Chapter 8 Composite Types

June 20, 2016

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.
- Types are cognitive constructs that can
  - allow humans to read code more easily
  - allow humans to understand and reason about code more fully
  - allow humans to *limit* constructs to reasonable inputs and constrain the outputs

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

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;
```

Record-like constructs in the ML family

- Generally, record-like constructs in the ML family are not in fact such (viz., records are not first-class or even inherent in Haskell), but generally are more awkward constructs; see for instance A modest proposal for records in Haskell, the discussions about GHC's record implementation at https://ghc.haskell.org/trac/ghc/wiki/Records and here.

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 subsequent 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 langaugges.

Slices

- Slices allow one to specify a “rectangular” portion of an array. Some languages support semantics for slices that include removal, addition, modification of values, comparison, and assignment.

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)).
- While of course either heap or stack allocation is feasible for a lexically declared array of fixed characteristics, if you want more flexible arrays, such as those created by malloc(3) or that have flexible characteristics (resizable components, for instance), then generally these will be heap-allocated, though of course the pointer to this could still be stack-allocated (or wherever the language implementation is putting its activation records.)

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 a powerfully convenient mechanism for 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 385 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 of 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