Values and Types

Types of values

Type equivalence, compatibility, & inference

Main ideas

- A type is a set of values, equipped with one or more operations that can be applied uniformly to all those values

- Inclusion of data types in a language definition supports:
  - readability, writability, and portability

- A type system includes
  - Type inference rules to infer an object's data type from the available information
  - A type equivalence algorithm for determining whether two objects are of the same type

Types

- A type is a set of values, equipped with one or more operations that can be applied uniformly to all those values

- How to categorize values
  - Primitive
  - Composite
  - Pointers
  - References
  - Functions/procedures

- Different PLs support different types of values. Why?

Primitive types

- A **primitive type** is one whose values can't be decomposed into simpler values.
- Typically supported directly by the hardware – implications for
  - Efficiency
  - Storage
- Includes:
  - Boolean
  - Character
  - String
  - Integer
  - Float
  - Numeric data type ranges
- Names of types vary from one PL to another; not significant

Boolean

- Boolean = {false, true}

- Not always a built-in type
  - Ex in C: 0 = false, non-zero = true

```
x = 5;
while ( x-- ) printf("x is %d", x );
```

- Storage
  - Only need 1 bit, but…
  - Memory addresses are larger than that

- Operations: support short-circuiting?

Integers and floats

- Integer = {..., -2, -1, 0, 1, 2, ... }
- Float = {... -1.0, ..., 0.0, ..., 1.0 ... }

- Implementation issues:
  - Different types for different sizes
  - Internal representation: 2's complement, IEEE 754
  - Range is hardware dependent, but language must help determine upper/lower bounds
  - Roundoff
- Reals: fixed point vs. floating point support
  - Fixed point has fixed number of digits after decimal
  - Floating point, decimal can 'float' relative to significant digits

Defined numeric data types

- Subrange type: a contiguous subset of a simple type
  - Base type: the type of elements in the subrange
  - In Ada and Pascal we can define new numeric types by specifying a range

    Ex in Ada: `type Population is range 0 .. 1e10;`

- Many languages support defining new enumeration types by listing their explicit values (called enumerands)
  - Underlying representation usually mapped to integers
  - Ex in Ada: `type Color is (red, green, blue);`

Characters and strings

- Character = {… ‘A’, …, ‘Z’, …, ‘0’, …, ‘9’, … }
- Some languages support a character-string type
  - Ex: ML, Prolog, Java
- Others support a character type with strings stored explicitly as an array of characters
  - Ex: C, Pascal, Ada
- Issues:
  - Allowable character set and collating sequence (order of characters)
    - Ex: EBCDIC, ASCII, ISO-Latin, Unicode
    - Ex: EBCDIC has lower case < upper case < numbers
    - Ex: ASCII has numbers < upper case < lower case
  - Representation
    - Null terminated complicates size (Ex: C string)
    - Limit on string size with length field

Pointers (?)

- Language support features
  - Null value
  - Allocation & deallocation operations
    - Implications for underlying memory management support
  - Dereferencing

- Issues
  - What can a pointer point to?
    - Restricted by type? `int x, *iptr = &x;`
    - Type compatibility issues?
    - “Generic” pointer? `void *genericPtr;`
  - Dangling pointer problem: a pointer that points to storage that has been deallocated

Composite types (data structures)

- Use type constructors to define new data structures
- Attributes of specifying data structures:
  - Number of components
    - Is there an upper bound?
    - Can the number change or is it fixed statically?
  - Type of each component
    - Homogenous (components are the same)
    - Heterogenous (components differ)
  - Component selection mechanism
    - Whole or part access?
  - Component organization
  - Composite type allocation and deallocation

Composites: structures (records)

- Defined with type constructors
- Can be understood in terms of cartesian products
- For example, in C:

```
struct myRec {
    type1 a;
    type2 b;
    type3 c;
  };
```

Domain(myRec) = Domain(type1) x Domain(type2) x Domain(type3)

```
struct myRec theStruct, rec2; // initialization allowed?
type1 n = theStruct.a;
rec2 = theStruct;   // should this be allowed? More later!
```

Composites: unions (variant records)

- Can be understood in terms of disjoint union
- For example, in C:

```
union myVariant {
    type1 a;
    type2 b;
    type3 c;
  }
```

Domain (myVariant) = Domain(type1) + Domain(type2) + Domain(type3)

- Space for the fields is **shared**

Composites: unions

- **Discriminated union**
  - Tag is attached to each field of the union
  - Can be checked at run time to determine the type stored in the union

- **Undiscriminated union** (or **free union**)
  - No tag
  - Program must provide other ways to ensure that values of the correct type are accessed
  - Possible to store a value of one type and inadvertently (or intentionally?) retrieve the “value” as another type

Example: Pascal Discriminated Union

```
type paytype = (salaried, hourly);
var employee : record
  id : integer;
  dept : array [1..3] of char;
  age : integer;
  case payclass : paytype of
  salaried : (monthlyRate : real;
       startDate : integer);
  hourly : (ratePerHour : real;
       regHours : integer;
       overtime : integer);
 end;
```

Type tag

Mappings

$m : S \rightarrow T$, m is a **mapping** from every value in S to every value in T

- Arrays (finite; ordered index set)
  - One or multi-dimensional
- Hashes (finite; unordered index set)
- In Pascal:

```
type Color = ( red , green , blue ) ;
Pixel = array ( Color ) of 0 . . 1;
```

- Functions (procedures)
  - Note: Ada uses the same notation for array accesses and function calls
- Sets? In Pascal:

```
type Color = ( red, green, blue );
Hue = set of Color;
```

Recursive types

- A **recursive type** is one defined in terms of itself
- Example: List
  - a sequence of 0 or more component values.
  - **length** = number of components.
  - **empty list** has no components.
  - A non-empty list consists of a **head** (its first component) and a **tail** (all but its first component).

- Type declaration for integer-lists in Haskell

```
data IntList = Nil | Cons Int IntList
```

Type Equivalence

Determines when two types are "equivalent" for purposes of some operation

The problem of determining type equivalence raises two related ideas:

- What does it mean to say that two types are the "same"?
  - A data type issue
- What does it mean to say that two data objects of the same type are "equal"?
  - A semantic issue

Structural equivalence

- $T_1 \equiv T_2$ if and only if T_1 and T_2 are built in the same way using the same type constructors from the same simple types

- Some issues:
  - Must the names of the fields be the same or is it enough that the structures contain the same number and type of components?
  - Consider:

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

```
struct bar {
  int c;
  char d;
};
```

```
struct tip {
  char d;
  int c;
};
```

  - Are foo and bar equivalent? How about tip?

Structural equivalence

- Structural equivalence does not mean that the two types *mean* the same thing.

- For example (Pascal): Is `len + vol` meaningful?

```
type
   Meters = integer;
   Liters = integer;
var
   len : Meters;
   vol : Liters;
   age : integer
```

Name Equivalence

- $T_1 \equiv T_2$ if and only if T_1 and T_2 were defined in the same place.

- Example: Which of `f1, f2, b1, b2` are equivalent under name equivalence? Under structural equivalence?

```
typedef struct foo {
  int a;
  char b;
} foo_t;

typedef struct bar {
  int a;
  char b;
} bar_t;

foo_t f1, f2;
bar_t b1, b2;
```

Name Equivalence

- $T_1 \equiv T_2$ if and only if T_1 and T_2 were defined in the same place.

- Example: Which of `f1, f2, b1, b2` are equivalent under name equivalence? Under structural equivalence?

```
typedef struct foo {
  int a;
  char b;
} foo_t;

typedef struct bar {
  int a;
  char b;
} bar_t;

foo_t f1, f2;
bar_t b1, b2;
```

under name equivalence:
`f1, f2` are equivalent
`b1, b2` are equivalent

under structural equivalence:
`f1, f2, b1, b2` are equivalent

Name Equivalence

- Anonymous types cannot be used. For example:

```
var x : array [1..10] of integer;  /* Ex. 1 */
    y : array [1..10] of integer;
```

- Here the **variables** are names, but the **types** are not
- x and y are structurally equivalent, but not name equivalent

- A similar, but more ambiguous, problem occurs with

```
var x, y : array [1..10] of integer; /* Ex. 2 */
```

Ada solves this problem by saying that, in a case like this, it is as if we had used the separate definitions given above in Ex. 1, so the two variables are not type equivalent.

Declaration Equivalence

- Types that lead back to the **same original structure declaration** by a series of re-declarations are considered to be equivalent types.
- By this rule, x&y in Ex. 1 are *not* equivalent, but they are in Ex. 2.
- Example:

```
type  t1 = array [1..10] of integer;
      t2 = t1;
      t3 = t2;
```

- Which are type equivalent under declaration equivalence?

  All of them

Example

```
type t1 = array [1..10] of integer;
     t2 = t1;
     t3 = array [1..10] of integer;
var  x : t1;
     y : t2;
     z : t3;
     w,v : array [1..10] of integer;
```

There are *three* different types here:
t1, t2, t3, and the unnamed type associated with w and v.

What is their equivalence under the three strategies?

Example

```
type t1 = array [1..10] of integer;
     t2 = t1;
     t3 = array [1..10] of integer;
var  x : t1;
     y : t2;
     z : t3;
     w,v : array [1..10] of integer;
```

There are *three* different types here: t1, t2, t3, and the unnamed type associated with w and v.

What is their equivalence under the three strategies?

<u>under structural equivalence:</u>

`x,y,z,w,v` are equivalent

<u>under name equivalence:</u>

`w,v` are possibly equivalent if we allow that they are defined for the same anonymous type (but most languages classify as separate types)

<u>under declaration equivalence:</u>

`x,y` are equivalent

`w,v` are equivalent

Type Compatibility

When can a value of one type be used in a context that expects another type?

- Where is this an issue?
  - Use of a value in some operation
  - Assigning a value to a variable
  - Passing a value as a parameter
- Primitives: create a type hierarchy based on principle "loss of information"
- Non-primitives?

Type Inference

What is the type of an expression, given the types of the operands and possibly the surrounding context?

An **expression** is a construct that will be evaluated to yield a value.

- Literals
- Variables and constants
- Conditionals
- Iterative expressions
- Function calls

Type Completeness Principle

- **Type Completeness Principle**: No operation should be arbitrarily restricted in the types of its operands
  - More special cases to learn creates more difficulty to program correctly

- First-class values
  - Can be stored arbitrarily into variables and constants
  - Can be passed into a function and returned from a function
  - Can be created dynamically at run time
  - Ex: Java object
- Second-class values
  - Can be passed as a parameter, but not returned from a subroutine or assigned to a variable
  - Ex: subroutines are 2nd class in most imperative languages, 1st class in functional languages
- Note: categories are somewhat loose and often used comparatively