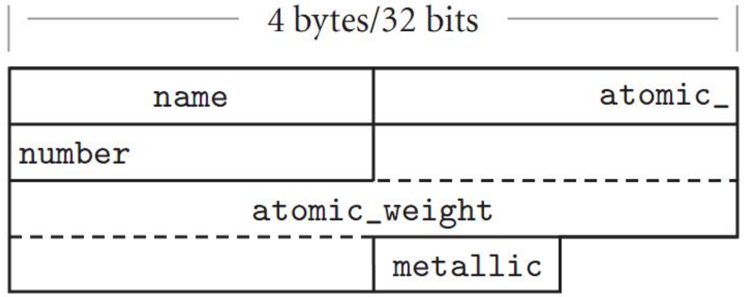


Figure 7.2 Likely memory layout for packed element records. The atomic_number and atomic_weight fields are nonaligned, and can only be read or written (on most machines) via multi-instruction sequences.

Records (Structures)

- Memory layout and its impact (structures)

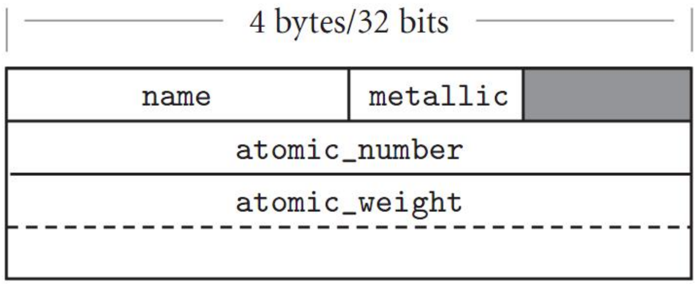


Figure 7.3 Rearranging record fields to minimize holes. By sor ting fields according to the size of their alignment constraint, a compiler can minimize the space devoted to holes, while keeping the fields aligned.

Unions (or variant records)

- Unions
  - overlay space
  - cause problems for type checking
- Lack of tag means you don't know what is there
- Ability to change tag and then access fields hardly better
  - can make fields "uninitialized" when tag is changed (requires extensive run-time support)
  - can require assignment of entire variant, as in Ada

Unions

• Memory layout and its impact

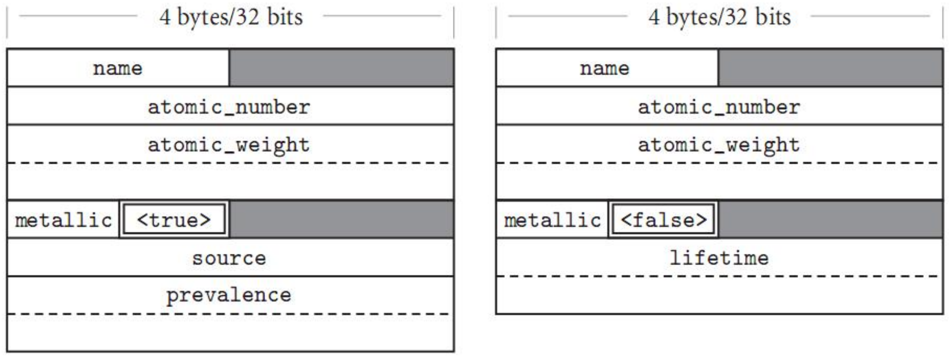


Figure 7.15 (CD) Likely memory layouts for element variants. The value of the naturally occurring field (shown here with a double border) determines which of the interpretations of the remaining space is valid. Type string_ptr is assumed to be represented by a (four-byte) pointer to dynamically allocated storage.

Unions

- C/C++ syntax:

```
union <name> {
  <datatype> <1st variable name>
   …
  <datatype> <nth variable name>
} <union variable name>
```

- Example:

```
union <name> {
  <datatype> <1st variable name>
   …
  <datatype> <nth variable name>
} <union variable name>
```


Unions

• In COBOL, 2 ways:

```
01 PERSON-REC.
   05  PERSON-NAME.
        10 PERSON-NAME-LAST PIC X(12).
        10 PERSON-NAME-FRST PIC X(16).
        10 PERSON-NAME-MID  PIC X.
   05  PERSON-ID            PIC 9(9) PACKED-DECIMAL.
01 PERSON-DATA              RENAMES PERSON-REC.

01 VERS-INFO.
   05  VERS-NUM             PIC S9(4) COMP.
   05  VERS-BYTES           PIC X(2)
                            REDEFINES VERS-NUM.
```


References

- Michael L. Scott, Programming Language Pragmatics FOURTH EDITION, Morgan Kaufmann, 2016.