Chapter 7 -- Type Systems

- Most languages include the idea of a "type" -- classification of expressions and objects
- Chapter covers:
  - general type systems
  - type equivalence
  - type compatibility
  - sub-typing
  - polymorphism

- Type -- Why?
  - Limit operations in a semantically valid program. (e.g no atn("hello")
  - Provide implicit context for many operations
    - e.g. new my_class: allocate, construct, return pointer
  - May help program to be easier to read if types are explicit
  - If known at compile time, may help with generation and optimization

- Hardware often does not type bits
- High level languages: value, type pairs typically
- Languages have "type systems" that define types, operations, ...
  - type equivalence is important in many languages
  - type compatibility -- how types work together
  - type-inference -- rules of an expression based on the types of its constituent parts or context
  - type-checking -- checking for type system violations

Type systems (page 2)

- In the 1970s, most languages were "strongly typed" -- can’t mix types easily
  - Haskell: very strongly typed including logical assertions
  - C: very weakly typed
    - "systems programming needs a weak type system" (may not be true)
    - each new "version" of C has become stronger typed
  - JavaScript is weakly typed: true + true == 2
  - Dynamic type checking -- a form of late binding
- What is a type?
  - built-in?
  - user defined?
  - denotational semantics: set of values is a domain (aka type)
    - allows mathematical definitions
  - structural -- how types are built, e.g. enumeration
- Sub-type?
  - sub-domain of a domain: 1..10 of integers
  - Java: List is a special Collection, has extra operations
    - (I would call it a derived type, not a sub-type)
    - Depending on language: Base class or a Super class
  - is byte a subtype of integer?
    - Book shows that it is ... in Java

Polymorphism

- definition: Having multiple forms
- code/data structures designed to work with multiple types
  - parametric polymorphism: code takes a type as a parameter
  - subtype polymorphism: user can define extra operations
- parametric polymorphism: (static typing)
  - genetics (ada), templates (C++)
- subtype polymorphism: base classes, derived classes, ...
- combination can be useful: List(T) or list of T
- orthogonality is again very important in type systems
  - type with a single value?
    - void?
    - (void) f_call_that_returns_a_value()
  - Pascal? declare a "dummy" variable
  - Option type?
    - Haskell:

```
let divide n d : float option =
  match d with
    | 0. -> None
    | _ -> Some (n /. d);;
```

    - return is None or Some(x)
    - In several languages: Scala, C#, Swift, Java, C++
    - Swift -- Double? and nil return value (works similar with strings)

Aggregates

- Initialization of of variables
  - simple types -- single value
  - structured types -- aggregates
    - X = record name:string; age : int; .... end;
    - Y : X <- {"Phil", 17};
    - int z[10] = {1,3,4,7,11,18,29};
  - Orthogonality --- use in expressions?
    - Y := (age => 10, name ="Jane"); -- ada
    - z := (1 -> 1, 2 | 5 | 8 => 3, others => 0); --ada

Standard Types / Classification

- Most compilers: characters, integers, floating, boolean
  - booleans -- varies: C -- integers, Icon: success/failure
- characters
  - byte vs 16 bits
  - ASCII vs Unicode vs UTF-8 (ISO 10646 -- universal character set)
    - inclusion for "artificial" languages
- Numeric types
  - implementation defined? C, OCaml
  - signed and unsigned? (Modula-2 cardinal)
  - complex numbers? (Fortran, C99, Scheme, Chapel)
  - Rational numbers? (Scheme, Smalltalk)
  - BCD (Binary coded Decimal) or decimal types

More types

- Enumeration types
  - Pascal: type weekday = (sun, mon, tue, wed, thur, fri, sat);
    - succ(), pred(), ord() functions
    - for today := mon to fri do begin ....
  - C: names for integers
    - const sun = 0; const mon = 1; ...
    - enum weekday { sun, mon, tue, wed, ... }
    - enum special { value1 = 8, value2 = 15, ...}
  - Ada: weekday'val(1) is mon, weekday'pos(mon) = 1
- Refinement types (what I have called sub-types)
  - type score = 0..100; workday = mon..fri; (* Pascal *)
  - type Test_Score is new Integer range 0..100; --Ada
  - Haskell -- liquid type, logically defined
    - type { n:Int | n > 0 } or type { n: int | n < m }
  - interaction with "original" type?
  - code generation verify variable of type don't go outside bounds?
- Multiple integer sizes
  - short, int, long
  - int32_t, int16_t, int64_t...
  - do refinement types take less space: 1000 .. 1050
    - byte?

Composite types (Chapter 8)

- Record / Structs
  - Memory layout?
  - Packed type?
  - Can they be assigned?
- variant records / unions
  - Severe memory constraints, several fields use the same memory location
  - Some variant records have a "discriminant" field ... says which fields are used
    - compiler may check discriminant before accessing fields
  - Memory layout?
- arrays
  - char upper[26]; or upper : array [’a’ .. ’z’] of character;
    - [] operator (some languages use ())
    - C++ can overload [] -- why?
  - multidimensional arrays .. Mat : array (1..10, 1..10) of Long_Float;
  - Array slice -- part of an array
    - Fortran 90: (assume 1..10, 1..10) matrix(3:6, 4:7), matrix(6:,5), matrix(:4,2:8:2), matrix(:, (/2,5,9/))
  - Array assignment?
  - Full array manipulation?
    - APL, Chapel

Composite types (page 2)

- arrays (continued)
  - Implementation:
    - Allocation: static, stack, heap (static sizes vs dynamic sizes)
    - Dope vector -- number of dimensions, lower and upper for each dimension ...
      - might also use one for a dynamic sized vector
    - run time checks ? (bounds checking)
    - parameters / local variables often are dynamic
    - void square(int n, double M[n][n]) { ... double T[n][n]; ....
    - Early languages (Pascal) had to know size of array parameter statically
    - Later modifications/languages ... shape defined at call ... e.g. conformant arrays
      - typically know number of dimensions, but not bounds
    - Stack implementation: fixed location pointer, variable part of stack (call) record
  - Ordering:
    - row major vs column major ordering in linear memory
      - which order of use can effect speed of array processing
    - read book about cache line performance
    - some languages user row-pointer implementations (eg. array of strings)
  - Address calculations -- how to calculate location of an element of an array
    - Depends or ordering ....
    - x : A(1..10, 10..15, 0..19) of integer
    - x[i,j,k]: base address + (i-1)*S1 + (j-10)*S2 + (k-0)*S3
    - Are methods for limiting math operations (read about indexing)

Composite types (page 3)

- sets
  - like a mathematical set, but composed of elements of types with distinct values (ints, chars, enumerated)
  - Pascal has these a base types, overloaded + (union), * (intersection) - (set difference)
    - Had an "in" operator for set membership
    - Typical implementation was a bit vector in Pascal ... has issues
      - set of uint64_t -- too many bits!
      - uses dynamic sized hash tables ...
  - Python and Swift have sets
  - many other languages have sets as library support types
  - other languages provide associative arrays or dictionaries or maps (not really sets)
- pointers and recursive types
  - Pointers -- high level concept, may not be implemented by a low level address
  - Typically needed in languages that use heap allocation
  - Two types: explicit use by programmer (may have limited operations)
    - implicit use by programmer (often uses garbage collection, Java, C#, Scala, Go, ...)
  - Explicit allocation often has explicit deallocation (Pascal, C, C++, ...)
  - General use of pointers (C) vs restricted use of pointers (Pascal)
    - pointers only can be created by the "new" operator
  - Typical operations: allocation, deallocation, dereferencing, assignment, comparison
  - Often used to point to records (and objects)
    - recursive is that a record can point to its record type

Composite types (page 4)

- pointers and recursive types (continued)
  - Assignment: depends on model: reference or value
    - A = B: // copy a pointer or copy the object?
  - Pascal: A = B; (* pointers -- copy pointer *) A^ = B^; (* copies data *)
  - Functional languages often use reference model
    - OCaml: Node ('R', Node ('X', Empty, Empty), Node ('Y', Node ('Z', Empty, Empty), Node ('W', Empty, Empty)))
    - Read book on Lisp
  - Recursive structs: C: struct x_tree { int val; struct x_tree *left, *right; }
    - heap allocation of "struct x_tree"
  - Access: C: var->val; Pascal: var^.val; var.val (* both simplevar or pointer *)
    - Read about functional language and dereferences
- Stack smashing (* an aside in the book *)
  - Could be solved by checking before use of indexes and so forth
- Explicit allocation/deallocation and dangling references
- Garbage collection avoids dangling references ...
  - If you have a reference, the object is valid
  - Once there is no more references, the object can be recycled/garbage collected
  - tradeoff between safety/convenience and performance
  - Reference counts (kept in the object pointed at, updated at pointer copy, pointer destruction)
    - reference counts and circular structures ... may not have access to a circular structure
    - frequent updates, updates in threaded/parallel contexts
  - Big question ... what is garbage?

Composite types (page 5)

- Garbage collection (continued)
  - various algorithms/methods for determining garbage
  - scanning the objects (mark and sweep) (O(n) number of objects!)
  - read about garbage collection
  - read about memory management in Rust
- lists -- a common abstraction
- Definition: List is either empty or an item and an associated list.
- Basic data type in some compiled languages (Clu) and many scripting languages
- Core data type in many functional and logic logic languages
- Any language with records and pointers can make them, often in some standard library
- Two kinds: homogeneous (ML) and heterogeneous (Lisp)
- Two implementations:
  - ML -- chain of variant records: node tag, element (based on tag), pointer to remaining
  - Lisp -- pair of pointers: "car" to the current element, "cdr" to the remaining list
    - IBM 704 -- "Contents of the Address field of Register", "Contents of the Decrement field of register)
- Variety of list notations.
  - Lisp: ’(a b c), (car ’(a b)) -> a (append ’(a b) ’(c d)) -> (a b c d)
  - OCaml: [’a’, ’b’, ’c’], hd [’a’; ’b’];; -> ’a’ ...
  - Various other mechanisms ...
- tuples
  - lightweight records or arrays:
- files -- Some languages have them as a composite type

Type Checking (Section 7.2)

- Statically typed languages ... every object has a type
- Use of objects together --- verify type compatibility
- What is type equivalence?
  - base types -- easy
  - user defined types -- have some rules ...
    - structural equivalence
    - name equivalence
    - declaration equivalence (not in book)
  - Consider these declarations (no specific language)

```
type R1 = record a, b : integer end;
type R2 = record a, b : integer end;
type R3 = record b, a : integer end;
type R4 = R1;
var v1, v2 : record c, d : integer end;
```

  - structural equivalence -- same structure
    - R1, R2, R3, R4 and types of v1, v2 are all the same (equivalent)
  - name equivalence -- named the same
    - All are different types
  - declaration equivalence -- type declared at the same point (record ... end)
    - R1 and R4 are equivalent, all others are different
  - name and declaration are easy for the compiler to enforce

Problems with type equivalence

```
type student = record name, address: string; age : integer; end
type school = record name, address: string; yearopening : integer; end
var x : student; y : school;
.....
x := y;
```

- Assignment won't be flagged in structural equivalence
- Assignment will be flagged in name equivalence (and declaration)
- Type aliases
  - The place where declaration and name differ

```
type A = B;   // B is original
typdef old_type new_type;  // C's definition of alias
```

  - Same type in declaration, differ in name --- which sounds "correct"?

```
type celsius_temp = real;  fahrenheit_temp = real;
Var c : celsius_temp;  f : fahrenheit_temp;
....
f = c;
```

- Error for that assignment

```
type A = record x: pointer to B; y : real; end; B = record X; pointer to A; y : real ; end
```

- A and B the same?
- Algorithm to check for type compatibility?

Scala solution for this issue

- structural for known definitions, strict name for "opaque" types

(Code from the book)

```
type mode_t = Int // type alias whose definition is visible

object Temperature {
    opaque type Celsius = Float
    opaque type Fahrenheit = Float
    def celsius(t:Float):Celsius = t
    def celsiusToFahrenheit(t:Celsius):Fahrenheit = (t * 9 / 5) + 32
    ... // other constructor and conversion declarations
}

val c : Temperature.Celsius = Temperature.celsius(34)
var f : Temperature.Fahrenheit = Temperature.celsiusToFahrenheit(c)
...
val mode : mode_t = 5 // OK: mode_t is known to be equal to Int
f = c // Error: found Celsius, required Fahrenheit
```

(Ada solution)

```
subtype Mode_T is Integer range 0..2**16-1; -- unsigned 16-bit integer
...
type Celsius_Temp is new Integer;
type Fahrenheit_Temp is new Integer;
```

Type checking, conversion and casts

- static typing requires type checking

  a := expr; // Same types?

  ... a + b ... // Same types, works with +

  .... f(e1, e2, e3); // do actual types match formal types

  - type matching in concert with overloading and default parameters?
- Compiler needs to do checking for types in many places
  - Typically done during semantic checking, AST generation
  - Implements one of the 3 kinds of type equivalence
- type conversion (casts) is how to "mix types"
  - enumerated <--> integers
  - integers -> floating
  - Some can cause code (e.g. float <--> integer), some no code generated (enum -> integer)
  - C: r = (float) i; // run-time conversion
  - C: i = (int) r; // run-time conversion, no overflow check
  - C++: static_cast<int>(expr)
  - C++: dynamic_cast<TargetType>(expr) // runtime checked, often in polymorphic classes
- type coercion -- changing types without an explicit cast
  - C does it a lot! (see book)
  - happens in weaker type systems
- Universal reference type?
  - C has "void *", Scala has "Any", modula-2 has "address", Java has "Object", ...
  - type checking with universal reference type?
  - Java, C# -- objects have tags, can’t cast from universal type to wrong object type

Overloading, Generic subroutines and classes

- Overloading -- typically subprograms, same name different parameters
  - more than just type checking ... match subprogram given parameters

    func min (x, y : integer) returns integer; ...

    func min (x, y : real) returns real; ...
  - require multiple functions
  - appears to be same function with different types
  - checking types of arguments only? use return type also to disambiguate
- generic subroutines -- define once, use for multiple types

  C++: template <typename T> T min(const T a, const T b) { return (a < b) ? a : b; }
  - requires a type for which < works ... < could be overloaded too (min.cc)
  - book has an ada version which also includes a "<" function as well as a type
- generic classes -- works well for container classes, e.g. lists, stacks, maps, ...

  C++: template <class item, int max_items = 100> class queue { ... }

  queue <process>; or queue<int,50>
- read 7.3.3 Generic parameter constraints
- C++ does implicit instantiation, Ada requires explicit
- C++ just use the function (e.g. min), Ada: int_min is new gMin(int,">") ...
  - (Ada has overloading so you might be able to do min id new gMin(int,">") ...
- Section 7.4 talks about generics in several languages ... read it

Overloading (page 2)

- An example of where overloading may work better than generics -- I/O
  - C++ overloads the << operator for I/O. New types can also overload << for I/O
    - has some disadvantages as: cout << setw(5) << intvar;
  - My solution ... in ATL(my language): overloading, default parameters, syntactic sugar

Equality testing and assignment (7.5)

- Simple types / base types -- easy assignment and equality testing
- More complicated types -- issues arise
  - Strings for example: s = t (= is equality, not assignment)
    - are aliases?
    - same bits over entire storage?
    - same sequence of characters up to some end-of-string character?
    - would look the same if printed?
    - only same bits over entire storage may be appropriate
- Questions of l-values or r-values in comparison / assignment
  - Reference: are they the same object (shallow comparison)
    - are they different objects with the same data (deep comparison)
    - deep comparison may require recursive traversal. e.g. tree?
- Similar questions appear for assignment
  - shallow copy -- copy the reference
  - deep copy -- copy the structure
  - C++ -- programmers have to know which is used
  - some languages provide different comparison/assignments operators