Elements of Type Systems

Random thoughts on types, genericity mechanisms, and more
What is a Type?

Way to think about it: a type is a set of values. It always makes sense to ask whether a value has a certain type (*values do not have a single type!*)

- Usually, expressing a type means being able to define variables of that type
  - variables: identifiers whose value can change at run-time (includes parameters and results of functions)

- Nevertheless, this is not an absolute truth: sometimes one can conceptually refer to types, but not define variables
  - then only constants can assume this type
  - from a type system standpoint, if you can represent constants, you can also represent variables. But there may be implementation issues making the latter undesirable
What are Types Good For?

• Types are used as static properties of values. They enable error checking, optimization

• Examples of types:
  - Simple types: int, bool, float, etc.
  - Composite types: records, classes, functions, unions
  - Parametric types or type templates: arrays

• Type systems vary w.r.t. what types they allow the user to define, what types are pre-defined, etc.

• If pre-defined types are considered constants, does the type system allow type variables? If yes: genericity or polymorphism. Many flavors:
  - parametric polymorphism
  - sybtyping
Subtyping

• Roughly, when a type is a subset of another

• What does that mean for method signatures? (covariance/contravariance of arguments result types)

• Consider (which one really defines a subset?):

```java
interface I1 {
    Animal foo(Dog d);
}
interface I2 extends I1 {
    Dog foo(Animal d);
}
interface I3 extends I1 {
    Object foo(PrettyDog d);
}
interface I4 extends I2 {
    Dog foo(Dog d);
}
```
Universal Types (Parametric Polymorphism) vs Root of Hierarchy (Subtyping)

- Subtype polymorphism allows (homogeneous) generic data types but not type-safely

- When elements are extracted from the data structure, they need to be cast back to their type

- Universal types allow homogeneous collections type safely:

  ```
  List<E> insert<forall E> (List<E> l, E e);
  
  E retrieve<forall E> (List<E>::iterator l);
  ```

- The latter occurs entirely at compile-time, and is commonly more efficient
Parenthesis: Named Conformance vs. Structural Conformance

- In Java, you have to explicitly declare that a class implements an interface (*named conformance*)

- This is an orthogonal (and largely “software engineering”) question to typing

- It is equally reasonable to accept any object that supports the right methods with the right signatures, regardless of whether its definition states that it implements the interface (*structural conformance*)

- Advantage of named conformance: no accidental conformance (*draw* for graphical object, *draw* for cowboy)

- Advantage of structural conformance: no need to adapt code when adding interfaces
Genericity Mechanisms in C++ and Java

A discussion of modern approaches to creating generic data types: parametric typing, virtual typing, etc.
C++ Templates (10 mile-high view)

- Class templates:

```cpp
template <class E1, class E2>
struct Pair {
    E1 fst;
    E2 snd;
    ...
};
```

- Function templates (and C++ type inference):

```cpp
template <class T>
const T& max(const T& e1, const T& e2) {
    if (e1 > e2)
        return e1;
    else
        return e2;
}
```

- C++ type inference allows expressing (polymorphic functions with) universal types
C++ Templates

- C++ templates are type templates, not types. There is no way to use the uninstantiated template.

- Approaches using type templates are called **parametrically polymorphic**.

- Example C++ code skeleton:

```cpp
template <class E>
class List {
    struct ListNode {
        E e;
        ListNode *next;
    };

    public:
    typedef ListNode *iterator;
    void insert(E e) { ... }
    E retrieve(iterator i) { ... }
};
```
Universal Types vs. Type Templates

- Note the difference between
  \texttt{List\langle E\rangle::insert} and the universally
typed (polymorphic) function
  \texttt{List\langle E\rangle insert\langle foreach E\rangle}
  \hspace{1cm} (\texttt{List\langle E\rangle l, E e});
- (this is not a C++ type signature)

- We can express the latter in C++ using
  function templates

  \texttt{template <class E>}
  \texttt{List\langle E\rangle insert(List\langle E\rangle l, E e) \{...\}}
Java Genericity: GJ

- GJ: a parametric polymorphism approach that is the blueprint for Java generics
  - a lot of emphasis on backward and forward compatibility with legacy Java code

```java
interface Collection<E> {
    public void insert(E e);
    public Iterator<E> init();
}

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

class List<A> implements Collection<A> {
    ... 
}
```
F-Bounded Polymorphism

- Type parameters can be bounded with `extends` clauses

- F-bounded polymorphism: the bound can be parameterized, possibly by a type parameter

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

class CollectionUtils {
    public static <A extends Comparable<A>> A max (Collection<A> xs) { ... }
}
GJ Translation

• GJ is translated by *erasure*: regular code with Object references and casts is generated. Type safety is ensured, though

```java
Collection<A> c;
...
c.init().next() ...
```

becomes

```java
Collection c;
...
(A)(c.init().next())...
```

• This limits the possible use of type parameters:
  - cannot inherit from a type parameter (no mixins)
  - cannot cast to a type parameter
  - cannot construct an object
  - originally could not read member classes from a type parameter
Virtual Typing (in the Java context)

• Virtual types: a superclass defines a version of a type variable, but the subclass can refine it (restrict it to a subtype)

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

class PointVector {
    typedef ElemType as Point;
}

• Not statically type safe in the simplest form: a PointVector is not a Vector, because the argument of the insert method is covariant - the implementation is done with casts so runtime type errors may arise
Virtual Types

- Virtual types come from the Beta programming language
- Virtual types are an existential, not a universal types approach
  - generic classes are real classes, not templates
  - code operating on generic classes just relies on the existence of some virtual type
- We saw something like virtual typing in AspectJ and generic aspects
Other Proposals

• Several more proposals for Java genericity, but the GJ model won
  - NexGen
  - PolyJ (\texttt{where} clauses instead of F-bounds)
  - Agesen, Freund, and Mitchell’s parametric types system with a heterogeneous translation (and mixins)