# Programming Languages/Values and Types 

Onur Tolga Șehitoğlu<br>Computer Engineering, METU

Spring'2008

## Outline

1 Value and Type
2 Primitive vs Composite Types
3 Cartesian Product
4 Disjoint Union
5 Mappings

- Arrays
- Functions

6 Powerset
7 Recursive Types

- Lists
- General Recursive Types
- Strings

8 Type Systems

- Static Type Checking
- Dynamic Type Checking

■ Type Equality
9 Type Completeness
10 Expressions
■ Literals/Variable and Constant Acces

- Aggregates
- Variable References
- Function Calls
- Conditional Expressions
- Iterative Expressions

11 Summary

## What are Value and Type?

- Value anything that exist, that can be computed, stored, take part in data structure.
Constants, variable content, parameters, function return values, operator results...
- Type set of values of same kind. C types:

■ int, char, long,...

- float, double
- pointers
- structures: struct, union
- arrays
- Haskell types

■ Bool, Int, Float, ...
■ Char, String
■ tuples,(N-tuples), records

- lists
- functions

■ Each type represents a set of values. Is that enough? What about the following set? Is it a type? \{"ahmet", 1 , 4 , 23.453, 2.32, 'b'\}
■ Values should exhibit a similar behavior. The same group of operations should be defined on them.

## Primitive vs Composite Types

■ Primitive Types: Values that cannot be decomposed into other sub values.
C: int, float, double, char, long, short, pointers Haskell: Bool, Int, Float, function values

- cardinality of a type: The number of distinct values that a datatype has. Denoted as: "\#Type". \#Bool $=2 \quad$ \#char $=256 \quad \#$ short $=2^{16}$ $\#$ int $=2^{32} \quad \#$ double $=2^{32}, \ldots$
- What does cardinality mean? How many bits required to store the datatype?


## User Defined Primitive Types

- enumerated types
enum days \{mon, tue, wed, thu, fri, sat, sun\}; enum months \{jan, feb, mar, apr, ... \};
■ ranges (Pascal and Ada)
type Day = 1..31;
var g:Day;
- Discrete Ordinal Primitive Types Datatypes values have one to one mapping to a range of integers.
C: Every ordinal type is an alias for integers.
Pascal, Ada: distinct types
- DOPT's are important as they
i. can be array indices, switch/case labels
ii. can be used as for loop variable (some languages like pascal)


## Composite Datatypes

User defined types with composition of one or more other datatypes. Depending on composition type:

- Cartesian Product (struct, tuples, records)
- Disjoint union (union (C), variant record (pascal), Data (haskell))
■ Mapping (arrays, functions)
- Powerset (set datatype (Pascal))
- Recursive compositions (lists, trees, complex data structures)


## Cartesian Product

- $S \times T=\{(x, y) \mid x \in S, y \in T\}$
- Example:
$S=\{a, b, c\} \quad T=\{1,2\}$
$S \times T=\{(a, 1),(a, 2),(b, 1),(b, 2),(c, 1),(c, 2)\}$
$\left.\begin{array}{l}\bullet \mathrm{a} \\ \bullet \mathrm{b} \\ \bullet \\ \bullet\end{array}+\begin{array}{l}\bullet 1 \\ \bullet 2 \\ \bullet(\mathrm{a}, 1) \\ \bullet(\mathrm{b}, 1) \\ \bullet(\mathrm{c}, 1) \\ \bullet(\mathrm{b}, 2) \\ \bullet(\mathrm{c}, 2)\end{array}\right)$
- $\#(S \times T)=\# S \cdot \# T$
- C struct, Pascal record, functional languages tuple
- in C: string $\times$ int

```
struct Person {
    char name[20];
    int no;
} x = {"Osman}\sqcupHamdi", 23141}
```

- in Haskell: string $\times$ int

```
type People=(String,Int)
(x::People) = ("Osman}\sqcupHamdi", 23141)
```

- Multiple Cartesian products: C: string $\times$ int $\times\{$ MALE,FEMALE $\}$
struct Person \{ char name[20]; int no; enum Sex \{MALE, FEMALE\} sex;
$\} x=\left\{" O s_{m a n}^{\sqcup} H^{H a m i} ", 23141\right.$,FEMALE $\} ;$
Haskell: string $\times$ int $\times$ float $\times$ String
$x=\left(" O s m a n_{\sqcup} H a m d i ", 23141,3.98\right.$, "Yazar")


## Homogeneous Cartesian Products

- $S^{n}=\overbrace{S \times S \times S \times \ldots \times S}^{n}$ double ${ }^{4}$ :
struct quad \{ double $x, y, z, q ;\} ;$
- $S^{0}=\{()\}$ is 0-tuple.
- not empty set. A set with a single value.
- terminating value (nil) for functional language lists.
- C void. Means no value. Error on evaluation.


## Disjoint Union

$■ S+T=\{$ left $x \mid x \in S\} \cup\{$ right $x \mid x \in T\}$

- Example:
$S=\{1,2,3\} \quad T=\{3,4\}$
$S+T=\{$ left 1 , left 2 , left 3 , right 3 , right 4$\}$


■ $\#(S+T)=\# S+\# T$

- C union's are disjoint union?

■ C: int + double:
union number \{ double real; int integer; \} $x$;

- C union's are not safe! Same storage is shared. Valid field is unknown:

```
x.real=3.14; printf("%d\n",x.integer);
```

■ Haskel: Float $+\operatorname{Int}+(\operatorname{lnt} \times \operatorname{lnt})$ :

```
data Number = Rea|Val Float | IntVa| Int | Rational (Int,Int)
x = Rational (3,4)
y = RealVal 3.14
z = IntVal 12 {-- You cannot access different values --}
```


## Mappings

- The set of all possible mappings

■ $S \mapsto T=\{V \mid \forall(x \in S) \exists(y \in T),(x \mapsto y) \in V\}$
■ Example: $S=\{a, b\} \quad T=\{1,2,3\}$


Each color is a mapping value There are many others

$$
\begin{aligned}
& S \mapsto T=\{\{a \mapsto 1, b \mapsto 1\},\{a \mapsto 1, b \mapsto 2\},\{a \mapsto 1, b \mapsto 3\}, \\
& \{a \mapsto 2, b \mapsto 1\},\{a \mapsto 2, b \mapsto 2\},\{a \mapsto 2, b \mapsto 3\}, \\
& \{a \mapsto 3, b \mapsto 1\},\{a \mapsto 3, b \mapsto 2\},\{a \mapsto 3, b \mapsto 3\}\} \\
& \#(S \mapsto T)=\# T \# S
\end{aligned}
$$

## Arrays

- double $\mathrm{a}[3]=\{1.2,2.4,-2.1\}$;
$a \in(\{0,1,2\} \mapsto$ double $)$
$\mathrm{a}=(0 \mapsto 1.2,1 \mapsto 2.4,2 \mapsto-2.1)$
- Arrays define a mapping from an integer range (or DOPT) to any other type
■ C: $T \mathrm{x}[N] \Rightarrow \mathrm{x} \in(\{0,1, \ldots, N-1\} \mapsto T)$
■ Other array index types (Pascal):

```
type
    Day = (Mon,Tue,Wed,Thu, Fri,Sat,Sun);
    Month = (Jan,Feb,Mar,Apr,May,Jun, Jul,Aug, Sep,Oct,Nov, Dec);
var
    x : array Day of real;
    y : array Month of integer;
    x[Tue] := 2.4;
    y[Feb] := 28;
```


## Functions

- C function:

```
int f(int a) {
    if (a%2 == 0) return 0;
    else return 1;
}
```

■ $\mathrm{f}: \operatorname{int} \mapsto\{0,1\}$
regardless of the function body: $\mathrm{f}:$ int $\mapsto$ int
■ Haskell:
f a $=$ if $\bmod$ a $2==0$ then 0 else 1
■ in C, $f$ expression is a pointer type int (*) (int) in Haskell it is a mapping: int $\mapsto$ int

## Array and Function Difference

## Functions

Arrays:

- Values stored in memory
- Restricted: only integer domain
- double $\mapsto$ double ?
- Defined by algorithms

■ Efficiency, resource usage

- All types of mappings possible
■ Side effect, output, error, termination problem.
- Cartesian mappings: double a[3] [4]; double $f$ (int $m$, int $n$ );
■ int $\times$ int $\mapsto$ double and int $\mapsto($ int $\mapsto$ double)


## Cartesian Mapping vs Nested mapping

■ Pascal arrays

```
var
        x : array [1..3,1..4] of double;
    y : array [1..3] of array [1..4] of double;
x[1,3] := x[2,3]+1; y[1,3] := y[2,3]+1;
```

Row operations:

$$
\begin{array}{lll}
\mathrm{y}[1] & :=\mathrm{y}[2] & ; \\
\mathrm{x}[1] & :=\mathrm{x}[2] & ;
\end{array}
$$



■ Haskell functions:
$f(x, y)=x+y$
$g \times y=x+y$
f $(3+2)$
g 32

- g 3
f 3
- Reuse the old definition to define a new function:
increment = g 1
increment 1
2


## Powerset

- $\mathcal{P}(S)=\{T \mid T \subseteq S\}$
- The set of all subsets
- $S=\begin{aligned} & \bullet 1 \\ & \bullet 2 \\ & \bullet 3\end{aligned} \quad \mathcal{P}(S)=\begin{array}{ll}\bullet \emptyset\{1\} & \bullet\{2\} \\ \bullet\{1,2\} & \bullet\{1,3\} \\ \bullet\{2,3\} & \bullet\{1,2,3\}\end{array}$
- $\# \mathcal{P}(S)=2^{\# S}$
- Set datatype is restricted and special datatype. Only exists in Pascal and special set languages like SetL
- set operations (Pascal)

```
type
    color = (red,green,blue,white,black);
    colorset = set of color;
```

var
a,b : colorset;
a $:=$ [red, blue];
b := a*b; (* intersection *)
b : = a+[green, red]; (* union *)
b := a-[blue]; (* difference *)
if (green in b) then ... (* element test *)
if (a = []) then ... (* set equality *)

- in C++ supported by library.


## Recursive Types

■ $S=\ldots S \ldots$
■ Types including themselves in composition.

## Lists

■ $S=\operatorname{Int} \times S+\{n u l /\}$

$$
\begin{aligned}
S= & \{\text { right empty }\} \cup\{\text { left }(x, \text { empty }) \mid x \in \operatorname{Int}\} \cup \\
& \{\text { left }(x, \text { left }(y, \text { empty })) \mid x, y \in \operatorname{Int}\} \cup \\
& \{\text { left }(x, \text { left }(y, \text { left }(z, \text { empty }))) \mid x, y, z \in \operatorname{Int}\} \cup \ldots
\end{aligned}
$$

- $S=$
$\{$ right empty, left(1, empty), left( 2, empty), left( 3, empty), $\ldots$, left( 1, left ( 1, empty ) ), left( 1, left ( 2, empty) ), left( 1, left( 3, empty $)$, left( $1, \operatorname{left}(1$, left( 1, empty $))$ ), left( $1, \operatorname{left}(1, \operatorname{left}(2$, empty $))), \ldots\}$
- C lists: pointer based. Not actual recursion.

```
struct List {
    int x;
    List *next;
} a;
```

- Haskell lists.

```
data List = Left (Int,List) | Empty
x = Left (1, Left(2, Left(3,Empty)))
y = Empty
--}
```

■ Polymorphic lists: a single definition defines lists of many types.

- List $\alpha=\alpha \times($ List $\alpha)+\{$ empty $\}$

```
data List alpha = Left (alpha,List alpha) | Empty
x = Left (1, Left(2, Left(3,Empty)))
y = Left ("ali",Left("ahmet",Empty))
z = Left(23.1, Left(32.2,Left(1.0,Empty))){-- [23.1,32.2,1.0]
```

■ Left (1, Left("ali", Left(15.23, Empty) $\in \operatorname{List} \alpha$ ? No. Most languages only permits homogeneous lists.

## Haskell Lists

■ binary operator ":" for list construction:

```
    data [alpha] = (alpha : [alpha]) | []
■ x = (1: (2: (3: [])))
```

- Syntactic sugar:

$$
\begin{aligned}
& {[1,2,3] \equiv(1:(2:(3:[])))} \\
& {[" a l i "] \equiv(" a l i ":[])}
\end{aligned}
$$

## L Recursive Types

$\left\llcorner_{\text {General Recursive Types }}\right.$

## General Recursive Types

■ $T=\ldots T \ldots$
■ Formula requires a minimal solution to be representable:
$S=\ln t \times S$
Is it possible to write a single value? No minimum solution here!

- List example:
$\mathrm{x}=\operatorname{Left}(1, \operatorname{Left}(2, \mathrm{x}))$
$x \in S$ ? Yes
can we process $[1,2,1,2,1,2, \ldots]$ value?
■ Some languages like Haskell lets user define such values. All iterations go infinite. Useful in some domains though.
■ Most languages allow only a subset of $S$, the subset of finite values.


## $\square_{\text {General Recursive Types }}$

- Tree $\alpha=$ empty + node $\alpha \times$ Tree $\alpha \times$ Tree $\alpha$

$$
\begin{aligned}
\text { Tree } \alpha= & \{\text { empty }\} \cup\{\text { node }(x, \text { empty, empty }) \mid x \in \alpha\} \cup \\
& \{\operatorname{node}(x, \operatorname{node}(y, \text { empty }, \text { empty }), \text { empty }) \mid x, y \in \alpha\} \cup \\
& \{\operatorname{node}(x, \text { empty }, \operatorname{node}(y, \text { empty }, \text { empty })) \mid x, y \in \alpha\} \cup \\
& \{\operatorname{node}(x, \operatorname{node}(y, \text { empty }, \text { empty }), \operatorname{node}(z, \text { empty }, \text { empty })) \mid x, y, z \in \alpha\} \cup .
\end{aligned}
$$

- $\mathrm{C}++$ (pointers and template definition)

```
template<class Alpha>
struct Tree {
    Alpha x;
    Tree *left,*right;
```

$\}$ root;

- Haskell

```
data Tree alpha = Empty l
    Node (alpha,Tree alpha, Tree alpha)
x = Node (1,Node (2,Empty,Empty),Node(3,Empty,Empty))
y = Node(3,Empty,Empty)
```


## Strings

Language design choice:
1 Primitive type (ML):
Language keeps an internal table of strings
2 Character array (C, Pascal, ...)
3 Character list (Haskell, Prolog, Lisp)

- Design choice affects the complexity and efficiency of: concatenation, assignment, equality, lexical order, decomposition


## Type Systems

■ Types are required to provide data processing, integrity checking, efficiency, access controls. Type compatibility on operators is essential.

- Simple bugs can be avoided at compile time.
- Irrelevant operations:

$$
\begin{aligned}
& y=\text { true } * 12 ; \\
& x=12 ; x[1]=6 ; \\
& y=5 ; x \cdot a=4 ;
\end{aligned}
$$

- When to do type checking? Latest time is before the operation. Two options:
1 Compile time $\rightarrow$ static type checking
2 Run time $\rightarrow$ dynamic type checking


## Static Type Checking

- Compile time type information is used to do type checking.

■ All incompatibilities are resolved at compile time. Variables have a fixed time during their lifetime.

- Most languages do static type checking

■ User defined constants, variable and function types:
■ Strict type checking. User has to declare all types (C, C++, Fortran,...)
■ Languages with type inference (Haskell, ML, Scheme...)
■ No type operations after compilation. All issues are resolved. Direct machine code instructions.

## Dynamic Type Checking

- Run-time type checking. No checking until the operation is to be executed.

■ Interpreted languages like Lisp, Prolog, PHP, Perl, Python.

- A hypothetical language:

```
int whichmonth(input) {
    if (isinteger(input)) return input;
    else if (isstring(input))
        switch(input) {
        case "January": return 1;
        case "February": return 2;
        case "December": return 12;}
}
read(input) /* user input at run time? */
ay=whichmonth(input)
```

- Run time decision based on users choice is possible.

■ Has to carry type information along with variable at run time.

- Type of a variable can change at run-time (depends on the language).


## Static vs Dynamic Type Checking

- Static type checking is faster. Dynamic type checking does type checking before each operation at run time. Also uses extra memory to keep run-time type information.
■ Static type checking is more restrictive meaning safer. Bugs avoided at compile time, earlier is better.
- Dynamic type checking is less restrictive meaning more flexible. Operations working on dynamic run-time type information can be defined.


## Type Equality

- $S \stackrel{?}{\equiv} T$ How to decide?
- Name Equivalence: Types should be defined at the same exact place.
- Structural Equivalence: Types should have same value set. (mathematical set equality).
- Most languages use name equivalence.
- C example:

```
typedef struct Comp { double x, y;} Complex;
struct COMP { double x,y; };
struct Comp a;
Complex b;
struct COMP c;
a=b; /*Valid, equal types */
a=c; /* Compile error, incompatible types */
```


## Structural Equality

$S \equiv T$ if and only if:
$1 S$ and $T$ are primitive types and $S=T$ (same type),
2 if $S=A \times B, T=A^{\prime} \times B^{\prime}, A \equiv A^{\prime}$, and $B \equiv B^{\prime}$,
3 if $S=A+B, T=A^{\prime}+B^{\prime}$, and $\left(A \equiv A^{\prime}\right.$ and $\left.B \equiv B^{\prime}\right)$ or $\left(A \equiv B^{\prime}\right.$ and $\left.B \equiv A^{\prime}\right)$,
4 if $S=A \mapsto B, T=A^{\prime} \mapsto B^{\prime}, A \equiv A^{\prime}$ and $B \equiv B^{\prime}$,
5 if $S=\mathcal{P}(A), T=\mathcal{P}\left(A^{\prime}\right)$, and $A \equiv A^{\prime}$.
Otherwise $S \not \equiv T$

■ Harder to implement structural equality. Especially recursive cases.

- $T=\{$ nil $\}+A \times T, \quad T^{\prime}=\{n i l\}+A \times T^{\prime}$
$T=\{n i l\}+A \times T^{\prime}, \quad T^{\prime}=\{n i l\}+A \times T$
■ struct Circle $\{$ double $x, y, a ;\}$;
struct Square $\{$ double $x, y, a ;\}$;
Two types have a semantical difference. User errors may need less tolerance in such cases.
- Automated type conversion is a different concept. Does not necessarily conflicts with name equivalence.

```
enum Day {Mon, Tue, Wed, Thu, Fri, Sat, Sun} x;
x=3;
```


## Type Completeness

- First order values:
- Assignment
- Function parameter
- Take part in compositions
- Return value from a function

■ Most imperative languages (Pascal, Fortran) classify functions as second order value. ( C represents function names as pointers)
■ Functions are first order values in most functional languages like Haskell and Scheme .

- Arrays, structures (records)?
- Type completeness principle: First order values should take part in all operations above, no arbitrary restrictions should exist.


## C Types:

|  | Primitive | Array | Struct | Func. |
| :--- | :---: | :---: | :---: | :---: |
| Assignment | $\sqrt{ }$ | $\times$ | $\sqrt{ }$ | $\times$ |
| Function parameter | $\sqrt{ }$ | $\times$ | $\sqrt{ }$ | $\times$ |
| Function return | $\sqrt{ }$ | $\times$ | $\sqrt{ }$ | $\times$ |
| In compositions | $\sqrt{ }$ | $\sqrt{n}$ | $\sqrt{ }$ | $\times$ |

Haskell Types:
Variable definition
Function parameter Function return In compositions


Pascal Types:
Assignment
Function parameter Function return In compositions


## Expressions

Program segments that gives a value when evaluated:
■ Literals
■ Variable and constant access

- Aggregates
- Variable references
- Function calls

■ Conditional expressions

- Iterative expressions (Haskell)


## Literals/Variable and Constant Access

■ Literals: Constants with same value with their notation 123, 0755, Oxa12, 12451233L, -123.342, -1.23342e-2, 'c', '\021', "ayse", True, False

■ Variable and constant access: User defined constants and variables give their content when evaluated. int x ;
\#define pi 3.1416
$x=p i * r * r$

## Aggregates

■ Used to compose composite values lexically.

$$
\begin{aligned}
& x=(12, \text { "ali", True }) \\
& y=\{\text { name="ali", no=12\}} \\
& f=\backslash x->x * x \\
& I=[1,2,3,4]
\end{aligned}
$$

```
{-- 3 Tuple --}
{-- record --}
{-- function --}
{-- recursive type, list --}
```

- C only has aggregates at definition. There is no aggregates in the executable expressions!

```
struct Person { char name[20], int no } p = {"Ali\sqcupCin", 332314
double dizi[3][2] = {{0,1}, {1.2,4}, {12, 1.4}};
p={"VeliபCin",123412}; X/* not possible!*/
```


## Variable References

■ Variable access vs variable reference
■ value vs I-value

- pointers are not references! You can use pointers as references with special operators.
- Some languages regard references like first order values (Java, C++ partially)
- Some languages distinguish the reference from the content of the variable (Unix shells, ML)


## LExpressions

LFunction Calls

## Function Calls

- $F\left(G p_{1}, G p_{2}, \ldots, G p_{n}\right)$

■ Function name followed by actual parameter list. Function is called, executed and the returned value is substituted in the expression position.

- Actual parameters: parameters send in the call
- Formal parameters: parameter names used in function definition
- Operators can be considered as function calls. The difference is the infix notation.
- $\oplus(a, b)$ vs $a \oplus b$
- languages has built-in mechanisms for operators. Some languages allow user defined operators (operator overloading): C++, Haskell.


## Conditional Expressions

- Evaluate to different values based on a condition.

■ Haskell: if condition then $\exp 1$ else $\exp 2$ case value of $p 1->\exp 1$; $p 2->\exp 2 \ldots$
■ C: (condition) $? \exp 1: \exp 2$;

- if .. else in C is not conditional expression but conditional statement. No value when evaluated!

```
x = (a>b)?a:b;
```

$y=((a>b) ? \sin : \cos )(x) ; \quad / *$ Does it work? try yourself

■ Haskell:

```
x = if (a>b) then a else b
y = (if (a>b) then (+) else (*)) x y
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
convert a = case a of
    Left (x,rest) -> x : (convert rest)
    Empty -> []
daynumber g = case g of
    Mon -> 1
    Tue -> 2
    Sun -> 7
```

- case checks for a pattern and evaluate the RHS expression with substituting variables according to pattern at LHS.


## Iterative Expressions

■ Expressions that do a group of operations on elements of a list or data structure, and returns a value.

■ [ expr | variable <- list , condition]

- Similar to set notation in math: \{expr|var $\in$ list, condition $\}$

$$
\begin{aligned}
& x=[1,2,3,4,5,6,7,8,9,10,11,12] \\
& y=[\text { a*2 } \mid \text { a }<-x] \\
& z=[\text { a } \mid \text { a }<-x, \bmod a 3==1]
\end{aligned}
$$



## Summary

- Value and type

■ Primitive types

- Composite types

■ Recursive types
■ When to type check
■ How to type check

- Expressions

