3. Subtyping and Parametric Types

Overview:

- Typing of objects
- Subtype polymorphism
- A Generic type system
- Virtual types

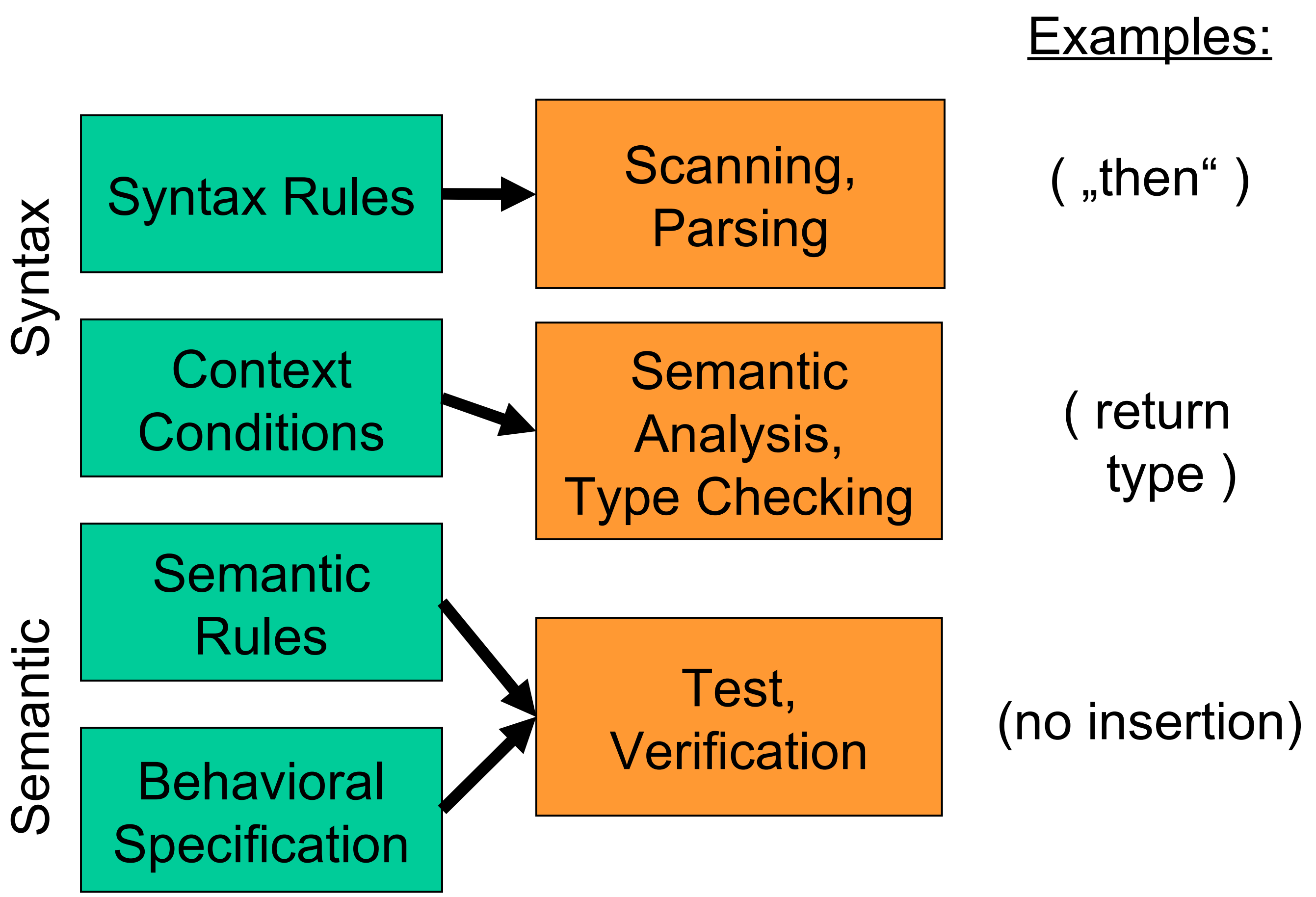
Motivation:

- Basic notions of type systems
- Typing as very light-weight specification
- Understanding subtyping
- Parametric polymorphism
- Alternative typing for OO programming

Correctness levels:

```
class ArraySet implements Set {
  private int[ ] array;
  private int next;
  …

  public void insert( int i ) {
    for ( int j = 0; j < next; j-- )
      if array[ j ] == i then return true;
    return false;
  }
}
```



3.1 Typing of Objects

A type describes properties of values and objects (e.g.: int, Object, LinkedList<A>).

Typing assigns types to program elements with slightly different meanings, e.g.:

- Expression (Ausdruck): Every evaluation of an expression of type T yields a value/object of type T.

- Variable: A variable of type T only holds values of type T.

- Procedures/methods: If the actual parameters have the types of the formal parameters, no type error occurs and the result is of the declared type.

- The type of an object usually captures the information which messages the object understands. This way, typing can avoid „message not understood“-errors.

Remark:

Type systems are probably the most successful formal checking technique.

Explanation: (Typing)

The ***type system*** of a language L describes

- the types supported by L,
- the typing of L-programs (assignment of types to program elements),
- the ***type conditions*** (*Typisierungsbedingungen*) that programs have to fulfill.

A program satisfying the type conditions is called ***type correct*** (*typkorrekt*). Otherwise the program has ***typing errors*** (*Typisierungsfehler*).

A ***type error*** (*Typfehler*) occurs at runtime if

- a value/object of type T is assigned to a variable that has a type incompatible with T;
- an operation is called with parameters of types that are incompatible to the formal parameters.

A language is called ***type safe*** (*typsicher*) if the execution of type correct programs never leads to type errors. Type safety may depend on dynamic checks.

A type system is called ***sound*** (*sicher*) if it guarantees type safety.

Remark:

- Programs with typing errors need not cause type errors.
- In type safe languages, type correctness (static property) implies the absence of type errors (dynamic property).

Explanation: (Type checking)

Type checking consists of

- checking the type conditions at compile time (***static***),
- checking type properties at runtime (***dynamic***).

A language is called ***statically typed*** (*statisch typisiert*) if it does not use dynamic type checks.

Examples: (Dynamic type checks)

Java is not statically typed. It needs dynamic checks:

a) Type casts (Typkonvertierung):

```
Object obj = "Ich neues String-Objekt";
String str = (String) obj;
...
```

b) Array access (Feldzugriff):

```
String[] strfeld = new String[2];
Object[] objfeld = strfeld;
objfeld[0] = new Object();
int strl   = strfeld[0].length();
```

Invariant properties in type safe languages:

- Variables of type T always hold values/objects of type T (or of types compatible with T).
- The evaluation of expressions of type T always yields values/objects of type T (or of a compatible type)

Example: (Typing invariant)

In StoJas, typing guarantees correct field access and existence of methods.

Example: (Type system)

A good first example is the type system of Featherweight Java, FJ (see citation below).

Recommended literature:

Kim B. Bruce: Foundations of Object-Oriented Languages. Types and Semantics.

MIT Press, 2002.

Atsushi Igarashi, Benjamin Pierce, Philip Wadler: Featherweight Java: A Minimal Core Calculus for Java and GJ.

ACM Transactions on Programming Languages and Systems, 23:3, 2001.

T. Nipkow, D. v. Oheimb:
$\text{Java}_{\textit{light}}$ is Type-Safe – Definitely.

Principles of Programming Languages, 1998.

3.2 Subtype Polymorphism

As discussed with Java, subtype polymorphism is obtained by a partial ordering relation on types:

Subtype relation:

Declaration: interface S extends T1, T2, ...
  implicit: S <T1, S <T2, ...

Declaration: class S extends T implements T1, T2, ...
  implicit: S < T, S < T1, S < T2, ...

S < T implies: S[] < T[]

Subtype ordering is the reflexive and transitive closure.

Typing Problem in OO-Languages:

To enable substitutability, a subtype object must have

- all attributes of supertype objects;
- all methods of supertype objects.

Attributes and methods in the subtype must be *compatible* to those of the supertypes. Compatibility concerns typing and accessibility.

Explanation: (Co-/contravariance)

Let S be a subtype of T. Let TV be a type occurrence in T (attribute-, parameter-, result type) and SV the *corresponding* occurrence in S. We say:

- SV and TV are ***covariant*** if SV is a subtype of TV.
- SV and TV are ***contravariant*** if TV is a subtype of SV.

Fact:

OO-languages can only be statically typed iff:

- Attribute types are co- and contravariant, i.e. *invariant*.
- Parameter types are contravariant.
- Result types are covariant.

Example: (Co-/contravariance)

Assuming that overriding of attributes is allowed:

1. Attributes have to be covariant:

```
interface C1 { String a;  ... }
class D1 implements C1  { Object a; ... }
...
C1      cvar = new D1(); // initializes a
String svar = cvar.a;
   // Type error: a not covariant in D1
```

2. Attributes have to be contravariant:

```
interface C2 { Object a;  ... }
class D2 implements C2  { String a; ... }
...
C2     cvar = new D2();
cvar.a = new Object();
   // Type error: a not contravariant in D2
```

3. Parameter types have to be contravariant:

```
interface C3 { int m(Object p);  ... }
class D3 implements C3 {
    int m(String s){ s.length();...}
}
...
C3     cvar = new D3();
cvar.m( new Object() );
   // Type error when executing m:
   // parameter not contravariant
```

4. Result types have to be covariant:

```
interface C4 { String m();  ... }
class D4 implements C4 { Object m(){...} ...}
...
C4     cvar = new D4();
String svar = cvar.m();
   // Type error: result type not covariant
```

Covariance must as well hold for exception types.

Remark:

- Co- and contravariance are important to understand object-oriented typing and the principles underlying behavioral subtyping.
- Java:
  - essentially no contravariance for parameters
  - up to 1.4 no covariance for result types

3.3 A Generic Type System

For many programming scenarios, subtype polymorphism is not the best choice. Often more precise type systems are desirable:

- Types with parameters
- Specialization of used types in subtypes

Disadvantages of subtype polymorphism:

- Specialization is only based on subtype order
- No parameterization w.r.t. the types used in class
- No restriction on parameters

In addition to subtype polymorphism, most modern OO-languages also provide parametric polymorphism: e.g. C++, C#, Java 5, Eiffel, Ada 95.

We consider here Java. The examples are from:

G. Bracha, M. Odersky, D. Stoutamire, P. Wadler: Making the future safe for the past: Adding Genericity to the Java Programming Language. Object-Oriented Programming, Systems, Languages, and Applications, 1998.

M. Naftalin, P. Wadler: Java Generics. O'Reilly

Generic/parametric Types:

Idea:

Allow type variables instead of type names at used occurrences of types in the program, i.e. in classes and interfaces.

Result: parametric class and interface declarations

Type variables have to be declared either

- together with class/interface name or
- together with method signature

Example: (Type variables)

```
class Pair<A,B>  {
    A fst;
    B snd;
    Pair(A f,B s){ ... }
    ...
}
```

`Pair<A,B>` is called a ***parametric/generic type***.

`Pair<_,_>` is called a ***type constructor***.

Type parameters are instantiated at object creation time. Thus, objects always have an unparametric type:

```
Pair<String,Integer> paarvar =
        new Pair <String,Integer> (
                 new String(), new Integer() );
```

A type in Java is represented by a type expression:

- a type identifier (without parameters),
- a type variable,
- a type constructor applied to type expressions.

<u>Examples:</u> (Type expressions)

- String where String is a declared type identifier.
- A where A is a declared variable.
- Pair<String, Object>
- Pair<Pair<A,A>, String>
- List< ? super Number >

Collection types and freely generated data types are the most important application of generic types.

Examples: (Use of generic types)

```
interface Collection<A> {
  public void add (A x);
  public Iterator<A> iterator ();
}


interface Iterator<B> {
  public B next ();
  public boolean hasNext ();
}


class NoSuchElementException
              extends RuntimeException {}


class LinkedList<C>
      implements Collection<C> {
   protected class Node {
      C elt;
      Node next = null;
      Node (C elt) { this.elt = elt; }
   }

   protected Node head = null, tail = null;

   public LinkedList () {}

// class continued on next slide
```

```
    public void add (C elt) {
        if (head == null) {
            head = new Node(elt);
            tail = head;
        } else {
            tail.next = new Node(elt);
            tail = tail.next;
        }
    }

    public Iterator<C> iterator () {

        return new Iterator<C> () {

            protected Node ptr = head;

            public boolean hasNext () {
                return ptr != null;
            }

            public C next () {
                if (ptr != null) {
                    C elt = ptr.elt;
                    ptr = ptr.next;
                    return elt;
                } else {
                    throw new
                      NoSuchElementException ();
                }
            }
        };
    }

}  // end of class LinkedList<C>
```

```
class Test {
  public static void main (String[] a) {
    LinkedList<String> ys =
                  new LinkedList<String>();
    ys.add("zero");
    ys.add("one");
    String y = ys.iterator().next();
  }
}
```

Generic methods:

Similar to type declarations, method declarations can as well be parameterized.

Example:

```
interface SomeCollection<A>
               extends Collection<A> {
     public A some();
}

class Collections {
  public static <A,B> Pair<A,B> somepair(
                          SomeCollection<A> xa,
                          SomeCollection<B> xb ){
    return new Pair<A,B>(xa.some(),xb.some());
  }
}
```

Type parameters of methods are not instantiated at object creation time, but inferred at invocation sites.

New Features of Java 5:

Together with generic types other new features were introduced in Java 5:

- boxing and unboxing of primitive types
- new forms of loops (foreach)
- methods with a variable number of arguments
- enumerated types
- assertions

Example:

```
List<Integer> ints = Arrays.asList(1,2,3);
int s = 0;
for( int n : ints ) { s += n; }
assert s == 6;
```

Remark:

- Although the introduction of type parameters seems to be only a small extension, the resulting type system and its rules become quite complex.

- Generic types and methods are simple examples for generic software components.

Subtype Relation on Generic Types:

Subtype relation is declared or by wildcards:

```
LinkedList<A>  implements  Collection<A>
SomeCollection<A> extends  Collection<A>
Byte implements  Comparable<Byte>
<A   extends  Comparable<A>>
List<Float> is subtype of  List<? extends Number>
```

The subtype relation between type parameters does not carry over to the instantiated types:

LinkedList<String> $\not\leq$ LinkedList<Object>

That is, type parameters are invariant.

Example: (Subtype relation)

The following example shows why equality is required for actual type parameters:

```
class Loophole {
  public static String loophole (Byte y) {
    LinkedList<String> xs =
                  new LinkedList<String>();
    LinkedList<Object> ys = xs;
                  // compilation error
    ys.add(y);
    return xs.iterator().next();
  }
}
```

Unlike with arrays, the types of the actual parameters for an object type are not available at runtime. Thus, a dynamic check is not possible.

Wildcard with “extends“:

```
interface Collection<E> {
    ...
    public boolean addAll(
           Collection<? extends E> c);
    ...
}
```

```
List<Number>  nums = new ArrayList<Number>();
List<Integer> ints = Arrays.asList(1,2);
List<Double>  dbls = Arrays.asList(2.7,3.1);
nums.addAll(ints);
nums.addAll(dbls);
assert nums.toString().equals(”[1,2,2.7,3.1]”);
```

Wildcards may appear in variable declarations:

```
List<? extends Number> nums2 = ints;
nums2.add(3.1); //compile-time error!
assert ints.toString().equals(”[1,2,3.1]”);
```

Wildcard with “super“:

The class Collections contains the following method:

```
public static <T> void copy(
                      List<? super T>  dst,
                      List<? extends T> src ){
   for( T elem: src ) {
      dst.add( elem );
}  }
```

Usage:

```
List<Object>  objs =
           Arrays.<Object>asList(2,3.1,"four");
List<Integer> ints = Arrays.asList(5,6);
Collections.copy(objs, ints);
assert objs.toString().equals("[5,6,four]");
```

The type parameter can as well be given explicitly:

```
Collections.<Object>copy(objs, ints);
Collections.<Number>copy(objs, ints);
Collections.<Integer>copy(objs, ints);
```

Remark:

The Get and Put principle:

Use an extends wildcard when you only get values out of a structure,

use a super wildcard when you only put values into a structure,

don't use a wildcard when you both get and put.

Bounds for Type Parameters:

Java enables to restrict type parameters T by so-called bounds. The bound requires that

T is a subtype of some other type.

Bounds are given at the declaration of the type variable:

<A extends S>

Example: (Type bounds)

```
interface Comparable<A> {
   public int compareTo (A that);
}

class Byte implements Comparable<Byte> {
   private byte value;
   public Byte (byte value) {
     this.value = value;
   }
   public byte byteValue (){ return value; }
   public int compareTo (Byte that) {
     return this.value - that.value;
   }
}

class Collections {
  public static <A extends Comparable<A>>
    A max (Collection<A> xs) {
      Iterator<A> xi = xs.iterator();
      A w = xi.next();
      while (xi.hasNext()) {
        A x = xi.next();
        if (w.compareTo(x) < 0) w = x;
     }
     return w;
} }
```

Implementation of Generic Types in Java:

Parametric types can be translated by raw types. Such an implementation is based on:

- Parameter erasure:
  -
  -
  -
- Bridge methods:
  - If a subclass instantiates a type variable of a superclass, it may be required to add further methods with an appropriate signature.

Example: (Erasure, bridge methods)

By means of parameter erasure and introduction of bridge methods, the above example results in the following:

```java
interface Comparable {
  public int compareTo (Object that);
}

class Byte implements Comparable {
  private byte value;
  public Byte (byte value) {
    this.value = value;
  }
  public byte byteValue () {
    return value;
  }
  public int compareTo (Byte that) {
    return this.value - that.value;
  }
  public int compareTo (Object that) {
    return this.compareTo((Byte)that);
  }
}
```

```
class Collections {
 public static
 Comparable max(Collection xs) {
   Iterator xi = xs.iterator();
   Comparable w = (Comparable)xi.next();
   while (xi.hasNext()) {
     Comparable x = (Comparable)xi.next();
     if (w.compareTo(x) < 0) w = x;
   }
   return w;
 }
}
```

To handle type parameters that are only used in return types, Java 5 supports covariant overriding:

```
class Interval
      implements Iterator<Integer> {
  private int i, n;
  public Interval (int l, int u) {
     i=l; n=u; }
  public boolean hasNext () {
     return (i <= n); }
  public Integer next () {
     return new Integer(i++); }
}
```

Parameter erasure yields the following class where next has a more specific result type than
next in Iterator:

```
class Interval implements Iterator {
  private int i, n;
  public Interval (int l, int u) {
     i=l; n=u; }
  public boolean hasNext () {
     return (i <= n); }
  public Integer next(){
     return new Integer(i++); }
}
```

3.4 Virtual Types

Instead of parameterization of types, specialization can be used. This results in another kind of genericity:

```
class Vector {
  typedef ElemType as Object;
  void addElement( ElemType e ){ ... }
  ElemType elementAt{ int index ){ ... }
}

class PointVector extends Vector {
  typedef ElemType as Point;
}
```

`ElemType` is called a ***virtual*** type. In subclasses, virtual types can be overridden with subtypes.

In the following, we consider an extension of Java with virtual types.

Literature:

Kresten Krab Thorup: Genericity in Java with Virtual Types. European Conference on Object-Oriented Programming, 1997

Aspects of Virtual Types:

A. It is not necessary to distinguish between parametric types and „normal" types:

```
interface IteratorOfObject {
  typedef A as Object;
  public A next();
  public boolean hasNext();
}

interface CollectionOfObject {
  typedef A as Object;
  typedef IteratorOfA as IteratorOfObject;
  public void add(A x);
  public IteratorOfA iterator();
}
```

```
class NoSuchElementException
      extends RuntimeException {}

class LinkedListOfObject
      implements CollectionOfObject {
  // inherits virtual types A, IteratorOfA
  protected class Node {
     A elt;
     Node next = null;
     Node (A elt) { this.elt = elt; }
  }
  protected Node head = null, tail = null;

  public LinkedListOfObject () {}

  public void add (A elt) {
     ... /* as on slide 3.16  */ }

  public IteratorOfA iterator () {
     return new IteratorOfA () {
       ... /* as on slide 3.16  */ };
  }
}
```

```
interface IteratorOfString
          extends IteratorOfObject {
  typedef A as String;
}

class LinkedListOfString
      extends  LinkedListOfObject {
  typedef A as String;
  typedef IteratorOfA as IteratorOfString;
}

class Test {
  public static void main (String[] a) {
    LinkedListOfString ys =
                new LinkedListOfString();
    ys.add("zero");
    ys.add("one");
    String y = ys.iterator().next();
  }
}
```

But each instantiation needs its own type declaration.

B. Subtype relations between generic types can be declared:

`LinkedListOfString` is subtype of `LinkedListOfObject` ;

That is, the following fragment is type correct:

```
LinkedListOfString strl =
            new LinkedListOfString();
LinkedListOfObject objl = strl;
objl.add( new Object() );
// VirtualTypeCastException
```

The covariance induced by virtual types is captured by dynamic type checking.

C. Recursive types and the connection with inheritance can be realised in a better way.

i) Mutual recursion illustrated by the observer pattern:

```
interface Observer {
  typedef SType as Subject;
  typedef EventType as Object;
  void notify (SType subj,EventType e);
}

class Subject {
  typedef OType as Observer;
  typedef EventType as Object;
  OType observers [];
  notifyObservers (EventType e) {
    int len = observers.length;
    for (int i =0;i <len;i++)
      observers [i ].notify(this,e);
  }
}


interface WindowObserver
          extends Observer {
  typedef SType as WindowSubject;
  typedef EventType as WindowEvent;
}

class WindowSubject extends Subject {
  typedef OType as WindowObserver;
  typedef EventType as WindowEvent;
  ...
}
```

ii) Inheritance with recursive occurrences of the declared type:

The declared type K can be designated with „This“ in the core of its declaration. In this case, all these occurrences are interpreted as occurrences of SUBK in a subclass SUBK, and respectively in further subtypes:

```
class SLinkedList {
   This tail;
   public This getTail() { return tail; }
}

class SLinkedListOfObject extends SLinkedList {
  typedef Elem as Object;
  Elem head;
  public Elem getHead() { return head; }
}

class SLinkedListOfString
      extends SLinkedListObject {
  typedef Elem as String;
  // SLinkedListOfString.getTail
  // yields SLinkedListOfString
}
```

Discussion of Virtual Types:

Advantages:

- A new relation between types is not necessary (only ”is subtype of”, not ”is type instance of”)
- Subtype relation between ”parametric” and ”instantiated” types is possible.
- Recursive dependencies can be handled in a more flexible way.

Disadvantages:

- Additional dynamic type checks are necessary (especially because of covariant method parameters):
  - –
  - –
- Recursive instantiation (example Byte, slide 3.24) is not supported.

Remark:

- Thorup’s proposal allows the declaration of virtual types as subtypes of more than one type, and the distinction of a class type for the instantiation of objects.

- There are several approaches proposing an integration of parametric and virtual types, e.g.: K. Bruce, M. Odersky, P. Wadler: A statically safe alternative to virtual types. European Conference on Object-Oriented Programming, 1998.

Implementation of Virtual Types:

In the realization of the presented variant, type safety is achieved by introducing dynamic checking.

We distinguish between *primary* and *overriding* declarations of virtual types.

Essential ideas:

- For the realization of a virtual type T, use the type of the primary declaration of T.
- Define a cast method for each primary declaration of a virtual type T. This cast method is overridden in subtypes by overriding declarations.
- Use the cast method for the checking of actual parameters.
- In order to achieve type correctness in the resulting Java programs, introduce further casts.

<u>Example</u>: (Implementation of virtual types)

```
class Vector {
  typedef ElemType as Object;
  void addElement( ElemType e ) { ... }
  ...
}

class PointVector extends Vector {
  typedef ElemType as Point;
  void addElement( ElemType e ) {
    ...
    e.getX()
    ...
  }
}

...
PointVector pv;
Point pnt;
...
pv.addElement(pnt);
```

is transformed into

```
class Vector {
  Object cast$ElemType(Object o) {
    return o;
  }
  void check$addElement( Object o ) {
    this.addElement(
      this.cast$ElemType(o) );
  }
  void addElement( Object e ) { ... }
  ...
}

class PointVector extends Vector {
  Object cast$ElemType(Object o) {
    try {
      return (Point)o;
    } catch( ClassCastException c ) {
      throw
        new VirtualTypeCastException(...);
    }
  }
```

```
void addElement( Object e$0 ) {
    Point e = (Point)e$0;
        ...
    e.getX()
    ...
  }
}

...
PointVector pv;
Point pnt;
...
pv.check$addElement(pnt);
```