Lecture 18-19
Data Types and Types of a Language

prof. Marco Bellia, Dip. Informatica, Università di Pisa

April 29, 2014

Data Types and Types of a Language

- Data, Data Types and Types
- Type: Generalities
- Type Systems and Type Safety
- Type Equivalence, Coercion and Cast
- Polimorphism: Ad Hoc, Generic and Subtype
- Data Structures and Expressiveness
- Memory Allocation and Deallocation

Terminology: Data, Data Types, Types

- Data: The simplest structure for introducing **values** in a program
  - Distinctive features are the ways in which they can be used:
    - Computable values;
    - Denotable values;
    - Storable values;
    - Expressible values.
  - They depend on the characteristics of the used language
- Data Types: **Collections of values** that are homogeneous in respect of the operations that can be applied on them
  - Distinctive features are the allowed operations
  - that obviously, depend on the used language
- Types: **Categorization Structures** that classify uniquely (all) the structures that occur in a program
  - They highlight the use (from the behavioral viewpoint) of each program structure
  - They may mark each program structure in a way that is useful to investigate static properties of the program

Types: Introduction

- The program below is considered a correct program in some languages, including Ansi C

Example

```
int main (int argc, char *argv[]){//cosa calcola?
  int x = 10;
  char a = 'e';
  printf("Totale da pagare in euro : %2d\n", x + a);
  return0;
  }
```

The program terminates computing :
Totale da pagare in euro : 111

- It should be evident that the programmer has made mistakes
- Nevertheless, this is the worst situation that can happen in programming
  - The program is wrong but it appears to be correct
    - The program appears correct because:
      - It has passed the compiler checks, and
      - Moreover, its computation runs by traversing apparently, **legal computation states**
    - What means legal computation state?

Types: Generalities

- What means Legal Computation State?
- A Legal Computation State of a program = is any state in which it is assumed that the program can traverse during its execution.
  - A legal State may contain exceptions of different nature ("illegal division by 0", "array outbounds",...) or
  - Anomalies in the used resources (too many active AR's, AR that consumes too much in time or in dynamic memory)
  - All these anomalies are signalled by the executor
  - Any program can be enriched to recover from these anomalies
- A Non-legal Computation State (or Stuck) = is any state that no correct execution of the program should reach
  - For instance, the sum of the effective amount, 10, with the integer representing the tag e of the symbol for euros.
  - No program can recognizes such an unexpected situation.
  - Hence, no recover code may be written for the program
  - The program is dangerous: The program must be rewritten

Type System (for Property Investigations

It consists of 3 structures:

- The Domain of the Types (including the basic language types)
- The (language of) Type Expressions with which new (derived) types can be defined
- The Rules with which the language associates one type to each structure of a program of the language

The type system F1 the Typed Lambda Calculus is shown below

- **Types:** A,B ::= k | A --> B | A + B | A x B | Unit   -- *in Haskell Unit is ()*
- **Expressions:** M,N ::= x | \x:A->M | M N | (M,N) | first M | second M | unit
- **Rules:**

(env ∅)
$$\frac{}{\varnothing \vdash \lozenge}$$

(env x)
$$\frac{\Gamma \vdash A \quad x \notin dom(\Gamma)}{\Gamma, x{:}A \vdash \lozenge}$$

(Tipo base)
$$\frac{\Gamma \vdash \lozenge \quad A \in k}{\Gamma \vdash A}$$

(val x)
$$\frac{\Gamma_1, x{:}A, \Gamma_2 \vdash \lozenge}{\Gamma_1, x{:}A, \Gamma_2 \vdash x{:}A}$$

Type System: F1 - NO

- The system F1 apply to the programs of the Typed Lambda Calculus:
- If the program passes the checks then the program never gets stuck

(Tipo ->)
$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \to B}$$

(val fun)
$$\frac{\Gamma, x{:}A \vdash M{:}B}{\Gamma \vdash \backslash x{:}A \to M : A \to B}$$

(val appl)
$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

(Tipo Unit)
$$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \mathrm{Unit}}$$

(Val Unit)
$$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \mathrm{unit} : \mathrm{Unit}}$$

(Tipo prodotto)
$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B}$$

(Val prodotto)
$$\frac{\Gamma \vdash N{:}A \quad \Gamma \vdash M{:}B}{\Gamma \vdash (N,M) : A \times B}$$

(val first)
$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathrm{first}\ M : A}$$

(val second)
$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathrm{second}\ M : B}$$

- A program `P` is correctly typed if and only if one type `T` exists such that:

  $$\emptyset \vdash \mathtt{P} : \mathtt{T}$$

  holds in F1 (i.e. can be obtained by using the rules of F1)

Type Safety or Soundness = Progress + Preservation

- A language can have a Type System which guarantees that the language programs are Type Safe (Haskell, Ocaml, Java)
  - **Progress.** It ensures that the execution of a **well typed program**, [$\vdash \mathtt{P} : \mathtt{T}$], never gets stuck. It reaches: [below, $\rightarrow$ = 1 program computation step]
    - Either a final states with the expected values, [$\mathtt{P} \in \mathtt{Val}$];
    - Or a new legal state, [$\vdash \mathtt{P} \rightarrow \mathtt{P}'$] and [$\vdash \mathtt{P}' : \mathtt{T}'$], for some $\mathtt{T}'$.
  - **Preservation or Subject Reduction.** A **well typed program**, [$\vdash \mathtt{P} : \mathtt{T}$], leaves unchanged its type and that of tits components during the execution [if $\vdash \mathtt{P} \rightarrow \mathtt{P}'$ then, $\vdash \mathtt{P}' : \mathtt{T}'$ and $\mathtt{T} = \mathtt{T}'$].
- Checking for Type Safety can be done **statically** (Haskell, Ocaml, Java)
  - at compile time
  - strong typing: Programs that do not pass the check are rejected
  - types are no more useful during computation: then are removed form the object code
- Checking for Type Safety is done **only dynamically** (C++),
  - Executor checks the operands for the right type
  - Wrong Programs are stopped only when the type check fails
  - Types are maintained in the object code
- Checking for Type Safety **cannot be** done (C)
  - Dangerous program run without anyone noticing

Type Expressions

Type Expressions allow the definition of new types:

- using type operators that may include product (record or struct) ["*",in Ocaml], sum (union) ["|", in Ocaml), map ["→",in Ocaml];
- using type constructor;

Example

State = Env * Store `//naming, product`
Env = Ide → Den `//naming, map`
Store = M(Loc * Loc → Mem) `//naming, type constructor, product`
Enum = A | B | C `//naming, sum, type constructors`

- using generic polymorphism, i.e. type expressions with type variables (universally qualified variables, raging over the domain of the types);
- using naming and recursively defined types;

Example

('a,'b)Env = 'a → 'b `//type variables, naming, map`
'a list = Cons 'a ('a list) | Nil `//type variable, type constructors, sum, recursive types`

Relations and Properties on the type structures

- **Equivalence:**. Let x:T and y:T': Are x and y of the same type?
  - **Nominal:** It is equivalent only to itself
  - **Structural** Same type expression when replacing names by definitions
- **Subtype**
  - Explicit (Java) or
  - Implicit (contra-co-variance of functions)
- **Coercion**: Conversion of a value representation and consequent change of type;
- **Cast**: Type constraint that must be satisfied at run time;
- **Overloading** Different types and values for a same (Function) identifier;
- **Subtype Polymorphism** In combination with the subtypes: A method also applies to values of subtypes of the types expected.

Data: Generalities

- Structure:
  - Scalar Data are atomic values: Only operations for computing new values
  - Structured Data: Have in addition, components and operations for visiting (and possibly, modifying) components
- Mutability
  - Scalar Data are not mutable values, but may be used in mutable values
  - Structured Data may be mutable or not, may have mutable components
  - In functional languages, Structured Data are immutable with immutable components
- Expressiveness
  - Scalar Data are in general, expressible values
  - In Imperative Languages, Structured Data are not expressible values
  - In functional languages, all data are alway expressible values
  - Ocaml (in addition to scalar data, functions, tuples, and lists) has mutable, arrays and records values: All such data are expressible values
- Allocation
  - Static: At Compile/Loading Time
  - Dynamic
    - (for intermediate values) in the stack RI, locally to the AR's
    - Heap: With program controlled allocation/deallocation
    - In a pool memory (Heap): With automatic allocation/deallocation
    - In functional languages, is automatic and made transparent to computation.

Dynamic Allocation and Expressiveness - NO

- User Controlled Allocation: The constructor implementation is under user responsibility
- Automatic Allocation: The constructor implementation is automatically provided by the language

Example

```
struct elem {
  int hd;
  struct elem *tl;
};
typedef struct elem *list;

list Cons(int v, list n){
  list r = malloc(sizeof(struct elem));
  r->hd = v;
  r->tl = n;
  return r;
}//User Controlled - in Ansi-C
```

```
type list = NULL | Cons of int * list;;
                 //Automatic in Ocaml
```

- Expressiveness: Data Constructors furnish a **Data Presentation** of the values

Example

```
int main (int argc, char *argv[]){
  list r = Cons(3,NULL);
  r = Cons(2,r);
  }
```

```
# let r = ref (Cons(3,NULL));;
val r :  list ref = {contents = Cons (3, NULL)}
# (r:= Cons(2,!r); !r);;
- :  list = Cons (2, Cons (3, NULL))
```

Controlled and Automatic, Dynamic, Deallocation - NO

Controlled Deallocation. The Tombstone Technique: The access to the dynamically allocated structure is through a guard, called Tombstone.

- They are marked when the structure is released (dispose, free): Dangling References
- Locks and Keys are used to re-allocate the memory and to recognize accesses from dangling references

Automatic Deallocation (Garbage Collector). The Counter Technique: A counter, `C(V)` is set to 1 when a new structure is created and with reference `V` (e.g. malloc...)

- Whenever a "copy" (i.e. assignment, parameter passing by value,...) with source p_s and target p_t is made:
  - $C(V_{p_s})$++, where V_{p_s} be the dynamic allocated structure of reference p_s;
  - $C(V_{p_t})$--, where V_{p_t} be the dynamic allocated structure of reference p_t;
- With each pop of an AR (end/exit of an inline block, or return/exit of a procedure)
  - We consider all local variables and for each of them, the possibly reference to a dynamically allocated structure or to a component of it. Let $\{p_{s_i}\}$ be the set of them.
  - $C(V_{p_{s_i}})$--, for each i
  - If $C(V_{p_{s_i}}) == 0$, then the structure $V_{p_{s_i}}$ is deallocated.

Excercises

Exercise1.

a What kind of type equivalence is defined in Ocaml?;
b Show an example that confirms it.

Exercise2.

a What kind of type equivalence is defined in Java?;
b Show an example that confirms it.

Exercise3.

a Write in Ocaml, a type for data representing Activation Records in static languages;
b Write in Java, a type for data representing Activation Records in static languages;
b Write in C, a type for data representing Activation Records in static languages;