Types

Angelo Gargantini
informatica 3
Università di Bergamo

Type

A type is a collection of computable values that share some structural property.

- Examples
  - Integers
  - Strings
  - $int \to bool$
  - $(int \to int) \to bool$
- "Non-examples"
  - ∀ $\{3, true, \lambda x.x\}$
  - Even integers
  - ∀ $\{f:int \to int \mid \text{if } x>3 \text{ then } f(x) > x*(x+1)\}$

Distinction between types and non-types is language dependent.

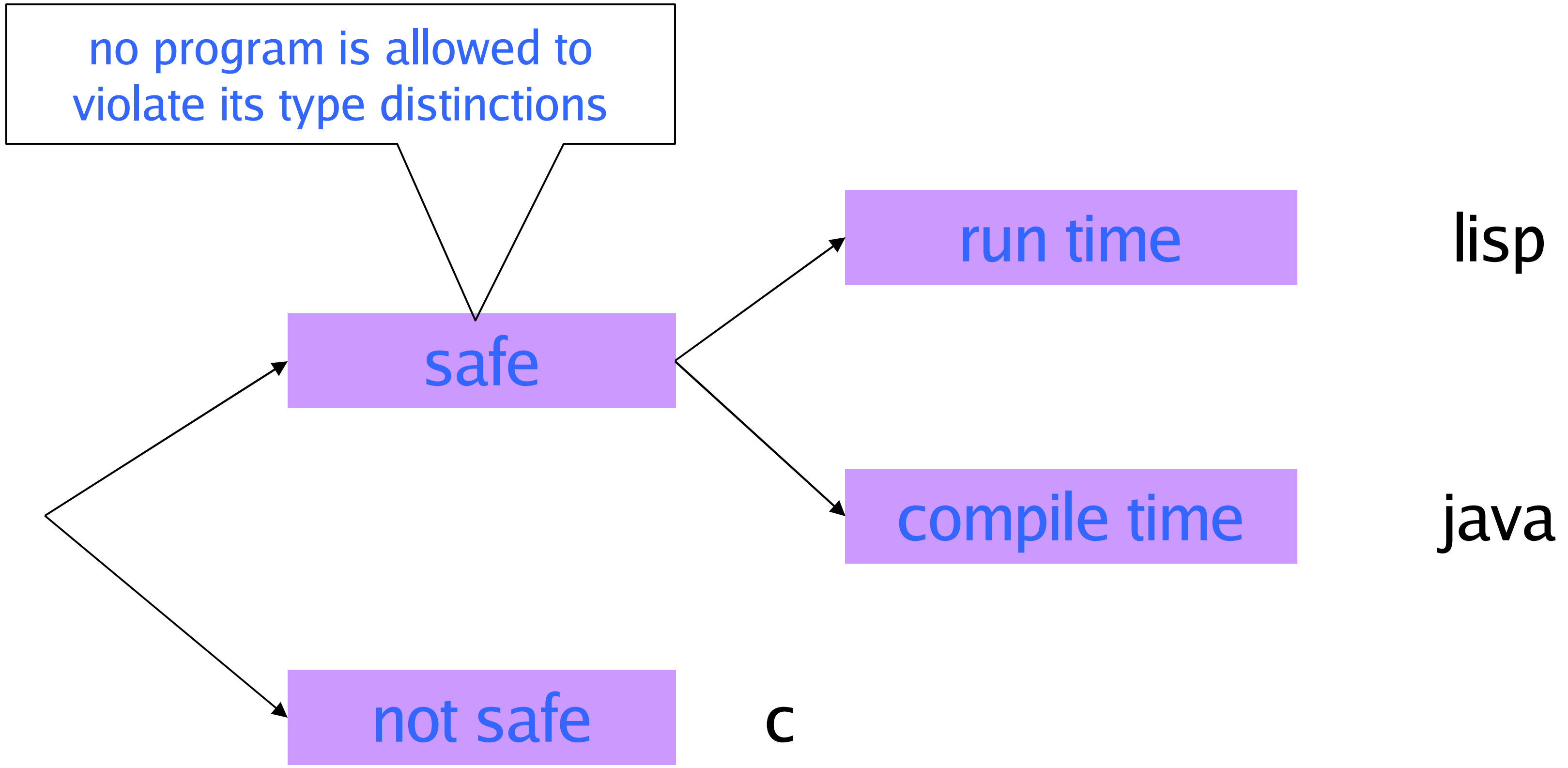
Uses for types

- Program organization and documentation
  - Separate types for separate concepts
    - Represent concepts from problem domain
  - Indicate intended use of declared identifiers
    - Types can be checked, unlike program comments
- Identify and prevent errors
  - Compile-time or run-time checking can prevent meaningless computations such as 3 + true - “Bill”
- Support optimization
  - Example: short integers require fewer bits
  - Access record component by known offset

Type errors

- VEDI LUCIDI PDF

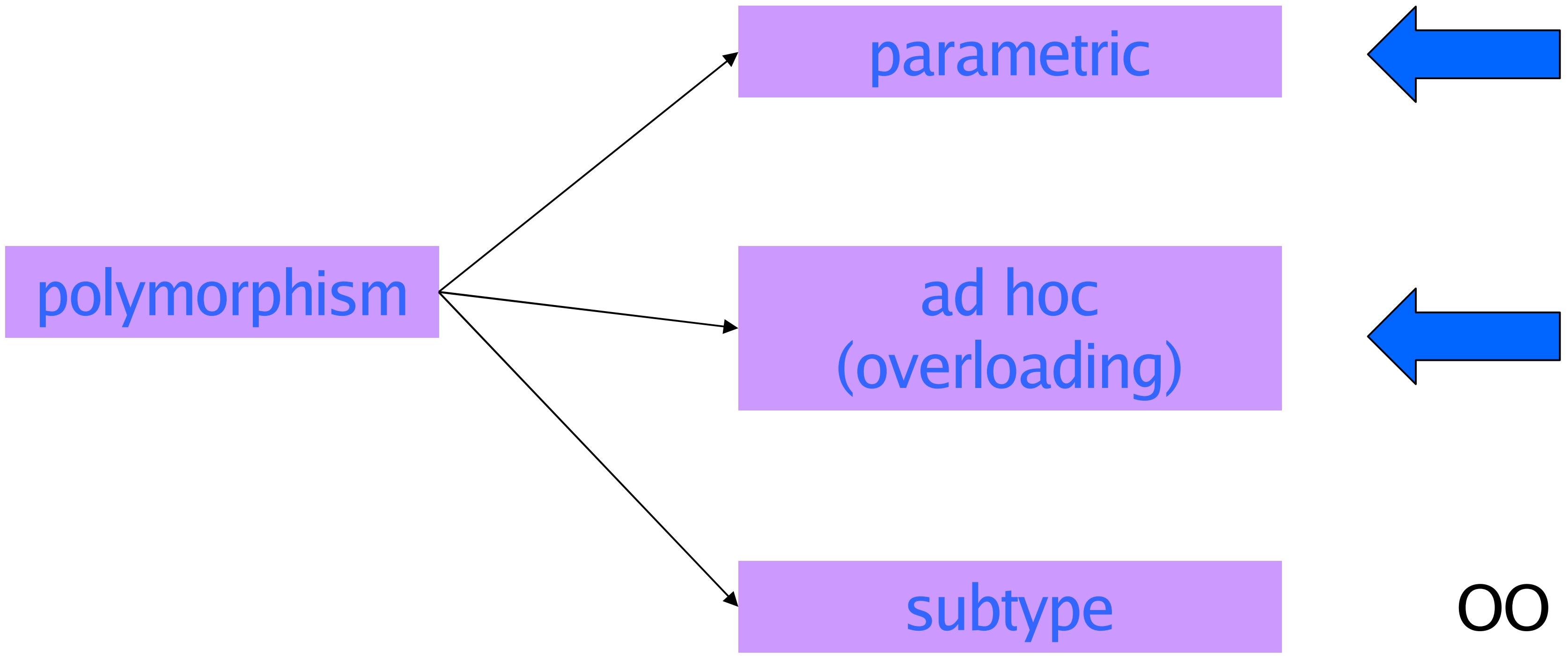
classificazione



Compare C++ templates

- Sec 6.4.1 – Parametric polymorphism
- Sec 6.4.2 – Implementation of parametric poly

Polymorphism



Polymorphism vs Overloading

- Parametric polymorphism
  - Single algorithm may be given many types
  - Type variable may be replaced by *any* type
  - f : t→t => f : int→int, f : bool→bool, ...
- Overloading
  - A single symbol may refer to more than one algorithm
  - Each algorithm may have different type
  - Choice of algorithm determined by type context
  - Types of symbol may be arbitrarily different
  - + has types int*int→int, real*real→real, *no others*

Parametric Polymorphism: ML vs C++

- ML polymorphic function
  - Declaration has no type information
  - Type inference: type expression with variables
  - Type inference: substitute for variables as needed
- C++ function template
  - Declaration gives type of function arg, result
  - Place inside template to define type variables
  - Function application: type checker does instantiation

ML also has module system with explicit type parameters

Example: swap two values

- ML
  - fun swap(x,y) =

    let val z = !x in x := !y; y := z end;

  val swap = fn : 'a ref * 'a ref -> unit

- C++ (parametric functions)

```
template <typename T> void swap(T& x, T& y){
    T tmp = x;  x=y;  y=tmp;
}
```

Declarations look similar, but compiled is very differently

template: example

```
int i,j; . . .
swap(i,j); // Use swap with T replaced with int

float a,b; . . .
swap(a,b); // Use swap with T replaced with float

String s,t; . . .
swap(s,t); // Use swap with T replaced with String
```

Implementation

- ML
  - Swap is compiled into one function
  - Typechecker determines how function can be used
- C++
  - Swap is compiled into linkable format
  - Linker duplicates code for each type of use
- Why the difference?
  - ML ref cell is passed by pointer, local x is pointer to value on heap
  - C++ arguments passed by reference (pointer), but local x is on stack, size depends on type

Another example

- C++ polymorphic sort function

```
template <typename T>
void sort( int count, T * A) {
    for (int i=0; i<count-1; i++)
        for (int j=i+1; j<count-1; j++)
            if (A[j] < A[i]) swap(A[i],A[j]);
}
```

- What parts of implementation depend on type?
  - Indexing into array
  - Meaning and implementation of <

Another example (with error)

```
template <typename T>
        T diff(T& x, T& y){  return x - y; }

int main (){
  int i = 0 ,j = 1;
  cout<<“delta  :"<< diff(i,j);
  float a= 2.5,b=6.3;
  cout<<" delta :"<< diff(a,b);
  string s1 = "s1";   string s2 = "s2";
  cout<<" delta :"<< diff(s1,s2);
}
```

Generics methods in Java

Consider writing a method that takes an array of objects and a collection and puts all objects in the array into the collection. Here's a first attempt:

```
static void fromArrayToCollection(Object[] a,
    Collection<?> c) {
    for (Object o : a) {
        c.add(o); // Compile time error
                  // Collection<?> cannot store Object
    }
}
```

Solution

A generic method defines one or more type parameters in the method signature, before the return type:

```
static <T> void fromArrayToCollection(T[] a,
                                  Collection<T> c) {
    for (T o : a) {
        c.add(o); // Correct
    }
}
```

Posso usarla così:

```
String[] sa = new String[100];
Collection<String> cs = new ArrayList<String>();
fromArrayToCollection(sa, cs); // T is String
```

Altri esempi

- static <T> boolean
  myMethod(List<? extends T>, T obj)
- Visitor
- Se volessi fare un metodo che restituisci il max di una collezione?
- I soluzione – uso compareTo:

```
<T> T max(Collection<T> c){
    T max = null;
    for(T a: c){
        if ( a.compareTo(max) >0 ) max = a;
    }
    return max;
}
```

Come faccio ad assicurarmi che T ha compareTo?

soluzioni

1) chiedo che T estenda comparable –

```
<T extends Comparable<T>>
                T max(Collection<T> c){ ...
```

in questo modo il controllo che T abbia compareTo sarà fatto compile time

2) converto T a comparable:_

```
<T> T max(Collection<T> c){
        ...
        ((Comparable<T>)a).compareTo
```

il controllo viene fatto run-time: posso compilare con T qualsiasi ma a run time dovrà essere un comparable

ML Overloading

- Some predefined operators are overloaded
- User-defined functions must have unique type

```
- fun plus(x,y) = x+y;
> Error: overloaded variable cannot be resolved: +
```

- Why is a unique type needed?
  - Need to compile code $\Rightarrow$ need to know which +
  - Efficiency of type inference
  - Aside: General overloading is NP-complete

    Two types, *true* and *false*

    Overloaded functions

    and : {*true***true*$\rightarrow$*true*, *false***true*$\rightarrow$*false*, …}

Main Points about ML

- General-purpose procedural language
  - We have looked at “core language” only
  - Also: abstract data types, modules, concurrency,….
- Well-designed type system
  - Type inference
  - Polymorphism
  - Reliable -- no loopholes
  - Limited overloading

TYPE DECLARATIONS & equality in C

- leggi sezione 6.5 del libro (salta costrutti ML)
- Studia 6.5.2
- The basic type declaration in C is typedef:

```
typedef <oldtype> <newtype>
typedef char byte;
typedef int numero;
typedef byte ten_bytes[10];
typedef struct {...}   Persona;
```

equality

- Without struct different types defined in the same way are compatible:

```
typedef int TA; typedef int TB;
TA x; TB y;
x = y; //OK
```

- With struct, different typedef are different:

```
typedef struct {int m} TA;
typedef struct {int m} TB;
TA x; TB y;
x = y; // ERRORE
```