Chapter 7 Data Types

June 23, 2015

7.1 Type systems

- Hardware can interpret bits in memory in various ways: as instructions, addresses, integers, floating point, fixed point, or perhaps even as character data.
- But bits are bits; types are conceptually imposed over the bits, and not realized in hardware.

7.1 Type systems

- A type system consists of
  - a mechanism to define types and associate these types with language constructs
  - and a set of rules for type equivalence

7.1 Type systems and subroutines

- So what about subroutines? Some languages imbue these with type; others don't. If a subroutine is first-class, that's a strong reason to give a type; if it is third-class, and the language is statically scoped with no provision for dynamic references to a subroutine, then one could dispense with types for subroutines.

7.1.1 Type checking

- Type checking is an attempt to ensure that a program enforces its type compability rules. Violations are genearlly called a *type clash*.
- A language can be called *strongly typed* if it prohibits the application of any operation to any object that is not intended to support that application.
- A language can be called *statically typed* if it is strongly typed and type checking can be performed at compile time. (The term is also sometimes applied to languages that can do most of their type checking at compile time with only a small portion done at run time.)

7.1.1 Type checking

- “Dynamic type checking is a form of late binding” [page 291], and is a common attribute of languages that do other forms of late binding.
- See also Wikipedia’s take on this

7.1.2 Polymorphism (revisited from page 148)

- Fundamental definition: “Polymorphism allows a single body of code to work with objects of multiple types.” [page 291]
- This idea becomes more subtle than one might think, and often what polymorphism actually entails is part and parcel of the language definition.
- For instance, Haskell’s take on its polymorphism.

7.1.3 The meaning of “type”

- Some languages offer on a small set of fixed types (Fortran 77 and Basic, for instance)
- Some languages can infer types at compile time (ML and Haskell, original work was done by Milner; take a look at Mil78 in the book’s references on page 860)
- Most languages depend on the programmer to provide type declarations.

7.1.3 Three ways to think about types

- Denotational: a type is a set of values. This way of thinking lets us use the well-honed tools of set theory.
- Constructive: a type is either built-in, or can be constructed from other types.
- Abstraction-based: a type is an interface consisting of a set of operations with well-defined and consistent semantics (and that is a pretty abstract definition since it seems to cover more than just types)

7.1.4 Classification of types

- Numeric types: unfortunately, a lot of languages are quite lackadaisical about just how many bytes are in their numeric types. Some languages have signed and unsigned types; some have fixed point numbers or BCD; some languages support complex numbers, rationals, imaginary numbers, and even p-adic numbers.
- Scripting languages usually support multiprecision numbers.

7.1.4 Enumeration types

- Since Pascal, it has been common for languages to allow enumeration as a set of values that generally given as a sequence of characters.

```
type weekday = (sun, mon, tue, wed, thue, fri, sat);
for today := mon to fri do begin ...
var daily_attendance : array [weekday] of integer;
```

7.1.4

- Some languages have treated these enumerations as some version of integer, and allowed integer operations over these, like C:

```
enum weekday {sun, mon, tue, wed, thu, fri, sat};
int main()
{

  enum weekday day1 = sun;
  enum weekday day2 = mon;
  enum weekday day3;
  day3 = day1 + day2;
}
```

7.1.4 Subranges

- Many languages allow constructions like 12..24 or sun..fri.
- Haskell, because of its lazy evaluation scheme, even allows just 12..

7.1.4 Composite types

- Records/structures
- Variant records/unions
- Arrays
- Sets
- Pointers
- Lists
- Files

7.2.1 Type equivalence

- Structural equivalence: two types are equivalent if they have the same components in the same order.
- Name equivalence: different names mean different types, even if they are structurally identical.
- But what if you create type x = y, thus creating only an alias? Strict name equivalence means that aliases are different; loose means that aliases are treated as equivalent.
- Ada lets you specify how you want it viewed.

7.2.1 Type conversions and casts

- Could be merely conceptual if the underlying representations are the same
- Could be that the underlying values are formed the same way, but still might need run-time checks to verify that non-translatable values are detected.
- Could be that the underlying representations are different, but that feasible means exist for creating correspondences (integers to reals, for instance.)

7.2.2 Type compatibility

- Some languages allow a lot of “type compatibility”, going to the extent of doing extensive type coercions (Perl is a champ at this; C also does quite a number.)

Containers

- Some languages that allow “containers” also allow a “universal” reference type (think of C’s void *, for instance). These usually lead to some difficult questions, though, that end up having run-time solutions, such as including a type tag.

7.2.3 Type inference

- Consider the following:

```
type Atype = 0..20;
type Btype = 10..30;

var a: Atype;
var b: Btype;
```

- What if you wanted to add a to b? Does that even make sense? If so, what "type" would the result be? How about just making it an integer? Or . . . ?

What about indexing?

- Some languages infer the type of an index... if a language does so, should that be an implicit subrange type?

7.3 Records/Structures and Variants/Unions

- Fundamentally, records and structures allow 1) aggregation into a single data structures components of different types and 2) a method of addressing each component of an aggregate
- Usually this means 1) simply putting components adjacent to each other in memory, and 2) maintaing an offset for each component

Simplifying the syntax

- Some languages (notably the Pascal family) allow one to simplify accesses to complicated structures with the idea of a "with" statement:

```
with complex.complicated.deep.stuff do
begin

  fieldx := 'STUFF GOES HERE';
  fieldy := 'STUFF GOES THERE';
  fieldz := 'STUFF GOES EVERYWHERE'

end;
```

7.4 Arrays

- A very common composite data structure. Most languages have significant syntactical and semantic support for arrays. Fundmentally, an array takes an index to a value.
- Your text takes the approach of treating associative arrays as part of this spectrum of various types of indices; this works well with the then conflation of arrays with functions.

Array declarations

- It is popular for programming languages to allow enumerations and ranges to used in an array specfication.
- Multi-dimensional array declarations are also common in programming langauges.

Slices

- Slices allow one to specify a “rectangular” portion of an array;

Conformant arrays

- Conformant arrays give the ability to reference the bounds of an array from its “dope vector” rather than having to pass these dimensions along explicitly.

Dynamic arrays

- When the dimensions of an array are allowed to change, it is said to be “dynamic”; while the meta-information about such a dynamic array can be kept on a runtime stack, the actual array needs to be in a heap of some sort.

Memory layout

- Most languages put all of the elements of an array in a single contiguous area of memory (though certainly not all; Perl apparently does not do this with its “l-o-l” (also, look here)).

Strings

- While many languages treat strings largely as just an array of characters, some give strings a separate type that allows operations that are not applicable to other arrays.
- Generally, most languages give strings a reasonably prominent place, both in syntax and semantics, since string processing is so fundamental to real-world problems.

Sets

- Pascal was the first language to explicitly have a set type; it overloaded the “+”, “*“, and”-” operators to support support set union, intersection, and difference.
- Generally done as a bit vector for smallish sets, and either the language forbids largish sets, or uses a more sparse approach.

Pointers and recursive types

- Pointers are fundamental to many implementations of many of computer science's favorite idioms: lists, trees, red-black trees, tries, skip lists, splay trees, scapegoat trees, heaps, sets, . . .

Pointers and models

- Reclaiming unneeded storage: some languages, like C, leave this task to the programmer; others automate the process with *garbage collection*.
- Failing to reclaim storage creates *memory leaks*; conversely, freeing storage that is still in use creates *dangling references*. Freeing already freed storage creates *double frees*.

Pointers and value models

- If there's anything that can be said generally on this subject, it's that this has been a "free-for-all" area for programming languages, spanning various ideas about l-values and r-values, and what is an explicit reference to a memory location and what is an implicit one. The best current reference on the web for this subject is at Rosetta code's Pointers and references page.

Note on page 353 relating to “stack smashing”:

- “It would never have been a problem in the first place, however, if C had been designed for automatic bounds checks.”
- Yes, and I would note that it also would not have been as serious a problem had return addresses been kept on a separate stack from activation records; or if there had been hardware support for array boundaries; or a host other techniques.

C allows conflation of pointer arithmetic with array access

- Although it's past its utility, the original idea of allowing the programmer to do the pointer arithmetic behind flat array implementation was quite efficient, though modern compilers can usually better a programmer's efforts.
- Try this program: test-sizeof.c

Garbage collection

- Reference counting is a popular and relatively simple way to implement garbage collection, but it needs some language support (for instance, reference counts in C would be hard to imagine, since malloc() is not even part of the language but is rather just a library call.)

Garbage collection

- Other conundrums:

Garbage collection

- As the book notes, even Perl is subject to the circular reference problem.

Garbage collection

- Mark-and-sweep, classic three steps:
  - Walk "the heap", identifying every program data item as "useless"
  - Walk all linked data structures that are outside "the heap", marking all items in "the heap" that are reachable as "useful"
  - Walk "the heap" again, removing all of the program data items still marked as "useless"
- Implementation issues galore, though, and language implementations planning to use mark-and-sweep should plan on this from the beginning

Garbage collection

- Stop-and-copy, classic unpredictable time problems (also known as “stop the world” pause problem)
- Usually fold compaction into this method

7.8 Lists

- “In Lisp, a program *is* a list, and can extend itself at run time by constructing a list and executing it. . . ” (page 365).

7.8 Lists

- While lists are much of a muchness is most languages, in many languages (and particularly the functional languages) now also support list comprehensions.

7.10 Equality and assignment

- As the text points out, consider equality of s == t expressed as
  - s and t are aliases
  - s and t contain exactly the same characters
  - s and t appear exactly the same if “printed”
  - s and t occupy storage is the same bitwise
- While the last is clearly problematic (what if s and t contain non-semantically significant bits that are not the same?) the others can be useful measures

Equality and assignment

- And what about assignment? If s := t and t is a large recursive structure, the language might just do a shallow copy of t as a pointer, or maybe it does a deep copy, walking the entire structure of t and creating a fresh copy in s.
- Or maybe the language supports both shallow and deep mechanisms.

7.11 Wrapping up

- A type system consists of any built-in types, mechanisms to create new types, and rules for type equivalence, type compatibility, and type inference
- A strongly typed language never allows an operation to be applied to an object that does not support it. (However, the notions of strong and weak typing are *not* universally agreed upon.)
- A statically typed language enforces strong typing at compilation time.

7.11 Wrapping up

- Explicit conversion: the programmer *expliclitly* converts s into t
- Implicit coercion: the program needs s to be a t, and *implicitly* makes it so.
- Nonconverting casts (type punning): welcome to C!

7.11 Wrapping up

- Composite types: records, arrays, and recursive types.
- Records: whole-record operations, variant records/unions, type safety, and memory layout.
- Arrays: memory layout; dope vectors; heap-based, stack-based, and static-based memory allocation; whole-array and slice-based operations.
- Recursive structures: value versus reference models for naming and reference; most general way of structuring data, but has the highest costs