CSE 3302 Notes 6: Data Types

(Last updated 10/19/12 12:12 PM)

History of types = History of programming languages

(Cardelli and Wegner, "On Understanding Types, Data Abstraction, and Polymorphism", *ACM Computing Surveys 17 (4)*, Dec. 1985. See section 1.4 and figure 2.)

7.1. TYPE SYSTEMS

Concepts

- Types = Means for assuring operations are applied to appropriate objects
- Type definitions
- Rules for equivalence, compatibility, and inference

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)

Polymorphism

- Weakly typed = many implicit casts ("freedom")
- In general - operation/function may be applied to different types
  - Set of ?
  - Matrix of ?
    - Linear transformation
    - Boolean relation
    - Graph
    - Closed semiring/Kleene algebras

Minimum spanning tree

Shortest paths

Maximum capacity paths

Regular expressions/FSAs

Parametric polymorphism

Implicit = Type inference (ML)

Limited use of declared name-to-type bindings

Gets context/hints from places like:

Operations/functions applied (e.g. from libraries)

Names that are referenced

Unification/pattern matching provides:

Type conflicts

Detection of types restricted to appropriate level

Lists

Equality (and possibly ordering)

Explicit = Classes with type parameters

Containers

Subtype

Variable may refer to any object of the variable's type or the type's subtypes

Approaches to Types

Denotational/mathematical - sets of values

Constructive - Built-in types and constructors for aggregating into tuples/records, sets, etc.

Abstraction - Brings in behavioral issues, information hiding and defined operations

Classifications - mostly review

Pascal sets (general bit vector)

7.2. TYPE CHECKING

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 17 (4)*, Apr. 1988, 189-203. Simulates traditional "unit cancellation".)
- Nonconverting cast in C through pointer casts or `void*` ("universal object reference")

Type Compatibility

- Coercion
- Overloading

Type Inference

- Subranges
- Composite Types
- ML

7.3. RECORDS AND VARIANTS

C - fields are allocated in order given

```
#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));
printf("a %d b %d\n",sizeof(a),sizeof(b));
}
```

Pascal (aside - unpacked and packed)

7.4. ARRAYS

Declarations

Slices - specifiying 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-ro-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
```

But may be simplified 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
```

7.5. Strings

Mutable (C) vs. immutable (Java) strings

7.6. Sets

Book describes implementation of Pascal sets as bit vectors

7.7. Pointers and Recursive Types

Pointers - familiar

Syntax

Reference Model - C

Value Model - Java

Programmer responsibilities . . .

Garbage Collection

Reference Counts

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

Some similarity to *distributed termination detection*

Mark-and-Sweep - 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.

Stop-and-Copy - extends graph traversal concept to copy graph from one workspace to another.

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

7.8. LISTS

ML

```
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)];

- 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
```

Arrays are in the ML “Standard Basis”

Lisp

```
'(a b c d)

expression  (' (a b c d))
value       (a b c d)

'(a 1 c 2.0)

expression  (' (a 1 c 2.0))
value       (a 1 c 2.0)

'(a (1 c) 2.0)

expression  (' (a (1 c) 2.0))
value       (a (1 c) 2.0)

car '((1 2) (3 4) (5 6))

expression  (car (' ((1 2) (3 4) (5 6))))
value       (1 2)

car car '((1 2) (3 4) (5 6))

expression  (car (car (' ((1 2) (3 4) (5 6)))))
value       1

cdr '((1 2) (3 4) (5 6))
expression  (cdr (' ((1 2) (3 4) (5 6))))
value       ((3 4) (5 6))

car cdr car cdr '((1 2) (3 4) (5 6))

expression  (car (cdr (car (cdr (' ((1 2) (3 4) (5 6)))))))
value       4

cons '(1 2 3 4) '(5 6 7 8)

expression  (cons (' (1 2 3 4)) (' (5 6 7 8)))
value       ((1 2 3 4) 5 6 7 8)

cons '(1 2 3 4) '((5 6 7 8))

expression  (cons (' (1 2 3 4)) (' ((5 6 7 8))))
value       ((1 2 3 4) (5 6 7 8))
```

Scheme

```
> (car '((1 2)(3 4)(5 6)))
(1 2)

> (car(car '((1 2)(3 4)(5 6))))
1

> (cdr '((1 2)(3 4)(5 6)))
((3 4) (5 6))

> (car(cdr(car(cdr '((1 2)(3 4)(5 6))))))
4
```

List Comprehensions / Set-Builder Notation / Generators

Three elements:

Generating expression based on index

Mechanism for expressing values of index (e.g. loop or range)

Logical expression indicating indices to be kept

"map" - generating expression is a function applied to each element

"filter" - generating expression is the identity function

"iterator" - values are from a container

Are results provided as entire list (or other container) or through a stream (lazy evaluation)?

Restricting number of results wanted

7.10. EQUALITY & ASSIGNMENT

Fundamental difficulties with equality in logic & mathematics . . .

Notions of *equivalence* may be defined WRT a single function

Is an integer odd or even?

Is a function g in $\Theta(f)$?

What about *equality*?

Has to cover all notions of equivalence

For x and y to be equal, they are indistinguishable to any function

PLs

Shallow equality test - no dereferencing, tests whether values refer to same object?

Deep equality test - dereference and check values (cycles . . .)

ML = deep for *equality* types

Doesn't include `real`

```
- (1,2,3)=(1,2,3);
val it = true : bool

- [1,2,3]=[1,2,3];
val it = true : bool

- beatles=beatles;
val it = true : bool

- beatles=tl(beatles);
val it = false : bool

- [hd(beatles)]=tl(beatles);
val it = false : bool
```

(ML does allow `ref` types which function like pointers)

Scheme

Understand the difference between `eq?`, `equal?`, and `eqv?`

C

Besides comparing pointers with `==` and `!=`, can also use other comparisons (meaningful when dealing with same array, `struct`, etc.)

Pascal

Only allows equality comparisons for pointers

Assignment

Shallow - to one level or just a pointer

Deep - needed for distributed systems

AKA - marshalling, serializ . . .