Data Types

- modeling data as a collection of bits is complex and inefficient
- data types are abstractions provided by most programming languages
  - simple types: int, float, char, Boolean
  - structured types: arrays
  - mechanism for constructing new types: records

- pitfall: data types hide machine dependencies
- use of types improves reliability, readability, and maintainability
- types are typically associated with variables by a declaration

- a data type is a set of values and operations on those values.
  - example: type int has a set of values defined by max and min values
    - in Pascal maxint is a predefined, machine-dependent constant
    - in Modula-2 MAX and MIN are predefined functions
      - in C, limits.h contains constants
        - int_min is -32767 int_max is 32767
        - long_min is -2,147,483,647 long_max is 2,147,483,647
- operators
  - arithmetic on integers
  - succ and pred on enum types
  - field dereferencing
  - subscripts on arrays
- type checking is done by the translator, uses notion of type equivalence and rules for inferring new types from existing types
- a type system consists of type inference rules, algorithms for type equivalence, and methods for constructing types
- every language has predefined types

- type information can be used to improve error detection and allocate space
- type info can be implicit or explicit
  - implicit: types of constants and values, types that can be inferred from name conventions or context
  - explicit: in declarations

Strong Typing
- all objects have well-defined, statically determined types
- there is a set of rule for type equivalence and type inference that can be applied statically
Ada and Algol-68 are strongly typed

Modula-2 and Pascal have a few loopholes

C is not strongly typed

LISP, APL, and SNOBOL are weakly typed or untyped

Java is strongly typed but explicit casts allow explicit type errors

subset
  - subrange
  - inherit all operations from parent type

powerset
  - set of in Modula-2

Type Constructors

- Cartesian product – record or struct
- union discriminated or undiscriminated
  - Algol family – variant records (discriminated)
  - C – union (undiscriminated)

- array
- pointers and recursive types
- others – file type (Pascal), string types, packed types, void
Type Equivalence

- **Structural Equivalence** – two data types are the same if they have the same structure: built the same way from the same type constructors using the same simple types.

- **Name Equivalence** – two named types are equivalent if they have the same name.

- **Declaration Equivalence** – type names that lead back to the same original structure declaration are equivalent.

Type Checking

- **static vs. dynamic**
  - static checking – translator checks at compile time
  - dynamic checking – translator generates code to check at runtime
  - may not be specified by the language definition

- **Pascal**
  - standard requires static checking
  - runtime checking includes array index out of range, variant field record inactive
- C
  * uses static checking
  * many inconsistencies are removed by the compiler rather than flagged as errors

- Scheme
  * dynamically typed
    · no types in declarations,
    · type of variable changes dynamically with value,
    · types are maintained as explicit attributes of values

* requires dynamic checking
  · error checking is restricted to checking parameter types
  · there are facilities for the programmer to check types such as the function atom?

* type inference is essential to type checking
  - types of expressions are inferred from subexpressions
  - ML and Miranda use powerful mechanisms for type inference
Issues and Problems in Type Rules

• type compatibility
  – relaxes type equivalence
  – different types that may be correct when combined are compatible
  – assignment compatibility – needed to deal with l-values and r-values.

• overlapping types
  – assignments from overlapping subranges may cause error, need runtime check

• implicit types – must be inferred from rules that are explicit in the language

• shared operations – overloaded operators

Type Conversion

• multiply-typed objects – integer and cardinal in Modula-2, null ptrs

• needed in all languages

• can be built into type system
  – automatic or implicit conversion or type coercion
  – weakens type checking, compromises strong typing, less reliable
  – may not produce expected results
• can be supplied by type conversion functions
  – explicit conversion
• can be supplied by casts
• can be achieved by undiscriminated unions (loophole)