CSE 3302 Notes 6: Data Types

(Last updated 7/13/14 7:40 PM)

6.1. Introduction

Types = Means for assuring operations are applied to appropriate objects (values)

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

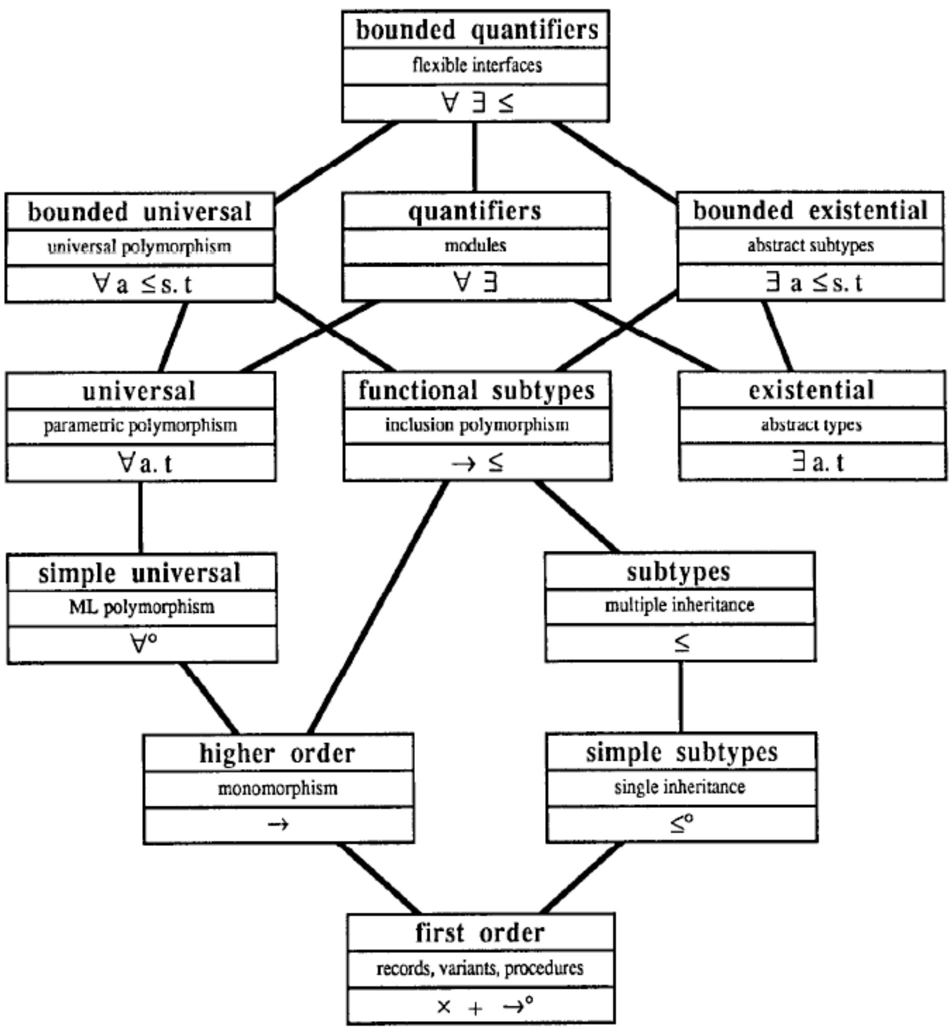

Figure 2. Classification of type systems.

6.2. Primitive Data Types

Numbers . . .

Booleans . . .

JavaScript:

`false` `undefined` `null` `0` `-0` `NaN` `""`

`true` everything else not on previous line

Characters . . .

6.3. Strings

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

6.4. Ordinals

Enumerations

C: Maps to 0 . . .

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

Subranges

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

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

6.5. Arrays

1-d arrays of integers for PL/0: http://ranger.uta.edu/~weems/NOTES3302/LAB3SUM13/

Declarations

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)

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

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

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

The address of `a[i,j,k]` is computed as:

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

(see http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/pascals.pas
routines `arraytyp`, `selector`, and `interpret` codes `20` and `21`)

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 `a[0,0,0]`) is a constant and the second line may be computed as:

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

Run Pascal-S on http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES06/subscript.6.5.pas
and observe array properties and code

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

6.6. Associative Arrays

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 arrays are actually associative arrays, typically implemented by hashing a string:

`arr[1]` and `arr["1"]` refer to the same *property* of object `arr`

`arr[5/2]` and `arr["2.5"]` refer to the same property of object `arr`

BUT, an object is an `array` only if the object was created using:

```
arr=['donut','chips',{a:1,b:'cat'},undefined];
```

```
arr=new Array('donut','chips',{a:1,b:'cat'},undefined);
```

For any object, `object.property` and `object["property"]` are available, BUT . . .

`.property` possibilities are more limited than `["property"]` possibilities

For an array, `length` is one more than the maximum positive integer property name

Set operations may be based on: `for . . . in` as: *loop* (p. 24 of *The Good Parts*), along with: string/index `in` object as *operator* returning `true`/`false` to indicate property presentce

`delete x[propName]` removes the property

See http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES06/objSet.html for manipulation of properties (on simple objects)

JavaScript provides *prototypal inheritance* as a simple *delegation* mechanism:

"In JavaScript, a class is a set of objects that inherit properties from the same prototype object." (D. Flanagan, *JavaScript: The Definitive Guide*, chapter 9, along with 6 and 8)

Several ways to set the *prototype* for an object:

`Object.create()` is usually the simplest

Constructor used with `new` can lead to difficulties (see Crockford book and webpage)

(Browser dependent techniques, including changing the prototype)

`objectName.propertyName = ... ;` can only set (l-value) the property value on `objectName`

... = ... `objectName.propertyName` ... `;` searches the prototype chain (r-value)

`delete` will not follow the prototype chain - it will only remove property from provided object:

`delete` *object-reference*`[`*property-name-as-string*`]`
`delete` *object-reference*`.`*property-name*

object-reference`.has_property(`*property-name-as-string*`])` is the way to check presence (without following prototype chain) before `delete`

Examples:

http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES06/objArray.html demonstrates simple data values as properties

http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES06/quo.html demonstrates use of `new`

http://ranger.uta.edu/~weems/NOTES3302/LAB2SUM13/ demonstrates prototypal inheritance by having instances inherit from "class object" with needed methods. In addition, boxes may inherit drawing properties (colors for fill and stroke, thickness for stroke) from containing box

6.7/6.10. RECORDS AND UNIONS

C - fields are allocated/aligned in order given (http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES06/recsize.c)

```
#include <stdio.h>

typedef struct {
char a,b,c,d;
short e,f;
int g,h;
} compact;

typedef struct {
char a;
short e;
char b;
int g;
short f;
char c;
int h;
char d;
} sloppy;

main()
{
printf("compact %d sloppy %d\n",sizeof(compact),sizeof(sloppy));
}
```

Pascal: (http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES06/dt.pas)

`packed` option to reduce space (for data, but not code)

`with` to abbreviate field selection expressions

"Overlapping" portions of records:

COBOL - `redefines`

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

6.8. Tuples

Similar to records, but no field names - use position to access.

Easily simulated

6.9. Lists

ML (aside) - 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
```

ML uses type inference heavily:

- Declared name-to-type bindings are avoided
  - Gets context/hints from places like:
    - Operations/functions applied (e.g. from libraries)
    - Names that are referenced

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) <=)
```

6.11 Pointers and References

Pointers - familiar

Syntax

Value Model (containers and addresses) - C and Pascal

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

Programmer responsibilities . . .

Garbage Collection (reference model)

Explicit freeing of unneeded space or reachability checking (or reference counts)?

Free lists only or *compact* active memory to remove external fragmentation?

Reference Counts (eager)

Each allocated object has count of pointers to it

Count decremented to zero . . . reclaim

Various schemes to reduce counter update costs

Cycles are a potential problem

Mark-and-Sweep (lazy) - analogous to directed graph traversal techniques (DFS and BFS) to determine reachable heap locations. Requires separate sweep of heap to clean-up external framentation.

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.

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 generations.

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

6.12.-6.15. TYPE CHECKING, STRONG TYPING, TYPE EQUIVALENCE

Concepts

(Types = Means for assuring operations are applied to appropriate objects)

Type definitions

Rules for equivalence, compatibility, and inference (connections to lambda calculus)

Type Checking

Strongly typed = blocks inappropriate application of operation ("safety", no untrapped errors)

Statically = at compile-time

Dynamically = at run-time

How big are the loopholes? ("safety", `void` pointers, memory-to-memory operations)

Equivalence

Structural ("shape")

Components of `record` have same types, order fixed

Issue - array subranges

Name

Strict - aliased types clash

Loose - aliased types are equivalent

ML records

Order of fields doesn't matter

Individual fields must have matching types

```
{ name="Jones", age=25, salary=45000 }
```

```
{ salary=45000, name="Jones", age=25 }
```

Type Conversions and Casts

(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")