CSE 3302 Notes 8: Structuring Data

(Last updated 11/14/15 4:59 PM)

References: Gabbrielli: 8; Crockford: 3-6

8.1. DATA TYPES

Types = Means for assuring operations are applied to appropriate objects (values) (Gabbrielli, Definition 8.1)

History of types = History of programming languages

Cardelli and Wegner, "On Understanding Types, Data Abstraction, and Polymorphism", *ACM Computing Surveys 17 (4)*, Dec. 1985, `http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=6041.6042`
See section 1.4 and figure 2 (p. 516)

Also, W.R. Cook, "On Understanding Data Abstraction, Revisited", OOPSLA '09,
`http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=1640089.1640133`

`http://en.wikipedia.org/wiki/Expression_problem` `https://www.dreamsongs.com/Files/Incommensurability.pdf`

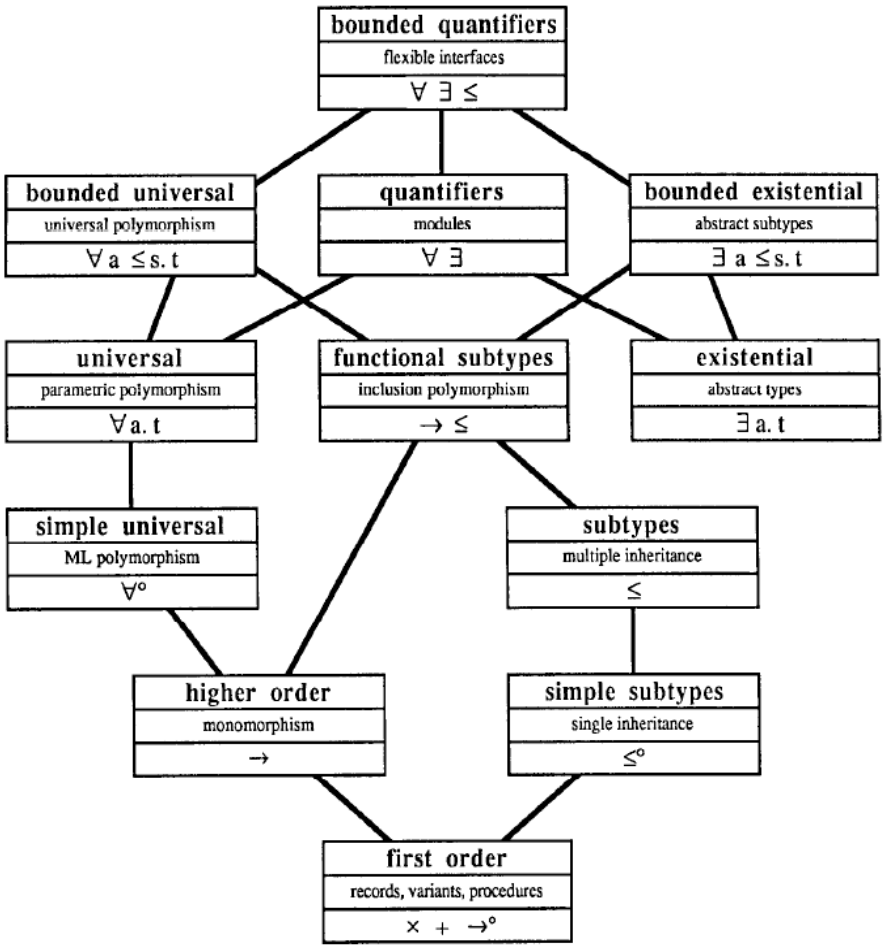

Figure 2. Classification of type systems.

8.2. TYPE SYSTEMS

Friction between flexibility and point of type error detection:

`http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=2661061.2659764`

`http://www.typescriptlang.org/`

Software Types ≠ Hardware Types (interpreter vs. compiler, dynamic vs. static; undecidability, Gabbrielli, p. 204)

In what ways are user-defined types different from built-in types?

C++: Can't overload operations for built-in types

Strong vs. Weak Typing:

Is it clear when the type system is being abused?

Racket - mutable lists (`mcons`, `mcar`, `mcdr`)

Aside: Software Engineering - "domain analysis" (circa 1990) and "software product line engineering" (`http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=2620784.2580950`)

8.3. SCALAR TYPES

Booleans . . .

Characters . . .

Strings (not mentioned in Gabbrielli)

Mutable (C) (see CACM, Sept. 2011, "The Most Expensive One-Byte Mistake", `http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=1995376.1995391`)

vs. Immutable (Java, JavaScript) strings

vs. Storing lengths

Numbers . . .

Enumerations

C: Maps to 0 . . .

Pascal: Maps to 1 . . . (many compilers allow overriding)

Intervals (of allowed values, subranges)

Pascal: `var negval: -2002 . . -2001;`

Implementations typically use the smallest hardware type that contains (bits are not minimized)

Not type safe

8.4. COMPOSITE TYPES (AKA structured types)

Records

C - fields are allocated/aligned in order given
(http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES08/recsize.c)

Pascal: (http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES08/dt.pas)

`packed` option to reduce space (for data, but not code)
`with` to abbreviate field selection expressions

"Overlapping" portions of records:

COBOL (aside: http://fivethirtyeight.com/features/the-queen-of-code/) - `redefine`

Pascal - variant part of record

```
type conrec =
  record case tp: types of
    ints, chars, bools: (i: integer);
    reals: (r: real);
    notyp, arrays, records: ();
  end;

s: array [1..stacksize] of    (* blockmark:                *)
   record case types of       (*    s[b+0] = fct result    *)
     ints:   (i: integer);    (*    s[b+1] = return adr    *)
     reals:  (r: real);       (*    s[b+2] = static link   *)
     bools:  (b: boolean);    (*    s[b+3] = dynamic link  *)
     chars:  (c: char);       (*    s[b+4] = table index   *)
     notyp, arrays, records: ()
   end;
```

C - `union`, similar situation with tagged (discriminated) and untagged (free) versions

Leads to type conversion shortcuts and limits type checking

Arrays

1-d arrays of integers for PL/0 (Gabbrielli, figure 8.5)

Slices - specifying a vector or sub-matrix for use in functions or built-in operations

Dope Vectors

Offsets for fields within records

Constants needed for subscripting (historical)

Dimension sizes (e.g. decrease left-to-right across dimensions)
Range lower bounds (not for C/Java)

Memory Layout

Row-major (rows are contiguous bytes)

Column-major (columns are contiguous bytes, FORTRAN)

Row-pointer (multidimensional array handled using 1-d concepts multiple times)

Row subscript indexes array of pointers
Column subscript goes to position within row
Allows ragged arrays (e.g. triangular situations)

Address Calculation for row-major

Suppose an array is to be stored starting at location `1000000` and is declared:

```
a: array[10..25,50..70,200..300] of integer;
                  21     101           4
```

The address of `a[i,j,k]` (with 4 byte integers) is computed as:

```
1000000 + (i-10)*21*101*4 + (j-50)*101*4 + (k-200)*4
```

But may be simplified (at compile time) to:

```
1000000 + (0-10)*21*101*4 + (0-50)*101*4 + (0-200)*4
        +      i*21*101*4 +      j*101*4 +       k*4
```

for which the first line (address of non-existent `a[0,0,0]`) is a constant and the second line may be computed as:

```
        +     ((i*21   +          j)*101   +      k)*4
```

(It is not difficult to go from an address back to the subscripts)

Sets

Pascal sets (`set of`) - not an associative array, provides convenient implementation of long bit vectors

`in` membership `+` union `-` difference `*` intersection

Many examples in PL/0 and Pascal-S environments

(JavaScript - sets are easily simulated by manipulating an object's properties)

Pointers and References

Value Model (containers and addresses) - C (pointer arithmetic, order comparisons) and Pascal

Reference Model (every access involves both "address" and container) - Java and JavaScript

Recursive Types . . .

ML Built-In Types - lists `[]`, tuples `()`, records `{}`

```
val a=[1, 2, 3, 4];
val b=[1.0, 2.0, 3.0, 4.0];
val c=["cat", "dog", "fish"];
val d=[#"a",#"b",#"c"];

- hd(a);
val it = 1 : int

- tl(a);
val it = [2,3,4] : int list

- hd(a)::tl(a);
val it = [1,2,3,4] : int list

- val e=[(1,2.0),(3,4.0),(5,6.0)];
val e = [(1,2.0),(3,4.0),(5,6.0)] : (int * real) list

- datatype ('a,'b) element=P of 'a * 'b | S of 'a;
datatype ('a,'b) element = P of 'a * 'b | S of 'a

- val f=[S(2.0),P(2.0,1),S(3.0),P(4.0,3)];
val f = [S 2.0,P (2.0,1),S 3.0,P (4.0,3)] : (real,int) element list

- tl(f@f);
val it = [P (2.0,1),S 3.0,P (4.0,3),S 2.0,P (2.0,1),S 3.0,P (4.0,3)]
  : (real,int) element list

- tl(f)@f;
val it = [P (2.0,1),S 3.0,P (4.0,3),S 2.0,P (2.0,1),S 3.0,P (4.0,3)]
  : (real,int) element list

- #2(hd(tl(e)));
val it = 4.0 : real
```

```
- val P(_,h)=hd(tl(f));
val h = 1 : int

- val beatles=[{name="John",plays="keyboards",born=1940},
= {name="Paul",plays="bass",born=1942},
= {name="George",plays="guitar",born=1943},
= {born=1940,plays="drums",name="Ringo"}];

val beatles =
  [{born=1940,name="John",plays="keyboards"},
   {born=1942,name="Paul",plays="bass"},
   {born=1943,name="George",plays="guitar"},
   {born=1940,name="Ringo",plays="drums"}]
  : {born:int, name:string, plays:string} list

- tl(beatles);
val it =
  [{born=1942,name="Paul",plays="bass"},
   {born=1943,name="George",plays="guitar"},
   {born=1940,name="Ringo",plays="drums"}]
  : {born:int, name:string, plays:string} list

- hd(tl(beatles));
val it = {born=1942,name="Paul",plays="bass"}
  : {born:int, name:string, plays:string}

- #name(hd(tl(tl(beatles))));
val it = "George" : string

(** Top-down merge sort **)

fun merge([],ys,_) = ys
  | merge(xs,[],_) = xs
  | merge(x::xs,y::ys,pred) =
      if pred(x,y) then x::merge(xs,y::ys,pred)
                   else y::merge(x::xs,ys,pred);

fun tmergesort([],_)  = []
  | tmergesort([x],_) = [x]
  | tmergesort(xs,pred) =
      let val k = length xs div 2
      in  merge(tmergesort(List.take(xs,k),pred),
                tmergesort(List.drop(xs,k),pred),
                pred)
      end;

- tmergesort([3.0,1.0,5.0,4.0,2.0],op<=);
val it = [1.0,2.0,3.0,4.0,5.0] : real list
```

Scheme code for mergesort

```
(define (tmergesort lst pred)
  (define (merge lst1 lst2)
    (cond
      ((empty? lst1) lst2)
      ((empty? lst2) lst1)
      ((pred (car lst1) (car lst2))
       (cons (car lst1) (merge (cdr lst1) lst2)))
      (else (cons (car lst2) (merge lst1 (cdr lst2))))))
  (define (mergesort lst)
    (if (pred (length lst) 1)
        lst
        (let ((k (floor (/ (length lst) 2))))
          (merge (mergesort (take lst k))
                 (mergesort (drop lst k))))))
  (mergesort lst))

(tmergesort '(5 7 3 4 2 9 1 0 6) <=)
```

8.5. EQUIVALENCE

Equivalent = "interchangeable" (ignoring scope issues)

Opaque (name) equivalence

Weak = aliases are equivalent Strong = aliases are not equivalent

Transparent (structural) equivalence

Structs/records - Pascal and C use weak name equivalence

Arrays - C uses structural equivalence; Pascal uses weak name equivalence (top of p. 232 example)

8.6. COMPATIBILITY & CONVERSION

Compatibility = "substitutability" especially as an argument to a function (Gabbrielli, p. 234)

Subtypes (`S <: T`, p. 242) enter the picture:

Ranges for numbers (e.g. `char` vs. `int`)

Racket structs may inherit from a supertype

C++ structs may inherit from a supertype `struct` (what about those functions in the struct?)
`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES08/notes08.struct.cpp`

Forms of Compatibility (p. 234)

1. Equivalence.
2. Values of the type used are in a subset of the type expected.
3. Operations on values of type `S` are allowed on values of type `T` by inferring `T <: S`.

   http://en.wikipedia.org/wiki/Liskov_substitution_principle

   Liskov & Wing, "A Behavioral Notion of Subtyping", *ACM TOPLAS 16 (6),* http://dl.acm.org.ezproxy.uta.edu/citation.cfm?id=197383 (especially figures 4 and 5)

   (Are circles a subtype of ellipses? Squares of rectangles? `long`s of `short`s? The issue is?)

4. There is a canonical way to convert values of type `T` uniquely to type `S`.
5. There is a way to convert values of type `T` non-uniquely to type `S`.

   (Liskov & Wing, stacks and queues could be subtypes of a bag supertype)

4 and 5 are difficult to distinguish, so checked casts are useful:
http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES08/notes08.narrowCast.cpp
(Same idea is used in JavaScript code to check string-to-number conversions.)

Type Conversions and Casts

Narrowing - going from a value in a large set to a small set (often unsafe, can't "undo" back to original values)
Widening - going from a value in a small set to a large set (often safe, can "undo" to original value)
Coercion - implicit conversion, defined by language/implementation (*mixed mode* operations)
Cast - explicit conversion by programmer

(P.N Hilfinger, "An Ada Package for Dimensional Analysis", *ACM TOPLAS 10 (2)*, Apr. 1988, 189-203, http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=42190.42346 simulates traditional "unit cancellation". (Also done for C++ in http://www.stroustrup.com/Software-for-infrastructure.pdf)

Nonconverting cast in C through pointer casts or `void*` ("universal object reference")

8.7. POLYMORPHISM

Overloading/Ad-Hoc Polymorphism

A name corresponds to several objects, such as:

Built-in operators (for different types) in most languages
Reusing a function/method name for different argument signatures

Coercions are a separate issue and reduce the apparent number of reuses.

(Universal) Parametric Polymorphism

Simple Java Generics or C++ Templates - unrestricted type parameter
Non-Dictionary Collections - stacks or queues
Dictionary Collections - even without inherent ordering
Explicit - type parameter is supplied (C++ queue example in Notes 02)
Implicit - type possibilities are inferred at compile-time and instantiated at run-time
C++ - types may be inferred, without explicit instantiation, for function templates from the argument types or by using `auto` instead of an explicit type.

Subtype/Inclusion (Universal) Polymorphism

The type argument for a type parameter is limited to subtypes of a type
Includes subtyping based on built-in structured types and class hierarchies
`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES08/notes08.max.cpp`

Skip 8.7.4 (Implementation)

8.8. Type Checking & Inference

Only in the presence of parametric polymorphism is this a general/interesting problem - traditional compilers derive type information bottom-up for expressions

ML uses type inference heavily by avoiding declared name-to-type bindings

Gets context/hints from places like:

Operations/functions applied (e.g. from libraries)
Names that are referenced

8.9. Safety

Unsafe - C, C++
Locally Unsafe - Pascal
Safe - Java, functional languages, JavaScript?

8.10. Dangling References

Tombstones/ indirection

Any conceptual pointer always takes two steps
Freeing the actual object's memory clears the tombstone's pointer
Prevents memory re-allocated for a different object from getting clobbered

Locks and Keys

> Also takes two steps, but now it is a comparison between accessor's key and the stored lock value
>
> http://en.wikipedia.org/wiki/Capability-based_security

For C++ 14, these issues (along with reference counts and concurrency) are often handled through *smart pointers* (unique or shared)

And, just to make the next topic more tedious: *weak pointers*, largely in support of result caching

http://en.cppreference.com/w/cpp/memory/weak_ptr
http://docs.oracle.com/javase/7/docs/api/java/lang/ref/WeakReference.html
http://docs.racket-lang.org/reference/eval-model.html#%28part._gc-model%29

8.11. Garbage Collection

Reference Counts (eager)

> Each allocated object has count of pointers to it
> Count decremented to zero . . . reclaim
> Various schemes to reduce counter update costs (exploit compiler optimizations)
> Cycles are a potential problem

Mark-and-Sweep (lazy) - uses techniques analogous to directed graph traversals (DFS and BFS) for determining reachable heap locations. Requires separate sweep of heap (Compact) to clean-up external fragmentation.

Pointer Reversal (Schorr-Waite) - avoids stack for backing-up on tree edges. Tree edges are explictly reversed to allow retreating later.

Stop-and-Copy - extends graph traversal concept to copy graph from one *workspace* to another, including compacting and redirecting pointers. (Same principles are used for distributed systems for marshalling/serialization)

Generational (Racket default)- separate the heap into several workspaces. Only clean older spaces when younger spaces have little to reclaim. Objects can be moved into older ("tenured") generations.

(available online: P.R.Wilson, "Uniprocessor Garbage Collection Techniques")