CS342: Organization of Prog. Languages

Topic 14: Type systems

- Dynamic and static typing
- Type inference
- OO as a blend of static and dynamic typing
- Some basic types and constructors
- Mutable vs immutable types
- Extensible vs closed systems
- Data abstraction
- ADT vs OO.
- Subtypes
- Dependent types

Why Types?

- Values are ultimately stored in a computer as sequences of bits.
- A given sequence of bits can have different interpretations when used by different programs in different ways.
- For example, on one computer the 32 bits

  `01000101011010000011111100000000`

  can represent
  - the integer `1164459776`,
  - the floating point number 3.7159375×10^3,
  - the character string `"Eh?"`.
- A type provides an interpretation of binary data as a value which a program can manipulate.
- Different programming languages have different ways to associate types with data.

Dynamic vs Static Typing

- In Scheme, we have seen that a variable may hold values of different types at different points in the program.

  ```
  (let ((x #f))
     (set! x #\A)
     (set! x 32)
     (set! x "Hello")
     (set! x '(1 2 3)))
  ```

  Each value has some information associated to it which allows a program *to determine its type at execution time*.

  This is known as **dynamic typing**.
  (Sometimes people call this "weak typing.")

- In ANSI C, you have seen that a variable must be *declared to contain values of only one specific type prior to execution*.

  This is known as **static typing**.
  (Sometimes people call this "strong typing.")

Benefits of Static Typing

- Certain errors are caught earlier.
- Extra info => programs more self-documenting.
- Extra info => compilers can produce code with fewer tests (and therefore more efficient code).

Benefits of Dynamic Typing

- (Practical) Small programs easier to write.
- (Practical) Same program can be used for values of different types, provided the operations are available on the arguments.

```
(define (saytwice a)
   (write a)
   (write a)
   (newline))

(saytwice "hip")
(saytwice 30)
```

  This is called *polymorphism.*

- (Theoretical) Some “higher-order” programs are impossible to type statically in the usual type systems.

  E.g. consider the type of a function which is called with itself as an argument... `(f f)`.

Implementation of Dynamic Typing: Tagged Values

- Associate with each value extra info giving the value's type. E.g.

```
[0x01 | 0x45 0x68 0x3F 0x00] <=> [Int32   | 1164459776 ]
[0x02 | 0x45 0x68 0x3F 0x00] <=> [Float32 | 3.7159375e3]
...
```

- Operations test the types of their arguments at run time to determine what to do. E.g.

```
Plus(a, b) {
        switch (typeof a) {
        case Int32:
                switch (typeof b) {
                case Int32:    ...
                case Float 32: ...
                ...
                }
        case Float32:
                switch (typeof b) {
                case Int32:    ...
                case Float 32: ...
                ...
                }
        ...
}
```

- Most implementations represent all values as pointers to chunks of tagged memory.
- Can still have declarations in the language, and make use of that information to avoid certain tests or check for run-time errors.

Implementation of Static Typing: Type Inference

- Want efficient use of type information:
  `=>` avoid type tests at run time
   `&` avoid storing type tags
  `=>` determine choices at compile time

- **Type inference** determines the types of all intermediate subexpressions.
- *Overloaded* operators and functions give *many different implementations* for the same name.

  Often used in languages with static typing.
- **Overload reslolution** determines *which* implementation is required in each case.
- This is done *before* program execution.

Type inference and overload resolution

- Suppose we have overloaded meanings for:

```
+: (int, int) -> int
+: (string, string) -> string
length: string -> int
length: int -> int
```

and the following delcarations

```
int i, j;
string a, b;
```

- Then types for intermediate expressions can be determined bottom up.

  E.g., using @I to represent type `int` and @S for `string`

```
length(a + b) + length(i + j)

Use declaration info
-> length(a@S + b) + length(i + j)
-> length(a@S + b@S) + length(i + j)

Choose +: (S,S)->S, length: S->I
-> length({a@S + b@S} @S) + length(i + j)
-> length({a@S + b@S} @S)@I + length(i + j)

Use declaration info
-> length({a@S + b@S} @S)@I + length(i@I + j)
-> length({a@S + b@S} @S)@I + length(i@I + j@I)

Choose +: (I,I)->I, length: I->I
-> length({a@S + b@S} @S)@I + length({i@I + j@I}@I)
-> length({a@S + b@S} @S)@I + length({i@I + j@I}@I)@I

Choose +: (I,I)->I
-> {length({a@S + b@S} @S)@I + length({i@I + j@I}@I)@I}@I
```

- We now have determined types and meanings for all the identifiers in the program:

```
length_1(a +_1 b) +_3 length_2(i +_2 j)

+_1  is (S,S)->S   length_1 is S -> I
+_2  is (I,I)->I   length_2 is I -> I
+_3  is (I,I)->I
```

- We have completed *type inference* and *overload reslolution*.

More issues in type inference

- Inserting automatic coercions:

  ```
  strlen("Spam spam spam spam")/ 3.14159
  ```

  Need to insert conversion `int -> double` .

- Some languages (e.g. Java, C++) require that overloaded operators cannot differ only in return type.

```
foo: (A, B) -> C
foo: (A, B) -> D
```

This is so that type inference can be done in one pass.

Other languages allow disambiguation by return type.

```
*: (Vector, Vector) -> Vector  // cross product
*: (Vector, Vector) -> Real    // dot product
```

This requires type inference to consider *sets* of possible types.

Advanced Topic:
OO as a midway between static and dynamic typing

- Object-oriented programming allows us to know some information about an object early
  - it belongs to a certain base class,
  - it has certain methods
- Other information is known at until run time
  - which derived class does the object really belong to
  - what are the implementations of the methods.

- In this case, values are typically represented as

```
[ Ptr-to-class-info | Data Data Data ... ]
```

- This is a dynamically typed value, where the tag is a pointer to class info.
- The class info can contain the class name, a table of of methods, etc.
- In C++ implementations this table of methods is called the *vtbl*, for “virtual function table.”

Some Basic Types

These appear as basic types in many languages:

- Void.
  - An empty type. I.e. has no members.
  - Usefull in some languages where a function must always (formally) return a value.
- Singleton.
  - A type with only one element.
  - *E.g.* `'()` in Scheme or **nil** in Pascal.
  - Some languages have multiple singleton types with different uses.
- Boolean.
  - Usually stored in variables and structures as a byte, not a bit.
  - Special arrays of boolean will pack to bits.

- Integer.
  - can be signed or unsigned (different machine instructions).
  - can have various lengths (8, 16, 32, 64 or unlimited bits).
  - can be represented internally as power of 2 or power of 10 base (bigints, bcd).
- Floating point number.
  - Different lengths represented directly in hardware.
  - IEEE standard very common now.
  - Software implementation of arbitrary precision.
  - Some antique formats IBM, Vax, Cray.

- Character.
  - Interpret small numbers as letters, numerals, punctuation.
  - Different encodings, different widths (ASCII, EBCDIC, Unicode).
  - Surprisingly, encoding is not determined by hardware.

- String.
  - Some programming languages make strings a primitive type.
  - Strings of multi-byte characters can be represented either with characters of fixed width or varying width.

- I/O ports or Files.
  - These are primitive in some languages
  - Provide by libraries in others
  - Wide variation in how they are handled

Basic Type Constructors

- Record/Struct.
  - Heterogeneous collection of several "fields."
  - Associated field selectors `Record(a:A, b:B, c:C)`
  - Model as a "*Cartesian product type*" $A \times B \times C$
  - Some languages allow trailing arrays of varying size, e.g. C.

```
struct A {
        int      i;
        double   d;
        struct X x[1];
};
...

struct A *a = (struct A *) malloc(sizeof(struct A) + (N-1)*sizeof(struct X));

xi = a->x[3];
```

  - Some languages allow "variant" records, with different sets of fields, *e.g.* Pascal, Ada. We shall discuss these under "Union."

- Array.
  - Homogeneous collection of several values.
  - Formally, we can model an array of size n of values of type T as having type T^n, the n-fold Cartesian product: $T \times T \times \cdots \times T$.
  - Some languages allow very fancy operations to take slices of multi-dimensional arrays along different axes, concatenate them, etc, *e.g. APL*.

In different systems, the number of elements can be fixed statically or specified dynamically.

Minimal, static arrays:

– Simply several values placed consecutively in memory, *e.g.* `Blob *` in C.

– Can be implemented to allow specification of lower and upper index bounds, *e.g.* `a[3..4]` Pascal, `a(1:10)` Fortran.

Arrays with dynamic information:

– Some languages allow the same array object to change size over time, *e.g.* Common Lisp, Java. This requires storage of run-time info in an array header.

– In dynamic languages homogeneous arrays can be implemented efficiently by storing *one* type tag for *all* the elements in the array header, e.g. arrays of floating point numbers, bits, or characters.

```
;; Common Lisp
(make-array
    (list N)
    :element-type 'string-char
    :initial-element #\Space )
```

- Union.
  - This allows an object to store values of different types.
  - This corresponds to "variant records" in Pascal and Ada, to "union types" in C, etc.
  - Typically allow alternative to be selected by name: `Union(a: A, b: B)`.
  - May be "tagged" or "untagged."

    A tagged union is a structure with a special discriminant field to indicate which is the correct interpretation for the data at the moment. This allows for run-time checks when accessing data from the union.

    An untagged union does not have this special field: it is up to programmer discipline to keep track of what is in use. Errors arise when one kind of data is put in, and data is taken out with the other interpretation.

- Functions.
  - In general a function takes values from one type and returns values in another: $D \to R$.
  - In practice, it is useful to allow functions to take several arguments and return several results, rather than always use functions on records.

    $$(D_1, D_2, ..., D_n) \to (R_1, R_2, ..., R_m)$$

    vs

    $$D_1 \times D_2 \times \cdots \times D_n \to R_1 \times R_2 \times \cdots \times R_m$$

    In this formulation a function can return 0, 1 or more results.

Mutable vs Immutable Types

- If an object can be modified in-place, we say it is *mutable*.

  If an object is constant, we say it is *immutable*.
- Immutable values can be placed in ROM, or can be shared without worrying about side-effects.
- Mutable values need to be carefully managed to make sure a computation is consistent. E.g. Modifying a key string of a hash table can cause entries to become unreachable.
- Some programming languages allow declarations for immutable values, e.g.

```
const char *fmt = "The %s in %s falls mainly on the %s.\n";
printf(fmt, precipitation, country, region);
fmt[7] = 'X';  /* ERROR */
...

>> gcc foo.c
foo.c: In function ‘main’:
foo.c:9: warning: assignment of read-only location
```

Subtypes

- Somtimes it is useful to consider a subset of values from a type.
- These might have useful closure properties for certain operations:

```
+: (PositiveInt, PositiveInt) -> PositiveInt
```

- They might be useful to guarantee certain errors do not occur:

```
vector-ref: (Vector, PositiveInt) -> Object
```

- However, in these cases we do not want the `PositiveInts` to be excluded from `Int` operations.
- We want to think of the values to belong to a “subtype.”

Subtypes and Properties

- If S is a subtype of T, we write $S < T$.
- Normally a subtype has an associated property, or “predicate,” which can test whether a value is in the subtype or not.

```
PositiveInt = { n | n > 0 and n in Int }
```

  Some people use the notation

```
positive? n == n > 0;

PositiveInt == Subtype(positive?, Int)
```

Subtypes and Representation

- Subtyping can be realized in two different ways in programming languages:

  1. subtypes which have the same representation as the base type
  2. subtypes which have a different representation than their base type

- In the first case, the programming language has to ensure that the right properties hold at the right points in the program. E.g.

```
^: (Int, PositiveInt) -> Int;

n, m: Int;

n := 7; m :=  8;  n^m;  -- OK
n := 7; m := -3;  n^m;  -- ERROR
```

- In the second case, the implementation has to insert conversions as needed. E.g. in C, from `short` to `long`.

Subtypes and Constructors

- If we have $S < T$ and $X < Y$ what can we say about relationships like

$$\text{Record}(f1 : S, f2 : X) \quad vs \quad \texttt{Record}(f1 : T, f2 : Y)$$

- In the most general setting, we cannot presume any subtype relationship is induced on the constructed types.
- Under special circumstances, we can say more however.

Subtyping of Arrays, Lists, ,etc

- If a homogeneous aggregate type Agg is immutable, and $S < T$, then

  $$\text{Agg}(S) < \text{Agg}(T)$$

  E.g.

  $$\text{ArrayPositiveInt} < \text{ArrayInt}$$

  So we can write the following:

```
printarr(a: Array Int) {
    for e in a repeat print e;
}

ai: Array Int := [-1, 0, 1];
ap: Array PositiveInt := [1, 2, 3, 4];

printarr(ai);  -- OK
printarr(pi);  -- OK
```

- If the aggregate type is *mutable* however, then we can have no such relationship.

  This is because an Agg(T) operation applied to an Agg(S) object, might destroy its Agg(S) property.

  E.g.

```
modarr(a: Array Int) {
    a[1] := -2;
}

modarr(ai); -- OK
modarr(pi); -- Whoops!  pi now contains -2
```

Subtyping of Records

- This is the same as for arrays, etc. If the records do not support any updating operations, and $S < T$, then

$$\text{Record}(A_1, ..., S, ..., A_n) < \text{Record}(A_1, ..., T, ..., A_n)$$

Subtyping of Functions

- For pure functions, if $A \geq B$ and $S \leq T$, then

$$A \rightarrow S \quad \leq \quad B \rightarrow T$$

- Notice that the subtyping on the arguments is the opposite of the subtyping on the return values.
- We say that the result subtyping relationship is *covariant* and the argument subtyping relationship is *contravariant*.

The Subtype Lattice

- If B_1 and B_2 are both a subtypes of B_0, then we may consider for the sets of values $B_1 \cap B_2$ and $B_1 \cup B_2$.
- Given a number of subtypes B_i of B_0, we can consider all possible unions and intersections. This forms a *subtype lattice*. These can be expressed in terms of the subtype predicates $P_i(x)$:

$$\begin{aligned} B_i &= \{x \mid x \in B_0 \land P_i(x)\} \\ B_i \cup B_j &= \{x \mid x \in B_0 \land \Big(P_i(x) \lor P_j(x)\Big)\} \\ B_i \cap B_j &= \{x \mid x \in B_0 \land \Big(P_i(x) \land P_j(x)\Big)\} \end{aligned}$$

- The subtypes $B_i \cup B_j$ are union types in which an element may be in either or both branches.
  For this reason the union of two types (described earlier) is sometimes written using the mathematical symbol for *disjoint union* $S \uplus T$.
- The subtypes $B_i \cap B_j$ are very useful, because the elements lie in *both* types, can participate in both sets of operations, and satisfy both sets of closure properties.

Derived classes as subtypes

- Just as classes can be associated with types, derived classes can be associated with subtypes.
- Single inheritance (*e.g.* Java) gives chains of subtypes:

$$B_0 > B_1 > B_2$$

```
// Java
class B0 { ... }
class B1 extends B0 { ... }
class B2 extends B1 { ... }
```

- Multiple inheritance (*e.g.* C++) gives intersection subtypes:

$$B_I = B_1 \cap B_2$$

```
// C++
class B1 { ... };
class B2 { ... };
class BI : public B1, B2 { ... };
```

An *intersection subtype* has the *union* of the *methods and fields*.

Dependent Products

- A generalized form of product type can be defined, where the *type* of the second component *depends on the value* of the first: $(a : A) \times B(a)$.
- This is called a **dependent product** type.

$$(n : \text{Integer}) \times \text{Char}[n]$$

$$(T : \text{Type}) \times T^2$$

$$(n : \text{Integer}) \times (T : \text{Type}) \times T^n$$

- Sometimes these are implemented as classes or records so all components have names:

```
Record(n: Integer, s: Char[n]);
```

Dependent Functions

- A **dependent function** type is a generalized function type where the *type* of the result depends on the *value of an argument*.

$$(a : A) \to B(a)$$

- E.g.

$$\text{identity} : \ (n : \text{Integer}) \to \text{SquareMatrix}(n, \text{Integer})$$

$$\text{mod} : \ (\text{Integer}, m : \text{Integer}) \to \text{IntegerMod}(m)$$

- Like OO, **dependent types** allow a mid-point between dynamic typing and static typing. In fact, OO can be implemented with dependent types.