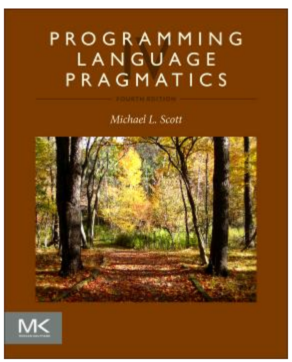
Chapter 7:: Data Types

Programming Language Pragmatics, Fourth Edition

Michael L. Scott



Data Types

- We all have developed an intuitive notion of what types are; what's behind the intuition?
  - 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)

ELSEVIER

Data Types

- What are types good for?
  - implicit context
  - checking - make sure that certain meaningless operations do not occur
    - type checking cannot prevent all meaningless operations
    - It catches enough of them to be useful
- Polymorphism results when the compiler finds that it doesn't need to know certain things

Data Types

- STRONG TYPING has become a popular buzz-word
  - like *structured programming*
  - informally, it 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

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

Type Systems

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

Type Systems

- ORTHOGONALITY is a useful goal in the design of a language, particularly its type system
  - A collection of features is orthogonal if there are no restrictions on the ways in which the features can be combined (analogy to vectors)

ELSEVIER

Type Systems

- For example
  - Pascal is more orthogonal than Fortran, (because it allows arrays of anything, for instance), but it does not permit variant records as arbitrary fields of other records (for instance)
- Orthogonality is nice primarily because it makes a language easy to understand, easy to use, and easy to reason about

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.

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 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 but can equate types the programmer doesn’t intend, considered lower-level
  - Name equivalence is more fashionable these days

Type Checking

- There are at least two common variants on name equivalence
  - The differences between all these approaches boils down to where you draw the line between important and unimportant differences between type descriptions
  - In all three schemes described in the book, we begin by putting every type description in a standard form that takes care of "obviously unimportant" distinctions like those above

Type Checking

- 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 Checking

- 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 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

Type Checking

- 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

- In effect, coercion rules are a relaxation of type checking
  - Recent thought is that this is probably a bad idea
  - Ada allows little conversion
  - Fortran and Pascal a little more
  - C++, Perl go hog-wild with coercions
  - They're one of the hardest parts of the language to understand

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)

Generic Subroutines and Modules

- Generic modules or classes are particularly valuable for creating *containers*: data abstractions that hold a collection of objects
- Generic subroutines (methods) are needed in generic modules (classes), and may also be useful in their own right