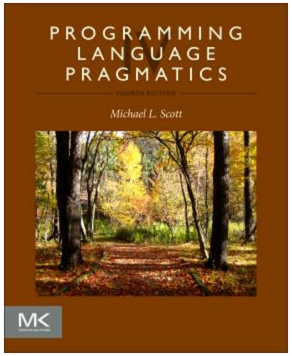
Chapter 7:: Data Types

Programming Language Pragmatics, Fourth Edition

Michael L. Scott



ELSEVIER

Data Types

- A ***data type*** defines a collection of data values and a set of predefined operations on those values.
- In the typical programming language, types serve two principal purposes:
  - Provide implicit context for many operations, freeing the programmer from the need to specify that context explicitly
  - Allow the compiler to catch a wide variety of common programming errors

Type Checking

- *Type checking* is the process of ensuring that a program obeys the language's type compatibility rules.
- STRONG TYPING means that the language prevents you from applying an operation to data on which it is not appropriate
- STATIC TYPING means that the compiler can do all the checking at compile time

Type Checking

- *Type checking* is the process of ensuring that a program obeys the language's type compatibility rules.
- STRONG TYPING means that the language prevents you from applying an operation to data on which it is not appropriate
- STATIC TYPING means strongly typed, and that the compiler can do all the checking at compile time

Type Systems

- Examples
  - Common Lisp is strongly typed, but not statically typed
  - Ada is statically typed
  - Pascal is almost statically typed
  - Java is strongly typed, with a non-trivial mix of things that can be checked statically and things that have to be checked dynamically
  - C has become more strongly typed with each new version, though loopholes still remain

Type Systems

- An Example:
  - Statically typed
    - int number;
  - Dynamically typed
    - number = 5;

Type Systems

- An Example:
  - Statically typed
    - int n;
  - Dynamically typed
    - n = 5;
    - More flexible but can lead to issues at runtime
      - nn = (n + 20) / 2

Type Systems

- Common terms:
  - Discrete types – countable – (ordinal types)
    - integer
    - boolean
    - char
    - enumeration
      - ex: `type weekday {sun, mon, tue, wed, thu, fri, sat};`
    - subrange
      - ex: `type test_score = 0..100;`
      - ex: `workday = mon..fri;`

Type Systems

- Common terms:
  - Scalar types – one-dimensional – (simple types)
    - discrete
    - rational
    - real

Type Systems

- Composite types – (nonscalar types) :
  - records (unions)
  - arrays
  - sets
  - pointers
  - lists
  - files

Type Checking

- A TYPE SYSTEM has rules for
  - 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

- Type compatibility / type equivalence
  - Compatibility is the more useful concept, because it tells you what you can DO
  - The terms are often (incorrectly, but we do it too) used interchangeably
  - Most languages say type A is compatible with (can be used in a context that expects) type B if it is equivalent or if it can be coerced to it

Type Checking

- Certainly, format does not matter:

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

is the same as

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

and we certainly want them to be the same as

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

Type Checking

- Two principal ways of defining type equivalence: *structural equivalence* and *name equivalence*
  - Name equivalence is based on declarations
  - Structural equivalence is based on the content of type definitions: roughly speaking, two types are the same if they consist of the same components, put together in the same way.
  - Name equivalence is more fashionable these days

Type Checking

- Structural equivalence depends on recursive 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
- Name equivalence depends on actual occurrences of declarations in the source code

Structural vs. Name Equivalence

```
struct person {
  string name;
  string address;
}


struct school {
  string name;
  string address;
}
```

Structural vs. Name Equivalence

```
1.  type student = record
2.          name, address : string
3.          age : integer

4.  type school = record
5.          name, address : string
6.          age : integer

7.  x : student;
8.  y : school;
9.  . . .
10. x := y;                 –– is this an error?
```

Name Equivalence: Aliases

- Depending on your language, the following might or might not be name equivalent:

  - `type fahrenheit = integer;`
  - `type celsius = integer;`

Types of Name Equivalence

- **Strict name equivalence**: aliases are distinct types
- **Loose name equivalence**: aliases are equivalent types

Name Equivalence: Aliases

- Modula-2 example:

```
TYPE celsius_temp = REAL;
TYPE fahren_temp = REAL;
VAR c : celsius_temp;
    f : fahren_temp;
...
f := c;
```

- Modula has loose name equivalence, so this is okay
  - But normally it probably should be an error

Type Checking

- Coercion
  - Implicit type conversion
  - 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 Checking

- 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

Coercion

```
type weekday = (sun, mon, tue, wed,
  thu, fri, sat);
subtype workday is weekday
  mon..fri;


d : weekday;
k : workday;
…
k := d;
d := k;
```

Type Checking

- C has lots of coercion, too, but with simpler rules:
  - all **`float`**s in expressions become **`double`**s
    - Float occupies 4 bytes in memory and has a precision of 7 digits.
    - Double occupies 8 bytes in memory and has a precision of 15 digits.
  - **`short`**, **`int`**, and **`char`** become **`int`** in expressions
  - if necessary, precision is removed when assigning into LHS

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)

ELSEVIER

Type Casting in C

- `r= (float) n;`
- `n= (int) r;`

Type Compatibility

- Generally two types are compatible if
  - they are equivalent,
  - one is a subtype of the other.

Summary

- Data types serve two principal purposes:
  - provide implicit context for many operations
    - freeing the programmer from the need to specify that context explicitly
  - allow the compiler to catch a wide variety of common programming errors

Summary

- In a typical pl, the *type system* consists of
  - a set of built-in types
  - a mechanism to define new types
  - rules for
    - *type equivalence*
    - *type compatibility*
    - *type inference*

Summary

- In a typical pl, the *type system* consists of
  - a set of built-in types
  - a mechanism to define new types
  - rules for
    - *type equivalence* determines when two values or named objects have the same type.
    - *type compatibility* determines when a value of one type may be used in a context that “expects” another type
    - *type inference* determines the type of an expression based on the types of its components or (sometimes) the surrounding context

Summary

- A language is said to be *strongly typed*
  - if it never allows an operation to be applied to an object that does not support it

- A language is said to be *statically typed*
  - if the compiler can do all the checking at compile time

Summary

- We introduced terminology for
  - the common built-in types
  - enumerations
  - subranges
  - the common type constructors (next week)

Summary

- In the area of type equivalence, we contrasted
  - *structural approach*
  - *name-based approach*

- We also examined
  - *type conversion*
  - *coercion*