Programming Language Design

2015

Week #7: Type Systems

Instructor: Hidehiko Masuhara

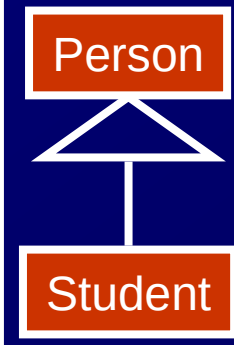
Quiz (1/2) (10 min.)

1. Define "types"
2. Define "(static) type checking"
3. Define "type safety"
4. Write a program in any language that does not pass type checking
5. Illustrate a programming language that does not guarantee type safety, and show a program as an evidence

Quiz (2/2) (5 min.)

1. What happens if it is compiled?
2. (assuming is was successfully compiled) What happens if it is executed?

```
// Java
void setPerson(Person[] pp, int i, Person p) {
  pp[i]=p;
}
void main(){
  Student[] ss = new Student[1];
  setPerson(ss, 0, new Person("Alice"));
}
```



Glossary

- A type: a set of data manipulated by programs
  - eg: integers, real numbers, Person class
- A type error: giving an operator a data of a type that cannot be handled by the operator
  - eg: 1 + “hello”, sqrt(“hello”), 123.getName()
- (Static) type checking: to determine whether a program may cause a type error without executing the program
- Type safety: a property that any statically type-checked program never causes a type error

Type systems in classical PLs

- No type-checking
  - so called "dynamically typed languages"
  - Lisp, Perl, Ruby, Python, Smalltalk, Javascript...
  - attach a type information to each data, and check at runtime
- Incomplete type-checking
  - COBOL, FORTRAN, Algol, Pascal, C, ...
  - Incompleteness
    - Casting: float f = 123.45; int *i = (int*)&f;
    - Array bound: float f[1]; int i[1]; f[2]=123.45;
    - Unions: union { int i; float f; } u; u.f=123.45;

Why not complete?
•easier
•on purpose
•for efficiency

Classical type-checking

"expression" means expressions and statements

- Type-checking rules
  - Rule: (for each type of expression) permitted combinations of types of subexpressions and a type of result
    - eg: if e1 then e2 else e3 ➔ e1 is bool, e2 and e3 have the same result type
  - Implementation: 1. typing each expression, 2. check against the rule
- Typing (giving a type to an expression)
  - Types of variables & functions are declared in a program
  - Interpret the type-checking rule as "deriving a result type from types of subexpressions"
    eg. if e1 then e2 else e3 ➔ result type = result type of e2

Designing real languages with type-safety is not easy

- A hypothetical language TypeSafe
  - Data types: int, double, string, boolean, array of Type, record Name of { Label: Type; Label: Type; ... }
    - eg. "array of int", "record pt of { x: int; y: int }"
  - Syntax:
    Decl ::= Type Name(Type: Var;...) { Stmt; Stmt;...}
    Stmt ::= Assign|VarDecl|FunCall| IfThenElse | WhileDO | return Exp | Exp
- Easy to guarantee type safety of TypeSafe
- If it is not easy for real languages, what are missing in TypeSafe?

Difficulty: code that handles multiple types (polymorphic types)

eg.,

- Polymorphic procedures:
  - "sorting procedure that can sort integer arrays as well as string arrays"
  - "sorting procedure that can sort arrays of comparable values"
  - passing a Student object to a method that receives Person objects
- Abstract datatypes:
  - "stack of (only) integers, stack of (only) strings"

Those are not easy to be type-checked

Difficulty: code that handles multiple types (polymorphic types)

eg..

- ◆ eg: if e1 then e2 else e3 ➔ e1 is bool, e2 and e3 have the same result type
- ➢ Implementation: 1. typing ... ainst the rule

Typing (giving a type ...

- ➢ Types of variables & function ... gram
- ➢ Interpret the type-checking rule as "deriving a result type from types of subexpressions"
  eg. if e1 then e2 else e3 ➔ result type = result type of e2

assuming types can be determined from inside to outside

Those are not easy to be type-checked

Approach to polymorphic types: subtyping and type-checking

- Subtyping: B is subtype of A when a set of B-values is a subset of A-values
  - Subtype polymorphism: code defined for values of a type can be used for values of any subtype
- Type-checking
  - for a variable of type A, permit to assign a value of A's subtype (eg, pass a Student to a method that prints Persons)
  - when values of two different types are merged, the merged value will have a least-upper-bound of those types (eg: "if … then new Student() else new Teacher()" has a result of Person type)

Subtyping and compound types

- Compound type:
  a type constructed by using other types
  - eg. "arrays of integer", "sets of Person", "functions that take a number and return a Person"
- Subtype relation of compound types: not obvious
  - eg. when Student <: Person, any of them can make some sense
    - Student[] <: Person[] — called *covariant*
    - Student[] :> Person[] — called *contravariant*
    - Student[] ≠ Person[] — called *invariant*

Subtyping and assignments

- class Cell { // a storage of a Person

```
  Person p;
  void set(Person p) { this.p = p; }
  Person get() { return this.p; }
}
```

- Can we use it for Teachers?

```
Cell teacherCell = new Cell();
teacherCell.set(new Teacher());
teacherCell.get().teach(aClass);
```

no teach in Person!

- A solution (in old Java) : cast the return value of get() as Teacher → not type-safe

Arrays in Java: covariant but not safe

- This can be compiled without errors:
  - void set(Person[] cell, Person p) {
    cell[0]=p;
    }
  - Teacher[] tCell = new Teacher[1];
    set(tCell, new Student());
  - Reason: in Java, T <: P then T[] <: P[]
- Consequence: at runtime, this raises an exception of "assigning a Student to a Teacher array"
- Why Java doesn't reject this code?

A solution: Templates

- Template: a program unit that generates definitions like classes and functions when a type is applied to it (C++)
  - eg.
    ```
    template <typename T>
    class Stack {
      T[] elements;
      void push(T elm) { … }
      T pop() { … } };
    ```
  - Stack<Student> s = new Stack<Student>();
- Definition is generated for each type →polymorphism without subtyping

Problems in Templates

What's wrong with this program?

```
class Person {};
int main () {
  std::vector<int> is = { 3,1,4,1,5,9,2 };
  std::sort(is.begin(), is.begin()+7);

  std::vector<Person> pp = { Person(), Person() };
  std::sort(pp.begin(), pp.begin()+1);
}
```

```
In file included from /usr/include/c++/4.6/algorithm:63:0,
                 from v.cpp:2:
/usr/include/c++/4.6/bits/stl_algo.h: In function 'void std::__insertion_sort(_RandomAccessIterator,
_RandomAccessIterator) [with _RandomAccessIterator = __gnu_cxx::__normal_iterator<Person*,
std::vector<Person> >]':
/usr/include/c++/4.6/bits/stl_algo.h:2181:4:   instantiated from 'void
std::__final_insertion_sort(_RandomAccessIterator, _RandomAccessIterator) [with _RandomAccessIterator =
__gnu_cxx::__normal_iterator<Person*, std::vector<Person> >]'
/usr/include/c++/4.6/bits/stl_algo.h:5409:4:   instantiated from 'void std::sort(_RAIter, _RAIter) [with _RAIter =
__gnu_cxx::__normal_iterator<Person*, std::vector<Person> >]'
v.cpp:8:37:   instantiated from here
/usr/include/c++/4.6/bits/stl_algo.h:2107:4: error: no match for 'operator<' in '__i.__gnu_cxx::__normal_iterator<_Iterator, _Container>::operator* [with
_Iterator = Person*, _Container = std::vector<Person>, __gnu_cxx::__normal_iterator<_Iterator, _Container>::reference = Person&]() <
__first.__gnu_cxx::__normal_iterator<_Iterator, _Container>::operator* [with _Iterator = Person*, _Container = std::vector<Person>,
__gnu_cxx::__normal_iterator<_Iterator, _Container>::reference = Person&]()'
/usr/include/c++/4.6/bits/stl_algo.h:2107:4: note: candidates are:
/usr/include/c++/4.6/bits/stl_pair.h:207:5: note: template<class _T1, class _T2> constexpr bool std::operator<(const std::pair<_T1, _T2>&, const std::pair<_T1, _T2>&)
/usr/include/c++/4.6/bits/stl_iterator.h:291:5: note: template<class _Iterator> bool std::operator<(const std::reverse_iterator<_Iterator>&, const
std::reverse_iterator<_Iterator>&)
/usr/include/c++/4.6/bits/stl_iterator.h:341:5: note: template<class _IteratorL, class _IteratorR> bool std::operator<(const std::reverse_iterator<_IteratorL>&, const
std::reverse_iterator<_IteratorR>&)
/usr/include/c++/4.6/bits/stl_iterator.h:1049:5: note: template<class _IteratorL, class _IteratorR> bool std::operator<(const std::move_iterator<_IteratorL>&, const std::move_iterator<_IteratorR>&)
/usr/include/c++/4.6/bits/stl_iterator.h:1055:5: note: template<class _Iterator> bool std::operator<(const std::move_iterator<_Iterator>&, const std::move_iterator<_Iterator>&)
/usr/include/c++/4.6/bits/stl_vector.h:1290:5: note: template<class _Tp, class _Alloc> bool std::operator<(const std::vector<_Tp, _Alloc>&, const std::vector<_Tp, _Alloc>&)
/usr/include/c++/4.6/bits/basic_string.h:2510:5: note: template<class _CharT, class _Traits, class _Alloc> bool std::operator<(const std::basic_string<_CharT, _Traits, _Alloc>&, const std::basic_string<_CharT, _Traits, _Alloc>&)
/usr/include/c++/4.6/bits/basic_string.h:2522:5: note: template<class _CharT, class _Traits, class _Alloc> bool std::operator<(const std::basic_string<_CharT, _Traits, _Alloc>&, const _CharT*)
/usr/include/c++/4.6/bits/basic_string.h:2534:5: note: template<class _CharT, class _Traits, class _Alloc> bool std::operator<(const _CharT*, const std::basic_string<_CharT, _Traits, _Alloc>&)
/usr/include/c++/4.6/tuple:586:5: note: template<class ... _TElements, class ... _UElements> bool std::operator<(const std::tuple<_TElements ...>&, const std::tuple<_Elements ...>&)
/usr/include/c++/4.6/bits/stl_algo.h: In function 'void std::__heap_select(_RandomAccessIterator, _RandomAccessIterator, _RandomAccessIterator) [with _RandomAccessIterator = __gnu_cxx::__normal_iterator<Person*, std::vector<Person> >]':
/usr/include/c++/4.6/bits/stl_algo.h:5254:7:   instantiated from 'void std::partial_sort(_RAIter, _RAIter, _RAIter) [with _RAIter = __gnu_cxx::__normal_iterator<Person*, std::vector<Person> >]'
/usr/include/c++/4.6/bits/stl_algo.h:2279:8:   instantiated from 'void std::__introsort_loop(_RandomAccessIterator, _RandomAccessIterator, _Size) [with _RandomAccessIterator = __gnu_cxx::__normal_iterator<Person*, std::vector<Person> >, _Size = int]'
/usr/include/c++/4.6/bits/stl_algo.h:5407:4:   instantiated from 'void std::sort(_RAIter, _RAIter) [with _RAIter = __gnu_cxx::__normal_iterator<Person*, std::vector<Person> >]'
v.cpp:8:37:   instantiated from here
/usr/include/c++/4.6/bits/stl_algo.h:1905:2: error: no match for 'operator<' in '__i.__gnu_cxx::__normal_iterator<_Iterator, _Container>::operator* [with _Iterator = Person*, _Container = std::vector<Person>, __gnu_cxx::__normal_iterator<_Iterator, _Container>::reference = Person&]() <
__first.__gnu_cxx::__normal_iterator<_Iterator, _Container>::operator* [with _Iterator = Person*, _Container = std::vector<Person>, __gnu_cxx::__normal_iterator<_Iterator, _Container>::reference = Person&]()'
/usr/include/c++/4.6/bits/stl_algo.h:1905:2: note: candidates are:
/usr/include/c++/4.6/bits/stl_pair.h:207:5: note: template<class _T1, class _T2> constexpr bool std::operator<(const std::pair<_T1, _T2>&, const std::pair<_T1, _T2>&)
/usr/include/c++/4.6/bits/stl_iterator.h:291:5: note: template<class _Iterator> bool std::operator<(const std::reverse_iterator<_Iterator>&, const std::reverse_iterator<_Iterator>&)
/usr/include/c++/4.6/bits/stl_iterator.h:341:5: note: template<class _IteratorL, class _IteratorR> bool std::operator<(const std::reverse_iterator<_IteratorL>&, const std::reverse_iterator<_IteratorR>&)
/usr/include/c++/4.6/bits/stl_iterator.h:1049:5: note: template<class _IteratorL, class _IteratorR> bool std::operator<(const std::move_iterator<_IteratorL>&, const std::move_iterator<_IteratorR>&)
/usr/include/c++/4.6/bits/stl_iterator.h:1055:5: note: template<class _Iterator> bool std::operator<(const std::move_iterator<_Iterator>&, const std::move_iterator<_Iterator>&)
/usr/include/c++/4.6/bits/stl_vector.h:1290:5: note: template<class _Tp, class _Alloc> bool std::operator<(const std::vector<_Tp, _Alloc>&, const std::vector<_Tp, _Alloc>&)
/usr/include/c++/4.6/bits/basic_string.h:2510:5: note: template<class _CharT, class _Traits, class _Alloc> bool std::operator<(const std::basic_string<_CharT, _Traits, _Alloc>&, const std::basic_string<_CharT, _Traits, _Alloc>&)
/usr/include/c++/4.6/bits/basic_string.h:2522:5: note: template<class _CharT, class _Traits, class _Alloc> bool std::operator<(const std::basic_string<_CharT, _Traits, _Alloc>&, const _CharT*)
/usr/include/c++/4.6/bits/basic_string.h:2534:5: note: template<class _CharT, class _Traits, class _Alloc> bool std::operator<(const _CharT*, const std::basic_string<_CharT, _Traits, _Alloc>&)
/usr/include/c++/4.6/tuple:586:5: note: template<class ... _TElements, class ... _UElements> bool std::operator<(const std::tuple<_TElements ...>&, const std::tuple<_Elements ...>&)
/usr/include/c++/4.6/bits/stl_algo.h: In function 'void std::__move_median_first(_Iterator, _Iterator, _Iterator) [with _Iterator = __gnu_cxx::__normal_iterator<Person*, std::vector<Person> >]':
/usr/include/c++/4.6/bits/stl_algo.h:2252:7:   instantiated from '_RandomAccessIterator std::__unguarded_partition_pivot(_RandomAccessIterator, _RandomAccessIterator) [with _RandomAccessIterator = __gnu_cxx::__normal_iterator<Person*, std::vector<Person> >]'
/usr/include/c++/4.6/bits/stl_algo.h:2284:54:   instantiated from 'void std::__introsort_loop(_RandomAccessIterator, _RandomAccessIterator, _Size) [with _RandomAccessIterator = __gnu_cxx::__normal_iterator<Person*, std::vector<Person> >, _Size = int]'
/usr/include/c++/4.6/bits/stl_algo.h:5407:4:   instantiated from 'void std::sort(_RAIter, _RAIter) [with _RAIter = __gnu_cxx::__normal_iterator<Person*, std::vector<Person> >]'
v.cpp:8:37:   instantiated from here
/usr/include/c++/4.6/bits/stl_algo.h:86:7: error: no match for 'operator<' in '__a.__gnu_cxx::__normal_iterator<_Iterator, _Container>::operator* [with _Iterator = Person*, _Container = std::vector<Person>, __gnu_cxx::__normal_iterator<_Iterator, _Container>::reference = Person&]() <
__b.__gnu_cxx::__normal_iterator<_Iterator, _Container>::operator* [with _Iterator = Person*, _Container = std::vector<Person>, __gnu_cxx::__normal_iterator<_Iterator, _Container>::reference = Person&]()'
/usr/include/c++/4.6/bits/stl_algo.h:86:7: note: candidates are:
```

Problems in Templates

What's wrong with this program?

sort assumes that elements can be compared by <, but the type sort can't tell that

```
class Person {};
int main () {
  std::vector<int> is = { 3,1,4,1,5,9,2 };
  std::sort(is.begin(), is.begin()+7);

  std::vector<Person> pp = { Person(), Person() };
  std::sort(pp.begin(), pp.begin()+1);
}
```

```
template <class RandomAccessIterator>
  void sort (RandomAccessIterator first, RandomAccessIterator last);
```

Another solution: generic types

- Generic type: a compound type that contains ***type variables*** (Java)
  - eg.
    ```
    class Stack<T> {
      T[] elements;
      void push(T elm) { … }
      T pop() { … } };
    ```
  - Stack<Student> s = new Stack<Student>();
- Difference from templates
  - Stack is compiled only once (cf. stack template generates definitions for each element type)
  - definition and use of Stack are separately type-checked (cf. errors in templates can only be reported after expansion)

Type constraints in generic types

- Type variables can be constrained
  - eg.
    interface Comparable { boolean compare(...) ; }
    class Algorithm<**T extends Comparable**> {
      void sort(Collection<T> lst) { ... } }
  - class Person {}
    Algorithm.sort(new Vector<Person>());

requires T <: Comparable
→ requires compare in T

error because it is
not Person <: Comparable
→ detecting absence of compare

Subtyping for generic types

- eg
  - class Cell<T> { //a storage for T
    T p;
    void set(T p) { this.p = p; }
    T get() { return this.p; }
    }
  - void getAndPrint(Cell<Person> cellP) {
    print(cellP.get().getName()); }
  - Cell<Student> c = new Cell<Student>();
    c.set(new Student());
    getAndPrint(c);
- Do we want to write this kind code?
- What subtype rules are needed to check this kind code?
- Any problems with such rules?

What if generic types are covariant?

- eg.
  - class Cell<T> { // a storage of T
    T p;
    void set(T p) { this.p = p; }
    T get() { return this.p; }
    }
  - void work(Cell<Person> cellP) {
    print(cellP.get().getName()); }
  - Cell<Student> c = new Cell<Student>();
    c.set(new Student());
    work(c);
- covariant — S <: P then Cell<S> <: Cell<P>
  This code is type-checked
- when work is change to･･･?
  void work(Cell<Person> cellP) {
  cellP.set(new Teacher()); }

A solution in Java: wildcard

- eg

```
void getAndPrint(Cell<? extends Person> cellP) {
    print(cellP.get().getName()); }
```

- semantics: element type of Cell is ***one of subtypes*** of Person
  - return type of cellP.get() is not fixed, but at least a subtype of Person
    →.getName() is correct
  - what if cellP.set(new Student()); ?
    →argument to cellP.set is one of Person subtypes
    →incorrect because it is not always Student

A solution in Java: wildcard

- Another kind of constraints:

```
void work(Cell<Student super ?> cellP) {
   cellP.set(new Student());
}
```

- Semantics
  - Element type of Cell is supertype of Student (contravariant)
- Purpose
  - Cell<Person> cp = new Cell<Person>();
    work(cp);
  - Element type T of cellP is not fixed, yet T is supertype of Student
  - argument type of cellP.set is T, hence correct to pass Student

Type-checking generic types in Java

- Not easy
  - theoretically, they correspond to existential types with F-bounded polymorphism
    Igarashi, Atsushi, and Mirko Viroli. "Variant parametric types: A flexible subtyping scheme for generics." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28.5 (2006): 795-847.