

The Joy of Types: Definitions

- A *type system* is a distinguishing characteristic of higher-level languages: from assembly on down, everything is bits
- *Type systems* consist of:
 - A mechanism for defining types and how they interact in a language
 - Rules for *type equivalence*, *type compatibility*, and *type inference*
- *Type equivalence*: when do values have the same type?
- *Type compatibility*: under what context(s) can values be used?
- *Type inference*: how does one determine the type of an expression or value?
- In languages with polymorphism, the type of an expression is not necessarily the type of the object/entity to which it refers
- Subroutines are a distinct type in languages where they are first- or second-class values

Type Checking

- *Type checking* is the process of ensuring that a program obeys the language's type compatibility rules
 - A *type clash* is a violation of type compatibility rules
 - A language is *strongly typed* if it prohibits, in a way that can be enforced, the application of operations on objects that are not meant to support them
 - A language is *statically typed* if it is strongly typed *and* type checking can be performed at compile time
 - Absolute 100% static typing is quite rare; typically, a “statically typed language” really means “statically typed most of the time”
 - Most of the commonly-known languages are mainly statically typed (Ada, Java, C)
- *Dynamic type checking* (a.k.a. *run-time type checking*) delays type checking until run-time: Lisp, Smalltalk, Perl, JavaScript
 - A form of late binding — types are not bound to objects until virtually the moment that types become relevant
 - Dynamic scope is associated with dynamic typing — after all, how can one check types statically if one doesn't know what an identifier refers to at compile time
 - Polymorphism does not necessarily imply dynamic type checking: Java, Eiffel

Defining Types

- Separate *declaration* from *definition*
 - *Declaration*: “There is a type called X.”
 - *Definition*: “The type X is...”
- Three perspectives on types
 - *Denotational*: types as sets of values
 - Types correspond to the mathematical notion of *domains*
 - *Constructive*: types as either a *built-in*, *simple*, or *primitive* type, or a *composite* of these types (possibly arbitrarily nested)
 - *Abstraction-based*: types as values over which specific sets of operations may be performed

Type Classification: “Types of Types”

- Close-to-the-hardware types (a.k.a. primitive, simple, built-in)
 - *Booleans* (a.k.a. *logicals*) representations of *true* or *false*
 - A single bit is sufficient, but for storage a byte is usually the minimum
 - Bit fields represent multiple boolean values within bytes or larger words
 - Some languages like C and Perl have multiple specific notions of what values are “true” or “false”
 - *Characters* represent individual symbols with human-attached meaning
 - Traditionally single bytes (ASCII), but now generally two bytes (Unicode)
 - *Numbers* include integers, reals (floating point); sometimes rationals and fixed point
 - Different “widths” may be supported (32-bit, 64-bit)
 - Traditionally implementation dependent; not in Java though
 - Signed vs. unsigned sometimes distinguished (what a difference a bit makes!)
- Discrete/ordinal types: countable domains, clear successor/predecessor relationships

Enumeration Types

- Invented by Wirth in Pascal; emphasizes readability

```
type weekday = (sun, mon, tue, wed, thu, fri, sat);
```

 - Defines an order
 - Allows use in enumeration-controlled loops
 - May be used to index arrays (in some languages)
- In other languages (C, Java), enumerations are syntactic labels for other types of constants (integers, strings)
 - Allows cross-usage of enumerations and their underlying representations (e.g. integers, strings)
 - “True” enumerations in Java can be simulated through clever class definitions
 - Real enumeration type support coming in Java 1.5 (officially Java 5)
- Some languages can map enumerated types to arbitrary values

```
public enum Coin { penny(1), nickel(5), dime(10), quarter(25); ... }
```

Subrange Types

- Contiguous subsets of a discrete *base* or *parent* type

```
type test_score = 0..100;  
type workday = mon..fri;
```
- Ada distinguishes between *derived* and *constrained* types
 - Derived types are a new, distinct type — not interchangeable with their base types
 - Constrained types are interchangeable
- Subrange types help to clarify the intent of program code; a form of non-comment documentation
- Subrange types also facilitate automated range checking or storage optimization

Composite Types

- a.k.a. *constructed* types
 - Described by combining one or more simpler or composite types
- *Records* are collections of *fields* of other types
 - Equivalent to mathematical *tuples*
 - *Relations* are sets of tuples of similar structure; formal reasoning about relations forms the foundation of *relational database systems*
- *Variant records* represent *overlapping* fields — the record is a *union* of its named fields
- *Arrays* map an *index* type to a *component* type
 - Arrays of characters form *strings*, frequently treated as a very special kind of array in many languages
- *Sets* represent unordered collections of a *base type*
- *Pointers* are *references* to an object in that pointer's base type; can be generalized as any l-value (from Scott 6.1)
- *Lists* are ordered collections of a base type, generally defined by its head and its tail — the head is of the base type, and the tail is another list
- *Files* are structurally like arrays, but integrate I/O-bound behavior

Orthogonality in Types

- Analogous to orthogonality in expressions and statements
 - Many languages define an “empty” type for constructs that are used solely for their side effects: `void` (C), `unit` (ML)
- Other orthogonal type behaviors/features
 - Subroutines as first-class values (several languages)
 - Special case: arbitrary blocks of code as first-class values (Smalltalk)
 - Functions as types are particularly evident in ML (semantics of “curried” functions)
 - Ability to use any discrete type to index an array (Pascal)
 - Expressing values of any type (simple, composite) as some literal (C/C++, Java, Perl, Ada, ML)
 - Anonymous classes in Java allow inclusion of behavior (code) in addition to state (value)