Data Types

CSE 307 – Principles of Programming Languages

Stony Brook University

http://www.cs.stonybrook.edu/~cse307

Data Types

- We all have developed an intuitive notion of what types are; what's behind the intuition?
  - collection of values from a "domain" (the denotational approach)
  - internal structure of data, described down to the level of a small set of fundamental types (the structural approach)
  - equivalence class of objects (the implementor's approach)
  - collection of well-defined operations that can be applied to objects of that type (the abstraction approach)

Data Types

- Computers are naturally untyped.
- Encoding by a type is necessary to store data:
  - as integer: -1, -396, 2, 51, 539
  - as float: -3.168, 384.0, 1.234e5
  - as Strings: "SBCS" (ASCII, Unicode UTF-16, etc.)
- So how do we know what it means?
  - We associate types with:
    - Expressions
    - Objects (anything that can have a name)
      - Type checking can also be done with user-defined types:

speed = 100 mile/hour

time = 2 hour

distance = speed * time (mile)

distance + 5 miles (ok!)

distance + 5 hours (bad!)

Data Types

- What has a type?
  - things that have values:
    - constants,
    - variables,
    - fields,
    - parameters,
    - subroutines.
    - objects.
  - A name (identifier) might have a type, but refer to an object of a different (compatible type):

    double a = 1;

    Person p = new Student(“John”);

Data Types

- What are types good for?
  - implicit context for operators ("+" is concatenation for Strings vs. integer summation for integers, etc.)
  - type checking - make sure that certain meaningless operations do not occur
    - type checking cannot prevent all meaningless operations
    - It catches enough of them to be useful
- Polymorphism results when the compiler finds that it doesn't need to know certain things

Data Types

- STATIC TYPING means that the compiler can do all the checking at compile time:
  - types are computed and checked at compile time.
- DYNAMIN TYPING: types wait until runtime.
- STRONG TYPING means that the language prevents you from applying an operation to data on which it is not appropriate:
  - unlike types cause type errors.
- WEAK TYPING: unlike types cause conversions.

Data Types

- Examples:
  - Java is strongly typed, with a non-trivial mix of things that can be checked statically and things that have to be checked dynamically (for instance, for dynamic binding):

    String a = 1; compile-time error.

    double d = 10.0;

    int i = d; compile-time error.

  - Python is strong dynamic typed:

    int a = 1

    b = "2"

    a + b run-time error

  - Perl is weak dynamic typed:

    $a = 1

    $b = "2"

    $a + $b no error.

Data Types

- There is a trade-off here:
  - Strong-static: verbose code (everything is typed), errors at compile time (cheap)
  - Strong-dynamic: less writing, errors at runtime
  - Weak-dynamic: the least code writing, potential errors at runtime, approximations in many cases

Data Types

- Duck typing is concerned with establishing the suitability of an object for some purpose.
  - JavaScript uses duck dynamic typing

```
var Duck = function() {
  this.quack = function() {alert('Quaaaaaack!');};
  return this;
};
var inTheForest = function(object) {
  object.quack();
};
var donald = new Duck();
inTheForest(donald);
```

Type Systems

- ORTHOGONALITY:
  - A collection of features is orthogonal if there are no restrictions on the ways in which the features can be combined
  - For example:
    - Prolog is more orthogonal than ML (because it allows arrays of elements of different types, for instance)
  - Orthogonality is nice primarily because it makes a language easy to understand, easy to use, and easy to reason about

What do we mean by type

- Three main schools of thought:
  - Denotational: a type is a shorthand for a set of values (e.g., the byte domain is: {0, 1, 2, ... 255})
    - Some are simple (set of integers),
    - Some are complex (set of functions from variables to values).
    - Everything in the program is computing values in an appropriate set.
  - Constructive: a type is built out of components:
    - int, real, string,
    - record, tuple, map.
  - Abstraction: a type is what it does:
    - OO thinking.

Type Checking

- A type system has rules for:
  - type equivalence (when are the types of two values the same?)
  - type compatibility (when can a value of type A be used in a context that expects type B?)
  - type inference (what is the type of an expression, given the types of the operands?)

$$\frac{a : int \qquad b : int}{a + b : int}$$

Equality Testing

- What should a == b do?
  - Are they the same object?
  - Bitwise-identical?
  - Otherwise the same?
- Languages can have different equality operators
  - Ex. Java's == vs `equals`

Type Casts

- Two casts: converting and non-converting.
  - Converting cast changes the meaning of the type in question.
  - Non-converting casts means to interpret the bits as the same type.
- Type coercion: May need to perform a runtime semantic check. Example: Java references:

```
Object o = "...";
String s = (String) o;
    // maybe after "if(o instanceOf String)..."
```

Type Checking

- The format does not matter:

```
struct { int a, b; }
```

is the same as

```
struct {
        int a, b;
}
```

and

```
struct {
        int a;
        int b;
}
```

Type Checking

- Two major approaches: structural equivalence and name equivalence
  - Name equivalence is based on declarations
  - Structural equivalence is based on some notion of meaning behind those declarations

Type Checking

- Name equivalence: there are other times when aliased types should probably not be the same:

```
TYPE   celsius_temp = REAL,
       fahrenheit_temp = REAL;
VAR    c : celsius_temp,
       f : fahrenheit_temp;
...
f := c; (* this should probably be an error *)
```

Type Checking

- Structural equivalence:

  type R2 = record

  a, b : integer

  end;

  should probably be considered the same as

  type R3 = record

  a : integer;

  b : integer

  end;

Type Checking

- Name equivalence:

```
TYPE new_type = old_type;
```

new_type is said to be an *alias* for old_type.

- Example:

```
TYPE stack_element = INTEGER; (* alias *)
MODULE stack;
IMPORT stack_element;
EXPORT push, pop;
...
PROCEDURE push(elem : stack_element);
...
PROCEDURE pop() : stack_element;
...
```

Type Checking

- Structural equivalence depends on simple comparison of type descriptions substitute out all names
  - expand all the way to built-in types
- Original types are equivalent if the expanded type descriptions are the same

Type Checking

- Types can be discrete (countable/finite in implementation):
  - boolean:
    - in C, 0 or not 0.
  - integer types:
    - different precisions (or even multiple precision),
    - different signedness,
    - Why do we define required precision? Leave it up to implementer.
  - floating point numbers:
    - only numbers with denominators that are a power of 10 can be represented precisely.
  - decimal types:
    - allow precise representation of decimals.
    - useful for money: Visual Studio .NET: decimal myMoney = 300.5m;

Type Systems

- rational types:
  - represent ratios precisely
- complex numbers
- subrange numbers
  - subset of the above (for i in range(1:10))
  - Constraint logic programming: X in 1..100
- character
  - often another way of designating an 8 or 16 or 32 bit integer.
  - Ascii, Unicode (UTF-16, UTF-8), BIG-5, Shift-JIS, latin-1

Type Systems

- Types can be composite :
  - arrays
    - Strings (most languages represent Strings like arrays)
      - list of characters: null-terminated.
      - With length + get characters.
  - sets
  - pointers
  - lists
  - records (unions)
  - files
  - functions, classes, etc.

Record Types

- A record consists of a number of fields:
  - Each has its own type:

```
struct MyStruct {
  boolean ok;
  int bar;
};
MyStruct foo;
```

- There is a way to access the field:

```
foo.bar; <- C, C++, Java style.
bar of foo <- Cobol/Algol style
person.name <- F-logic path expressions
```

Record Types

- When a language has value semantics, it's possible to assign the entire record in one path expression:

```
a.b.c.d.e = 1;
```

- With statement: accessing a deeply nested field can take a while. Some languages (JS) allow a with statement:

```
with a.b.c.d {
    e = 1;
    f = 2;
}
```

- Variant records (a and b take up the same memory, saves memory, but usually unsafe, tagging can make safe again):

```
union {
    int a;
    float b;
}
```

Arrays

- Arrays = areas of memory of the same type.
  - Stored consecutively.
    - Element access = O(1)
  - Two possible layouts of memory:
    - Row-pointer layout,
    - Row-major and Column-major:
      - storing multidimensional arrays in linear memory
      - Example: int A[2][3] = { {1, 2, 3}, {4, 5, 6} };
        - Row-major: A is laid out contiguously in linear memory as: 123456

          offset = row*NUMCOLS + column
        - Column-major: A is laid: 142536

          offset = row + column*NUMROWS
      - Row-major order is used in C, PL/I, Python and others.
      - Column-major order is used in Fortran, MATLAB, GNU Octave, R, Rasdaman, X10 and Scilab.

Arrays

- Row-major generalizes to higher dimensions, so a $2 \times 3 \times 4$ array looks like:

int A[2][3][4] = {{{1,2,3,4}, {5,6,7,8}, {9,10,11,12}}, {{13,14,15,16}, {17,18,19,20}, {21,22,23,24}}};

is laid out in linear memory as: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

- Efficiency issues due to caching.
  - Can effect behavior of algorithms.
- **Row/Column major require dimension to be part of the type.**

Arrays

- Indexing is a special operator, since it can be used as an l-value.
- In languages that let you overload operators, often need two variants:
  - __getindex__ and __setindex__
  - Different number of parameters (row, columns, depth indexes)

Sets

- Set: contains **distinct** elements without order.
- Bag: Allows the same element to be contained inside it multiple times.
- Three ways to implement sets:
  - Hash Maps (without values).
  - When we know # of values, can assign each value a bit in a bit vector.

Maps/Dictionaries

- Map keys to values.
- Multimap: Maps key to set of values.
- Can be implemented in the same way as sets.

Lists

- Prolog-style Linked lists (same with SML) vs. Python-style Array lists:
  - Prolog: matching against lists.
    - Head,
    - Tail.

    Can match more complex patterns:[], [1,2,3], [a,1,X|T].
  - Python lists: Array-lists are efficient for element extraction, doubling-resize

Representation of Lists in Prolog

- List is handled as binary tree in Prolog
  [Head | Tail] OR .(Head,Tail)
  - Where Head is an atom and Tail is a list
  - We can write [a,b,c] or .(a,.(b,.(c,[]))).

Append example in Prolog

```
append([],L,L).
append([X|L], M, [X|N]) :- append(L,M,N).
```

```
append([1,2],[3,4],X)?
```

Append example in Prolog

```
append([],L,L).
append([X|L],M,[X|N]) :- append(L,M,N).
```

```
append([1,2],[3,4],X)?    X=1,L=[2],M=[3,4],A=[X|N]
```

Append example in Prolog

```
append([],L,L).
append([X|L],M,[X|N]) :- append(L,M,N).
```

append([2],[3,4],N)?	
append([1,2],[3,4],X)?	X=1,L=[2],M=[3,4],A=[X|N]

Append example in Prolog

```
append([],L,L).
append([X|L],M,[X|N']) :- append(L,M,N').
```

append([2],[3,4],N)?	X=2,L=[],M=[3,4],N=[2|N']
append([1,2],[3,4],X)?	X=1,L=[2],M=[3,4],A=[1|N]

Append example in Prolog

```
append([],L,L).
append([X|L],M,[X|N′]) :- append(L,M,N′).
```

`append([],[3,4],N′)?`	
`append([2],[3,4],N)?`	`X=2,L=[],M=[3,4],N=[2|N′]`
`append([1,2],[3,4],X)?`	`X=1,L=[2],M=[3,4],A=[1|N]`

Append example in Prolog

```
append([],L,L).
append([X|L],M,[X|N']) :- append(L,M,N').
```

append([],[3,4],N')?	L = [3,4], N' = L
append([2],[3,4],N)?	X=2,L=[],M=[3,4],N=[2|N']
append([1,2],[3,4],X)?	X=1,L=[2],M=[3,4],A=[1|N]

Append example in Prolog

```
append([],L,L).
append([X|L],M,[X|N']) :- append(L,M,N').
```

```
A = [1|N]
N = [2|N']
N'= L
L = [3,4]
```

Answer: **A** = **[1,2,3,4]**

append([],[3,4],N')?	L = [3,4], N' = L
append([2],[3,4],N)?	X=2,L=[],M=[3,4],N=[2|N']
append([1,2],[3,4],X)?	X=1,L=[2],M=[3,4],A=[1|N]

More List Examples in Prolog

```
member(X,[X|R]).
member(X,[Y|R]) :- member(X,R)
```

- *X is a member of a list whose first element is X.*
- *X is a member of a list whose tail is R if X is a member of R.*

```
?- member(2,[1,2,3]).
Yes
?- member(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3 ;
No
```

More List Examples in Prolog

```
select(X,[X|R],R).
select(X,[F|R],[F|S]) :- select(X,R,S).
```

- *When X is selected from [X|R], R results.*
- *When X is selected from the tail of [X|R], [X|S] results, where S is the result of taking X out of R.*

```
?- select(X,[1,2,3],L).
X=1  L=[2,3] ;
X=2  L=[1,3] ;
X=3  L=[1,2] ;
No
```

More List Examples in Prolog

```
reverse([X|Y],Z,W) :- reverse(Y,[X|Z],W).
reverse([],X,X).

?- reverse([1,2,3],[],X).
X = [3,2,1]
Yes
```

More List Examples in Prolog

```
perm([],[]).
perm([X|Y],Z) :- perm(Y,W), select(X,Z,W).

?- perm([1,2,3],P).
P = [1,2,3] ;
P = [2,1,3] ;
P = [2,3,1] ;
P = [1,3,2] ;
P = [3,1,2] ;
P = [3,2,1]
```

Pointers/Reference Types

- Even in languages with value semantics, it's necessary to have a pointer or reference type.

```
class BinTree {
  int value;
  BinTree left;
  BinTree right;
}
```

  - It is value-only (it will be infinitely-size structure).
- The question is, what sort of operations to allow:
  - pointers usually need an explicit address to be taken

```
BinTree bt1;
BinTree bt2;
BinTree *foo = &bt1;
```

Pointers/Reference Types

- Pointers tend to allow pointer arithmetic: foo += 1
  - Only useful when in an array.
    - Leave the bounds of your array, and you can have security holes.
  - Problem: Can point to something that isn't a BinTree, or even out of memory.
- In Java, references are assigned an object, and don't allow pointer arithmetic.
  - Can be NULL.

Files and Input/Output

- Input/output (I/O) facilities allow a program to communicate with the outside world
  - interactive I/O and I/O with files
- Interactive I/O generally implies communication with human users or physical devices
- Files generally refer to off-line storage implemented by the operating system
- Files may be further categorized into
  - temporary
  - persistent