CS301 W'99 Lecture Notes
Lecture 13

Values and Types

Values are the entities or objects manipulated by programs.

We divide the universe of values according to **types**.

We characterize types by:

• a set of values.

• a set of operations defined on those values; and/or

• a set of valid contexts for those values.

(In particular, values that can be anonymously constructed, used in expressions, passed to and from procedures, and assigned into variables are called **first-class** values.)

• How values are **represented** and operations are **implemented**.

• How literal values are described.

Examples:

Integers with the usual arithmetic operations.

Booleans with operators `and,or,not` and valid as arguments to conditional operations.

Arrays with operations like fetch and store.

Sets with operations like membership testing, union, intersection, etc.

Hardware Types

Machine language doesn't distinguish types; all values are just bit patterns until **used**. As such they can be loaded, stored, moved, etc.

But certain **operations** are supported directly by hardware; the operands are thus implicitly typed.

Typical hardware types:

• **Integers** of various sizes, signedness, etc. with standard arithmetic operations.

• **Booleans** with boolean and conditional operations. (Usually just a special view of integers.)

• **Floating point** numbers of various sizes, with standard arithmetic operations.

• **Characters** with i/o operations.

• **Pointers** to values stored in memory.

• **Instructions**, i.e., code, which can be executed.

• Many others are possible, e.g., binary coded decimal.

Details of behavior (e.g., numeric range) are **machine-dependent**, though often subject to **standards** (e.g., IEEE floating point, ASCII characters).

Primitive (Atomic,Basic) Values and Types

Primitive values cannot be further broken down by user-defined code; they can be managed only via operators built into the language.

Typical primitive types include integers, floats, characters, booleans, enumerations, etc.

Usually closely allied to hardware types.

Example: booleans. Note that in most languages (except C/C++), this is a **different** type from integers, even though boolean values may be **represented** internally by integers.

Numeric types only **approximate** behavior of true numbers. Also, they often inherit machine-dependent aspects of machine types, causing serious **portability** problems.

Example: Integer arithmetic in most languages.

Partial counter-example: Numerics in Lisp.

Composite Values

Composite values are **constructed** from more primitive values, which can usually later be **selected** back from the composite, and perhaps selectively **updated**.

Example: Records (C Syntax)

```
struct emp {
    char *name;
    int age;
};

struct emp e = {"Andrew",99};

if (strcmp(e.name,"Fred")) ...;

e.age = 88;
```

In **statically typed** languages, it is generally necessary to declare new composite types (e.g., `struct emp`) before defining composite values (e.g., `e`). The type definition indicates how the type is constructed from more primitive types, using one of a few predefined **type constructors** (e.g., `struct`).

Static and Dynamic Typing

HLL's differ from machine language in that explicit types appear and type violations are ordinarily caught at some point.

Static typing is most common.

• Types are associated with identifiers (esp. variables, parameters, functions).

• Can be statically checked, if language and compiler allow.

• Compiler can optimize representations of values used at runtime.

Dynamic typing occurs in Lisp, Scheme, Smalltalk, VB, JavaScript, etc.

• Types are attached to values (usually implicitly).

• The type associated with identifiers can vary.

• Correctness of operations can't generally be checked until runtime.

• Optimized representation hard.

Static typing offers the great advantage of **catching errors early**, and generally supports more efficient execution.

Flexibility of Dynamic Typing

Why ever settle for dynamic typing?

• Simplicity. For short or simple programs, it's nice to avoid the need for declaring the types of identifiers.

• Flexibility. Dynamic typing allows **container** types, like lists or arrays, to contain mixtures of values of arbitrary types.

Note: Some statically-typed languages (e.g., Standard ML) offer alternative ways to achieve these aims, via **type inference** and **polymorphic typing**.

Dynamic Typing Example

Consider a function that reads and returns an integer or string literal.

```
function readliteral();
  begin
    read a string of nonblanks;
    if (string constitutes an integer literal) then
      return numeric-value-of-string;
    else
      return string;
  end;

(* read dates: year month day *)
y = read();
m = readliteral();
d = read();
if (m >=1) and (m <= 12) then (* do nothing *)
else if m = "JAN" then m = 1
else if m = "FEB" then m = 2
else ...
```

Type Constructors

Programmers usually define composite types in order to implement **data structures** appropriate to an application and/or algorithm.

Abstractly, such data structures can be seen as mathematical operators on underlying **sets** of simpler values. A small number of type operators suffices to describe most useful data structures:

- Cartesian product ($S_1 \times S_2$)
- Disjoint union ($S_1 + S_2$)
- Mapping (by explicit enumeration or by formula) ($S_1 \rightarrow S_2$)
- Set ($\mathcal{P}^S$)
- Sequence (S^*)
- Recursive structures (lists, trees, etc.)

Concretely, each language defines the internal **representation** of values of the composite type, based on the type constructor and the types used in the construction.

Example: The fields of a record might occupy successive memory addresses (perhaps with some alignment restrictions). The total size of the record is (roughly) the sum of the field sizes.

Often a range of representations are possible, from highly packed to highly indirected. There's often a tradeoff between space and access time.

Representation of Data Structures

Historically, most languages provide direct representations only for a few data structures, usually those whose values can be represented **efficiently** on a conventional computer. Often, they are restricted so that all values will be of **fixed size**.

For conventional languages, this is the short list:

- **Records.**
- **Unions.**
- **Arrays.**

Many languages also support manipulation of **pointers** to values of these types, in order to allow moving data "by reference" and to support recursive structures; more later.

Records = Cartesian Products

Records, tuples, "structures", etc. Nearly every language has them.

"Take a bunch of existing types and choose one value from each."

Examples (Ada Syntax)

```
type EMP is
  record
    NAME : STRING;
    AGE : INTEGER;
  end record;

E: EMP := (NAME => "ANDREW", AGE => 99);
```

(ML syntax):

```
type emp = string * int  (unlabeled fields)
val e : emp = ("ANDREW",99);

type emp =
   {name: string, age: int}  (labeled fields)
val e : emp = {name="ANDREW",age=99};
```

ML also permits record values to be written without declaring explicit named type first.

Records (continued)

Standard operations: construction, selection, selective update.

Representation: Usually as described above. Because records may be large, they are often manipulated by reference, i.e., represented by a pointer. The fields within a record may also be represented this way.

Allowed contexts: In many languages, treated like primitive values, e.g., can be assigned as a unit, passed to or returned by functions, etc. But since they may be large, some languages add restrictions.

Literals: Most languages allow a literal record to be specified by specifying each component, either by position or by name. (But C doesn't permit literals except as initializers.) Some languages require components to be initialized after creation.

Disjoint Unions

Variant records, discriminated records, unions, etc.

"Take a bunch of existing types and choose one value from one type."

Pascal Example:

```
type RESULT = record
                case found : Boolean of
                  true: (value:integer);
                  false: (error:STRING)
                end;

function search (...) : RESULT;
...
```

Generally behave like records, with **tag** as an additional field.

Represented by the variant's representation, usually plus a tag (thus forming a record). Size typically equals the size of the largest variant plus tag size.

Variant Insecurities

Pascal variant records are **insecure** because it is possible to manipulate the tag independently from the variant contents.

```
tr.value := 101;
write tr.error;

if (tr.found) then begin
   ...
   tr := tr1;
   x := tr.value
```

These problems were fixed in Ada by requiring tag and variant contents to be set simultaneously, and inserting a runtime check on the tag before any read of the variant contents.

Non-discriminated unions

C unions don't even have a tag mechanism: the programmer must provide the tag separately:

```
union resunion {
    int value;
    char *error;
};
struct result {
  int found;  /* boolean tag */
  union resunion u;
}

struct result search (...);
```

The tag need not be tightly associated with the union:

```
int search (union resunion *r,...);

union resunion res;
if (search(&res)) {
  ...res->value...
} else {
  ...res->error...
}
```

This might permit more efficient code in some cases.

Disjoint Unions Done Properly

ML has very clean approach to building and inspecting disjoint unions:

```
datatype result =
  FOUND of integer
| NOTFOUND of string

fun search (..) : result  =
   if ... then
     FOUND 10
   else
     NOTFOUND "problem"

val r = search (...)

case r of
  FOUND x =>
    print ("Found it : " ^ (Int.toString x))
| NOTFOUND s =>
    print ("Couldn't find it : " ^ s)
```

Here `FOUND` and `NOTFOUND` tags are **not** ordinary fields. Case **combines** inspection of tag and extraction of values into one operation.

Object-oriented languages like Java don't support disjoint unions directly, but subclasses provide a (somewhat awkward) way to achieve the same effect. (More later.)

Arrays and Mappings

Basic implementation idea: a **table** laid out in adjacent memory locations permitting **indexed access** to any element.

Mathematically: A finite **mapping** from an **index set** to a **component set**.

Index set is nearly always a set of integers `0..`n, where n is small enough to allow space for the entire array, *or* some other small discrete set isomorphic to them.

Pascal Example:

```
type day = (Sunday, Monday, ..., Saturday);
var workday = array[day] of boolean;
workday[Saturday] := false;
```

More general index sets are seldom supported directly by language because of the lack of a single, uniform, good implementation. Arrays with arbitrary index sets are sometimes called "associative arrays"

Awk Example:

```
workday["Saturday"] = workday["Sunday"] = false;
```

How might this be implemented?

Array Size

Many languages require the index set (and hence size) of arrays to be specified as part of each array type declaration, e.g., in Fortran:

```
ARRAY Q(100)
```

Others permit the size independently for each array value, when the array is first created

- as a local variable, e.g., in Ada:

```
function fred(size:integer);
  var bill: array(0..size) of real;
```

- or on the heap, e.g., in Java:

```
int[] bill = new int[size];
```

Arrays are often large, and hence manipulated by reference.

Major security issue for arrays is **bounds checking** of index values. In general, it's not possible to check all bounds at compile time (though often possible in particular cases).

Runtime checks are always possible, but may be costly. But they are a **good idea**!

Functions and Mappings

Mathematical mappings can also be represented by an algorithmic **formula**.

A **function** gives a "recipe" for computing a **result** value from an **argument** value.

A program function can describe an infinite mapping.

But differs from mathematical function in that:

- it must be specified by an explicit algorithm
- executing the function may have side-effects on variables.

It can be very handy to manipulate functions as first-class values. But most languages put severe limitations on what can be done with functions.

How does one represent a function as a first-class value? In some languages, can just use a **code pointer**. In others, representation must include values of **free variables**, so can get expensive. More on this later.

Sequences

What about data structures of essentially **unbounded** size, such as **sequences** (or **lists**)?

"Take an arbitrary number of values of some type."

Such data structures require special treatment: they are typically represented by small segments of data linked by pointers, and dynamic storage allocation (and deallocation) is required.

The basic operations on a sequence include

- **concatenation** (especially concatenating a single element onto the head or tail of an existing sequence); and
- **extraction** of elements (especially the head).

An important example is the (unbounded) **string**, a sequence of characters.

Best representation depends heavily on what nature and frequency of various operations. Hard to give single, uniformly efficient implementation. So many older languages don't support directly. But so useful that newer languages increasingly do (esp. strings).

Defining Sequences

Unless the programming language supports sequences directly, the programmer must define them using a **recursive** definition.

For example, a list of integers is either

- **empty**, or
- has a **head** which is an integer and **tail** which is itself a list of integers.

ML has particularly clean mechanisms for describing recursive types.

```
datatype intlist =
  EMPTY
| CELL of int * intlist
```

Internally, the non-empty case can be represented by a two-element heap-allocated record, containing an integer and a **pointer** to another list. (Obviously, the tail list itself cannot be embedded in the record, since it's size is unknown.) The empty case is conveniently represented by a `null` pointer. Corresponds directly to C representation:

```
typedef struct intlist *Intlist;
struct intlist {
  int val;
  Intlist next; };
```


Processing Sequences

Note that an iterative or recursive loop is required to process the data in a sequence, e.g.,

```
/* Iterative version */
int inlist(Intlist list, int i) {
  while (list) {
    if (list->val == i)
      return 1;
    else
      list = list->next;
  };
  return 0;
}

/* Recursive Version */
int inlist(Intlist list, int i) {
  if (list) {
    if (list->val == i)
      return 1;
    else
      return inlist(list->rest,i);
  } else
    return 0;
}
```


Recursive Types

Recursion can be used to define and operate on more complex types, in which the type being defined appears more than once in the definition.

ML Example: binary trees with integer labels (only) at the leaves.

```
datatype 'a tree =
  INTERNAL of {left:'a tree,right:'a tree}
| LEAF of {contents:'a}
```

Now we **must** use recursion (not iteration) to process the full tree:

```
fun sum(tree: int tree) =
  case tree of
    INTERNAL{left,right} =>
        sum(left) + sum(right)
  | LEAF{contents} => contents
```


Reference Semantics

Recursive structures naturally grow without fixed bound, so it is common practice to allocate them on the **heap**.

Many modern languages, such as Java and ML, **implicitly** allocate records (and disjoint unions) on the heap, and represent record **values** by **references** (pointers) into the heap.

As a natural result, both languages use **shallow copy** semantics for assignment and argument passing. Java Example:

```
class emp {
  String name;
  int age;
}
emp e1;
e1.age = 91;
emp e2 = e1;
e1.age = 18;
System.out.print(e2.age);
```

prints 18

Neither language allows user programs to manipulate the internal pointers directly. And neither supports explicit **deallocation** of records (or objects) either; both provide automatic **garbage collection** of unreachable heap values, thus avoiding both **dangling pointer** and **memory leak** bugs.

Explicit Pointers

Many previous languages had **pointer types** to enable programmers to construct recursive data structures, e.g., in C:

```
typedef struct intlist *Intlist;
struct intlist {
  int head;
  Intlist tail;
}
Intlist mylist =
    (Intlist) malloc(sizeof(struct intlist));
...free(mylist)...
```

Note that programmers must make explicit `malloc` (C++: `new`) and `free` calls to manage heap values, and must explicitly manipulate pointers.

Lots of opportunity for dangling pointer bugs and memory leaks!

In most such languages, pointers are restricted to addresses returned by allocation operations, but C/C++ allows the address of **anything** to be taken and later dereferenced, and supports **pointer arithmetic**. While this feature can support very sufficient code, it also destroys the safety of the type system.

Type Equivalence

When do two identifiers have the "same" type, or "compatible" types?

I.e., if `a` has type t_1, `b` has type t_2 and `f` has type $t_2 \rightarrow t_3$, how must t_1 and t_2 be related for these to make sense?

```
a := b
f (a)
```

To maintain whatever **security** type-checking of primitive types gives us, we must insist at a minimum that t_1 and t_2 are **structurally equivalent**.

Structural equivalence is defined inductively:

• Primitive types are equivalent iff they are the same type.

• Cartesian product types are equivalent if their corresponding component types are equivalent. (Record field names are typically ignored.)

• Disjoint union types are equivalent if their corresponding component types are equivalent.

• Mapping types (arrays and functions) are the same if their domain and range types are the same. (Sometimes the cardinality of the index type of an array is ignored.)

Equivalence (continued)

Another way to say this: two types are equal if they have the same set of values.

Recursive types are a problem. Are these two types structurally equivalent?

```
type t1 = record a:int, b: POINTER TO t1 end;
type t2 = record a:int, b: POINTER TO t2 end;
```

Intuitively yes, but it's (a little) tricky for a type-checking algorithm to determine this!

Type Names

Question becomes more interesting because of type **names**.

We name types for two possible reasons:

• As a convenient shorthand to avoid giving the full type each time. E.g.,

```
fun f(x:int * bool * real) :
                   int * bool * real = ...
type t = int * bool * real
fun f(x:t) : t = ...
```

• As a way of improving program correctness by subdividing values into types according to their meaning **within the program**.

```
type polar = record r:real, a:real end;
type rect = record x:real, y:real end;
function polar_add(x:polar,y:polar) : polar ...
function rect_add(x:rect,y:rect) : rect ...
var a:polar; c:rect;
a := (150.0,30.0) (* ok *)
polar_add(a,a)  (* ok *)
c := a  (* type error *)
rect_add(a,c) (* type error *)
```

For this to be useful, some structurally equivalent types must be treated as **inequivalent**.

Name Equivalence

Basic idea: Two types are equivalent iff they have the same **name**.

Supports `polar/rect` distinction.

But pure name equivalence is very restrictive, e.g.:

```
type ftemp = real
type ctemp = real
var x:ftemp, y:ftemp, z: ctemp;
x := y; (* ok *)
x := 10.0; (* probably ok *)
x := z; (* type error *)
x := 1.8 * z + 32.0; (* probably type error *)
```

Different types now seem **too** distinct; can't even convert from one form of real to another.

Also: what about unnamed type expressions?

```
type t = int * int
procedure f(x: int * int) = ...
procedure g(x: t) = ...
var a:t = (3,4)
g(a); (* ok *)
f(a); (* ok or not ?? *)
```

Most languages use **mixed** solutions.

C Type Equivalence

C uses structural equivalence for array and function types, but name equivalence for `struct`, `union`, and `enum` types. For example:

```
char a[100];
void f(char b[]);
f(a); (* ok *)
```

```
struct polar{float x; float y;};
struct rect{float x; float y;};
struct polar a;
struct rect b;
a = b; (* type error *)
```

A type defined by a `typedef` declaration is actually just an abbreviation for an existing type.

Note this policy makes it easy to check equivalence of recursive types, which can only be built using `struct`s.

```
struct fred {int x; struct fred *y;} a;
struct bill {int x; struct fred *y;} b;
a = b; (* type error *)
```


Pascal Type Equivalence

The original Pascal definition is vague. The standard implementation method is a variant of name equivalence, with the following wrinkles:

• Each `type` declaration defines a new type, unless the right hand side is a simple type name. E.g., `T` and `U` are different types, but `T` and `V` are the same type.

```
type T = RECORD a:INTEGER; b: REAL END;
type U = RECORD a:INTEGER; b: REAL END;
type V = T  (* just an abbreviation for T *)
```

• Each anonymous type expression defines a new type. E.g., the types of `x,y,z` are all different, but `z` and `w` are the same.

```
type T = RECORD a:INTEGER; b: REAL END;
VAR x: T;
VAR y: RECORD a:INTEGER; b: REAL END;
var z,w: RECORD a:INTEGER; b: REAL END;
```

Another way to describe the type system is that each application of a **type constructor** (i.e., `RECORD, ARRAY, POINTER, etc.`) creates a new type. In this case, must **not** think of built-in types like `INTEGER` and `REAL` as constructors.

ML Type Equivalence

ML uses structural equivalence, except that each `datatype` declaration creates a new type unlike all others.

```
datatype polar = POLAR of real * real
datatype rect = RECT of real * real
val a = POLAR(1.0,2.0)
and b = RECT(1.0,2.0)
if (a = b) ...  (* type error *)
```

Note that the mandatory use of constructors makes it possible to uniquely identify the types of literals.

Note that a datatype need not declare a record:

```
datatype fahrenheit = F of real
datatype celsius = C of real
val a = F 150.0
val b = C 150.0
if (a = b) ... (* type error *)
fun convert(F x) = C(1.8 * x + 32.0) (* ok *)
```

For type abbreviation, ML offers the `type` declaration, which simply gives a new name for an existing type.

```
type centigrade = celsius
fun g(x:centigrade) = if x = b ... (* ok *)
```