Values, Types and Expressions *overview*

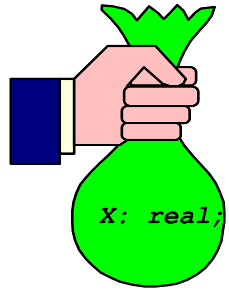
- Typology of values and types:
  - primitive, composite and recursive
  - set theoretical representation
  - polynomial representation of types
    (to be covered in the tutorials)
- Using values: first class values and second class values
- Typology of expressions

1

TYPOLOGY OF VALUES AND TYPES

2

Values



- A *value* is any entity that exists during a computation
- Alternatively, a value is anything that may be manipulated by a program
- Operational definition (in Pascal): anything that may be passed as an argument to a subroutine

- Two ways to classify values in a language:
  - By their *structure* (the way they are defined)
  - By their *functionality* (the ways they are manipulated)

Value Structure

- **Primitive value:** *is not* composed of other values
  - truth values, characters, integers, reals, pointers
- **Composite value:** *is* composed of other values
  - records, arrays, sets, files
- The ways to create composite values in a language are usually independent of its implementation
- The set of legal values in a language's implementation:
  - a closure of the primitive values in this *implementation* under the mechanisms the language *specification* allows for creating composite values
  - a potentially infinite set, but in fact finite but large

Value Manipulation

- Operations on values
  - Passing them to procedures as arguments
  - Returning them through an argument of a procedure
  - Returning them as the result of a function
  - Assigning them into a variable
  - Using them to create a composite value
  - Creating/computing them by evaluating an expression
- A value for which *all* these operations are allowed is called a *first-class value*
- We are used to integer or character values, but function values are also possible!

Values in Pascal

- **First-class values**
  - **Only the primitive values:** truth values, characters, enumerands, integers, reals, pointers.
- **Lower-class values** – can be passed as arguments, but cannot be stored, or returned, or used as components in other values
  - **References to variables in Pascal**
  - **Procedure and function abstractions in Pascal**
- **Composite values –** records, arrays, sets and files
  - *Not* first class: cannot be returned!

6

Values in ML

- All the values in ML are first-class values:
  - **Primitive values:** truth values, integers, reals, strings.
  - **Composite values:** records, tuples (records w/o field names), constructions (tagged values), lists, arrays.
  - **Function abstractions**
  - **References to variables**
- What we can do in ML but *not* in Pascal:
  - write a *function* that gets a function `f:int->int` and returns the composition of `f` with itself
  - create a *record* composed of two functions

7

Expressions

- **Expression:** A part of a program that is translated into a value *(evaluated)* during computation
- Some examples:
  - `3.1416` `'%'` `'Hello, world'` (Pascal)
  - `2*a[i]+7` `sqr(4)` `q^.head` (Pascal)
  - **if** leap (thisyear) **then** 29 **else** 28 (ML)
- We'll see more later….

8

The need for types

- In machine language – all values are just bit patterns => they are *untyped*
- In assembler languages – there is already a difference between *addresses* and *data*
  - The sum of two addresses is ill-defined => tagged architectures that add type information to values in runtime
  - Problem: usually deals only with predefined types, but not with user-defined types
- Observation: a type is also a property of
  - memory cells – where typed values are stored
  - expressions – from which typed values are evaluated
- High-level languages: attach types also to expressions (as will be seen later)
- Type checking (of expressions, arguments, etc.) is crucial to discovering more errors automatically

9

Historical Background

- Initially, an assumption of set theory:
  - For every imaginable property, there exists a set of objects that satisfy this property
- **Russell's Paradox:** The following set R leads to a contradiction: $R = \{x \mid x \notin x\}$
  - If we assume that R is a member of R we must conclude that R is *not* a member of R, and vice versa!
  - *In a town, a barber shaves precisely those men who do not shave themselves; who shaves the barber?*
- **Resolution:** Use tagging. The following is allowed only if *x* is restricted to range over a set *P*, where the set of possible *P*s is predefined

$$S_P = \{x \in P \mid \ldots\}$$

Each set carries a *tag*. These tags evolved into *types* in programming languages

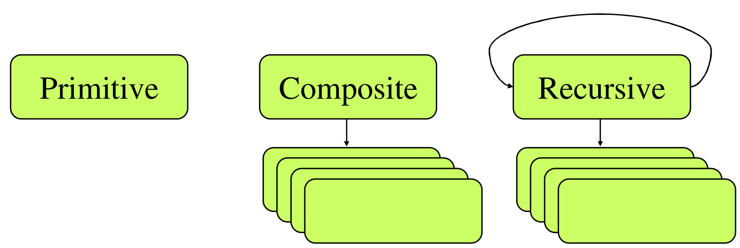
10

Types as Sets of Values

- Every type corresponds to a ***set of values***
  - A *value v* is of type *T* => *v* is in a set that *T* defines
  - An *expression E* is of type *T* =>
    the result of evaluating *E* is a value of type *T*
- However, not every set of values corresponds to a type!
- A ***type*** is a set of values with operations that can be applied uniformly for every value in the set
- For example:
  - `{Sunday, 13, "November"}` does not correspond to any type in any language
  - { **`false,true`** } corresponds to a type in many languages
- **Conclusion:** the definition of the set of types is a **pragmatic** decision, based on the objectives of the language, and not on the formal properties of its values

Types

- **Primitive type**: a type that has only primitive values
- **Composite type:** a type that has (many) composite values
- **Recursive type:** a type that has (many) values that are composed both of values of the *same type* as well as values of other (base) types



12

Primitive Types

- A **type system** is a collection of sets of values and the relations among these sets
- The primitive types are the building blocks of this collection: their values *cannot be broken* further into smaller values
- Primitive types are subdivided into**:**
  - **Rudimentary:** These types are built-in into the programming language (e.g. *truth value, integer, real, character*)
  - **Non-Rudimentary:** User defined primitive types
- Examples of non-rudimentary primitive types:
  - **Enumerated types** (Ada, Pascal, C, …)

    ```
    TYPE Month = (January, February, March, April, May,
    Jun, July, August, September, October, November,
    December)
    ```
  - **Subranges** (Pascal, …): a range of consecutive values.

    ```
    TYPE DayOfMonth = 1..31
    ```

13

Primitive Types (cont'd)

- Choice of rudimentary primitive types tells much about the intended purpose of the programming language:
  - **Fortran – numerical computation.** Choice of precision of real and complex numbers.
  - **Cobol – data processing.** Fixed length strings, fixed point numbers.
  - **Snobol – string processing.** Variable length strings.
- **Point of confusion:** Different programming languages use different names for the same primitive type.
  - Pascal: **Boolean, Integer, Real**
  - ML: **bool, int, real**
- **Point of difficulty**: Different implementations of the language may use different sets of values for the same type.
- **Distinction between name and value:** The enumerand values and their identifiers are distinct.
  - The identifier `December` can be redefined, but the value can still be accessed as `succ(November)`
  - Even worse in FORTRAN…

14

Primitive Types (cont'd)

- In C and C++, the values of `bool` (**false** and **true**) just denote 0 and other integers (respectively). In Pascal and other languages they are completely separate values.
- The same is true for values of `char` that are just different names for the small non-negative integers ('a' for 97, 'b' for 98, etc.) in C and C++
- The same is true for enumerated types in C and C++. Versus Pascal and Ada
- Some languages have several types for integers and reals
  - C and C++ have `float` and `double` for reals
  - Java has `byte short, int ,long` for differing ranges of integers

15

Cardinality of a Type

- The number of values in the set corresponding to that type.
- For type T, denoted #T
- #Boolean = 2
- #char = 256 (ISO Latin set) or 65,536 (Unicode)
- #Months = 12
- #integer = ???
  - Usually implementation dependent, or defined for special types of integers (#byte = 256)
  - In Pascal: 2∗(maxint+1)
- #real – even more complicated (not unique)

16

Ordering Primitive Types

- **Ordered type:** a type for which a total order relation is defined.
  - Useful in iterators and conditional expressions
- **Unordered types:** complex (FORTRAN, Mathematica, Matlab)
- **Discrete type:** an ordered type whose set of values has a one-to-one order preserving mapping with a (range of) integers. Examples: *Boolean, char, integer.*
- **Ordered indiscrete types:** *string, real.*
- **Pascal**: Only discrete types can be used in:
  - indices for loops and arrays (reasonable policy)
  - **Case** statements (arbitrary policy, as in machine language)
- **C++:** Only discrete types can be used as template arguments.

17

Semi-Primitive Types

- **Semi-primitive types 1:**
  - **Atomic:** values cannot be broken down into values of smaller types
  - **Non-rudimentary:** defined by the programmer

  Example: enumerated types

- **Semi-primitive types 2** (pseudo-primitive types):
  - **Non-atomic:** values can be broken down into values of smaller types
  - **Rudimentary:** are built-in into the programming language

  Example: strings in several programming languages
  - Values can be broken into smaller strings
  - Type string is built into the language

18

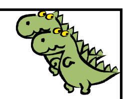
Strings

- **String**: a sequence of characters. Useful data type. Supported in one way or another by all modern programming languages.
- No consensus. Issues:
  - Primitive (as in Icon) or composite (as in Pascal) type?
  - Fixed length (as in C) or variable length (as in Cobol)?
  - What string operations are supported?
  - String literals? Delimiters or quotes?

A literal of type T is (roughly) a constant of type T which can be used in the program text

- **ML: a** primitive type of any length. Operations: equality test, concatenation, decomposition are built-in.
- **Pascal and Ada**: an array of characters. All array operations are available. Disadvantage: all string operations must be defined in term of these fixed length strings.
- **Algol-68**: same as Pascal and Ada, but with flexible arrays (size changed during runtime).
- **C**: semi-flexible arrays. String literals.
- **Prolog and Miranda**: a list of characters. Only the first character can be selected. All other string operations, e.g., ordering, sub-strings, etc. must be explicitly defined.

Composite Types



- **Type operators:** create a new composite type out of primitive and composite types:
  - tuples, records, variants, unions, arrays, sets, strings, lists, trees, serial files, direct files, relations, etc.
- All can be understood in terms of set theory operators:
  - Cartesian products: **tuples and records.**
  - Disjoint unions: **variants and unions.**
  - Mappings: **arrays and functions.**
  - Power sets: **sets.**
  - Enclosure under operator: **recursive types, dynamic data structures.**
- We will introduce *two* mathematical notations that provide a common foundation for composite types in many programming languages:
  - **Set theory:** primitive types are (arbitrary) sets of values, composite types are other sets defined in terms of these sets
  - **Symbolic functions:** primitive types are formal variables, composite types are polynomials over these variables

Cartesian Products: tuples and records

- Given two types *S* and *T*, we denote their Cartesian product by *S* x *T*.
  - $S \times T = \{ (x, y) \mid x \in S;\ y \in T \}$
  - $\#(S \times T) = \#S \times \#T$

  This can be generalized to more than two sets: $S_1 \times S_2 \times \ldots \times S_n$
- The tuple of ML, the records of Cobol, Pascal, Ada, Icon and ML, and the so-called structures Algol-68 and C can *all* be understood in terms of Cartesian products.
- ML tuples:

  ```
  type person = string * string * int * real
  ```

  A decomposition of a value `someone` of type `person`:

  ```
  val (surname, forename, age, height) = someone
  if age >= 18 then ... else ...
  ```
- ML records (*labeled* Cartesian product):

  ```
  type person = { surname: string,
                  forename: string,
                  age: int,
                  height: real }
  ```

Cartesian Products (cont'd)

- **Homogenous tuples:** a special case of the Cartesian product is one where all the tuple components are chosen from the same set:
  - $S^n = S \times S \times \ldots \times S$
  - $\#(S^n) = (\#S)^n$
- We observe that S^0 has exactly one value: the *0-tuple.*
  - Unit = { () }
  - Unit is not the empty set: it contains a single value which happens to be a tuple with no components.
- Unit corresponds to the **`unit`** type of ML, and **`void`** type of C.

The "Unit" Type

- Primitive types have cardinalities, e.g.,
  - #Boolean = 2
  - #integer = 2**n
  - #real = …
- A type with cardinality 1?
  - Values of such a type require no storage!
  - C's **`void`**, Pascal's **`nil`**, etc.
- Technically, the Unit type does not have to be a primitive type
  - It can be thought of as a record with no fields in it: the *neutral element* of the record "type operator"

23

Type `void` in C and Other Subtle Points

- **`void`** `f1(`**`int`**`)`
  - Is <u>not</u> a function that does not return anything
  - It is rather a function that can return in only one way, i.e. returns type unit
- **`int`** `f2(`**`void`**`)`
  - Takes no arguments
  - Can be thought of as taking an argument of type unit
- **`int`** `f3(...);`
  - function is declared to have a variable number of arguments
- **`extern`** `f4():`
  - return type is implicitly int
  - In C, no declaration on arguments
  - In C++, function takes no argument (an argument of type unit)

Note: In C, unlike Java, the return type of main() is not void. It is rather int, by which the program can report to the operating system its terminated successfully (value 0), or that it failed (any other value).

24

Unit Type in C

`void` is not an ordinary type in C. Although there is a void return type and a **`void`** argument, there is no way to make variables of type **`void`**. However, we can emulate unit type in C using one of the following tricks:

```
typedef enum Unit {
    unit
} Unit;
```

The value of this type is written {} in C, however, this value can only be used in initialization statements

Or...

```
typedef struct {
} Unit;
```

Ditto!

Or...

```
typedef int Unit[0];
```

25

Disjoint Unions:
Choice Type Operators

- Each value is chosen from either set *S* or set *T*
  - $S + T = \{ \text{left } x \mid x \in S \} \cup \{ \text{right } y \mid y \in T \}$
  - $\#(S + T) = \#S + \#T$
- Examples:
  - C's union
  - Pascal's variant record
  - ML's constructions
  - Java and Eiffel: missing! Typical to OO languages - the need for choice type operators is diminished with OO features.
- Useful for representing *pointers*:
  - Pointer to type X:
    - either points to a value of type X, or is nil/void/0 (depending on your terminology)
    - can be thought of as a choice between Unit type and the type (X)

26

Variant Records in Pascal

- Syntax of definition:

```
record case I: T of
    L1: (I1: T1);
    ...
    Ln: (In:Tn)
end
```

- Example:

```
type      Accuracy = (exact, approx);
   Number  = record case tag: Accuracy of
         exact: (ival: Integer);
         approx:(rval: Real)
   end
```

The values of type `Number` **are:**

{ ..., exact -2, exact -1, exact 0, exact 1, exact 2,...} ,
{ ... approx -1.0, ..., approx 0.0, ..., approx 1.0, ...}

Using Pascal's Variant Record

- In Pascal, Turbo Pascal and in C, a disjoint union type may be accessed in the same way as ordinary Cartesian product elements. *This is unsafe!*
- Consider the following code:

```
VAR n: Number;
...
n.tag  := exact; (* n.ival is still undefined *)
n.ival := 7;
...
n.tag := approx; (*  n's value is changed in one step
                     from exact 7 to approx undefined *)
```

- Safe decomposition of a variant record

```
function round_num(n: Number): Integer;
  case n.tag of
   exact:  round_num := n.ival;
   approx: round_num := round(n.rval)
  end;
end;
```

Variant Records in ML

- ML construction definitions:

```
datatype number = exact of int | approx of real;
```

- Construction literals:

```
exact(i + 1)
approx(r/3.0)
```

- Decomposing a construction:

```
case n of exact i => i | approx r => round(r);
```

- Observe that such a mechanism may enhance a dynamic typing system, in the same way that C's ternary `?:` operator does.

Tagging in Choice Types

- **Tag**: A mechanism for storing the selection made in a choice type.
- Different programming languages provide different levels of support for tagging:
  - **ML**: Tagging is built into the language.
    - **The tag is implicit**: You cannot access a "Tag Field" directly.
    - **Correct tagging is enforced**: There is no way to store a value into one choice selection and read it from another.
  - **Pascal**: The compiler forces a definition of a tag field. When a tag field is updated, the record's corresponding fields are created accordingly.
    - But: the compiler does not enforce correct <u>access</u> to record according to tag.
  - **Turbo-Pascal**: A Pascal extension which allows the programmer to get away without defining a tag field.
  - **C**: Responsibility lies entirely with programmer, both for defining (or not) a tag field and for using it correctly.
  - **Pointers in all languages**: Tagging is implicit in tests for null pointer. However, not all languages complain if you try to read from or write to a null pointer.
- We presume that tagging exists in all choice type operators, be it by language design or by programmer responsibility

30

The Need for Tagging

- Recall that tagging is needed for safe decomposition.
- But the need is also observed in the set-theoretical representation.
- Suppose that the representation of `height` were simply the union set $I \cup I$
- Then we would get: $I \cup I = I$
- With tagging we have e.g.:

  $I_{cm}=\{cm\}\times I,\quad I_{in}=\{in\}\times I,\quad cm \neq in$
- And we can simply represent `height` by a disjoint union of I_{cm} and I_{in}.

```
typedef union
height {
   int cm;
   int in;
};
```

31

More on tags

- The basic operations on disjoint union S+T:
  - Construction : build a disjoint union value by taking a value from S or T and adding the appropriate tag
  - Tag test: determine if the variant came from S or T (*right* v is from T)
  - Projection: get the value by removing the tag (*right* v will return v)
- A less desirable (albeit equivalent) alternative would be a structure of a Boolean and an integer.

32

Choice and Enumerated Types

- An enumerated type can be simulated as a choice between `unit`s:

```
typedef enum Suit {
     diamond, heart, spade, clover,
} Suit;
```

A choice between four empty structures with different names

```
typedef union Suit {
     struct {} diamond;
     struct {} heart;
     struct {} spade;
     struct {} clover;
} Suit;
```

33

The “none” Type

- **`none`**: a type with cardinality 0
  - This type has no legal values
  - It would be meaningless to define variables of this type, since no value could ever be stored into these variables.
- **Example**: The function `exit()` in C, which never returns.
  - The return type of this function should not be **`void`** but rather **`none`**.

34

Emulating `none` in C

```
typedef enum {
} none;
```

Enumerated type, with no possible values

Or...

```
typedef union {
} none;
```

A union which has no fields. There is no legal way to assign a value to such a union

Check how your C compiler deals with these types...

35

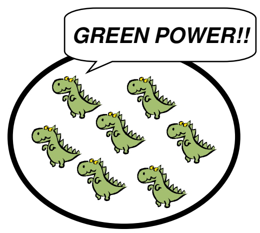
Mappings: arrays and functions

- Arrays as mappings:
  - the type "**array** [S] **of** T" (in Pascal S must be discrete) corresponds to a mapping S->T.
  - the type "**array** [S_1, S_2, ... , S_n] **of** T" corresponds to a mapping (S_1 x S_2 x ... x S_n)->T,
  - alternatively, using **Currying**, to (S_1->(S_2->(…-> S_n)))->T
- Functions as mappings:
  - "**function** even (n: Integer) : Boolean" corresponds to a function Integer->Boolean.
  - **Note:** functions in most programming languages are algorithmic *implementations* of mathematical functions/mappings. They have however other properties that functions do not have:
    - o **efficiency**
    - o **side effects**
- $\#(S\text{->}T) = \#T^{\#S}$

36

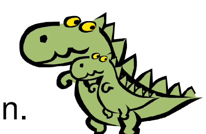
Power Sets:
sets

- $\#(\wp(T)) = 2^{\#T}$
- corresponds to "**set of** T" in Pascal
- Ideally: isomorphic to T->Boolean
- In Pascal: for efficiency T must be discrete
  => in effect similar to "**array [**T] **of** Boolean"



37

Recursive Types



- A recursive type is defined in terms of itself.
- Modern Languages like ML allow direct definition.
- **Example 1**- IntList = nil Unit + cons(Integer x IntList)
- lists in ML:

```
datatype intlist = nil | cons of int * intlist
```

  This is an abbreviation for `nil of unit | cons of ...`
- Values of intlist:

```
nil
cons(11,nil)
cons(2,cons(3,cons(5,cons(7,cons(11,nil)))))
```

- These can be obtained by substituting values so far in the right-hand side of the definition, to get new values.
- Doing this "forever" gives all finite lists, which are a "solution" of the defining equation (actually the "**least solution**")
- We could also consider infinite lists, but usually don't…

More on recursive lists

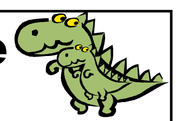
- We will see that such recursive definitions are useful and can be efficient.
- Each list is finite, but there is no global limit on the number of elements (so there are infinitely many lists defined here)
- Further, ML has a pre-defined type constructor called **`list`**:

```
int list
bool list
int list list
```

- Operations
  - **Basic Predefined:** test for emptiness, select head, select tail.
  - Concatenation and length (could be defined in terms of basic ops)
- In Ada/C/Pascal, recursive types must be defined with pointers.

39

Another Recursive Type



- **Example 2**: trees

  datatype inttree = leaf **of** int | branch **of** inttree * inttree
- Possible values:
  - leaf 11
  - branch (leaf 11, leaf 5)
  - branch (branch (leaf 11, leaf 5), leaf 11)
- Note: most values of this recursive type are composed of values of the same type.
- Set theoretical representation:

  Integer_Tree = Integer $\cup$ (Integer_Tree x Integer_Tree)
- This equation has many solutions but it defines a **least solution**, which can be approximated by substituting the empty set on the left side for **Integer_Tree** and iterating the process.

40