Genericity Mechanisms in C++ and Java

A discussion of modern approaches to creating generic data types: parametric typing, virtual typing, etc.

Yannis Smaragdakis

C++ Templates (10 mile-high view)

- Class templates:

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

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

```
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

Yannis Smaragdakis

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:

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

public:
  typedef ListNode *iterator;
  void insert(E e) { ... }
  E retrieve(iterator i) { ... }
};
```


Yannis Smaragdakis

Universal Types vs. Type Templates

- Note the difference between `List<E>::insert` and the universally typed (polymorphic) function
  `List<E> insert<foreach E>`
  `(List<E> l, E e);`
  - (this is not a C++ type signature)
- We can express the latter in C++ using function templates

```
template <class E>
List<E> insert(List<E> l, E e) {...}
```


Yannis Smaragdakis

Java Genericity: GJ

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

```
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

```
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

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

becomes

```
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)

```
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 has won
  - NexGen
  - PolyJ (`where` clauses instead of F-bounds)
  - Agesen, Freund, and Mitchell's parametric types system with a heterogeneous translation (and mixins)