Parameterized Types for Java

Andrew Myers
Joseph Bank
Barbara Liskov

MIT Lab for Computer Science

Parametric Polymorphism

- Parametric polymorphism: code is generic over a type parameter
- Generic collection classes (e.g., float [ ], set [int], map [string, string])
- Constrained parametric polymorphism → type checking, separate compilation
- Not in widespread use

Why Java?

- OO development languages don't match models of parametric polymorphism.
  - C++, Java
  - name equivalence
  - explicit subtype declarations
  - non–uniform type system
  - overloaded methods
- Java
  - potentially widely used, type–safe
  - only arrays are generic

Generic Code in Java

- Polymorphism through Object
- Vector, Dictionary, Enumeration

```
interface SortedList {
// a mutable list of items in sorted order
    Object first();
    void insert(Object item);
    SortedList rest();
}

class SL_impl implements SortedList {
     Object [ ] items;
     public Object first() { return items[0]; }
     ...
}
```

Java Limitations

```
Object first();
insert(Object item);
```

```
SortedList intlist = ... ;
Object x = ...;
intlist.insert(x);

Integer i = (Integer) intlist.first();
int j = i.intValue();
```

- Verbose
- Loss of compile–time checking
- Expensive run–time type checks

Design Goals

- Backward–compatible
- Sound, expressive type system
- Separate compilation: libraries
- Simple, concise
- Conforming to Java model
- Implementable w/o JVM extensions
- Efficiently implementable (space, time)

Related Work

Where clauses:	CLU, Theta
Subtype bounds:	Eiffel, Trellis/Owl
Template expansion:	C++, Modula−3
F−bounded polymorphism:	Sather, School, Rapide, Pizza
Matching:	Emerald, PolyTOIL

Type Parameters

```
interface SortedList[ T ] ... {
    T first();
    void insert(T item);
    bool member(T item);
    SortedList[ T ] rest();
}
```

formal parameter

actual parameter

```
SortedList[int] s;
int x = s.first();
```

instantiation

Parameter Constraints

- We constrain parameters using *where clauses* — concise, modular

```
interface SortedList[ T ]
  where T { bool lt(T);
            bool equals(T); }
{ T first(); ... }
```

- Restricts types used as T
  - instantiations (users)
  - implementors
- Contract permits separate compilation

Constraining Instantiations

- Actual parameter must satisfy where clause. SortedList[Num]? SortedList[SortedList[Num]]?

```
interface Num {
   bool lt (Num);
   bool equals (Num);
   bool gt(Num); ...
}
```

- Uses signature conformance, *not* explicit subtype.
- Primitive types can satisfy where

Constraining Impls

```
class SL_impl[ T ]
        where T { bool lt(T); bool equals(T);  }
        implements SortedList[ T ]
{
        T[ ] items;
        public bool member(T item) { ...
                while (true) { ... m = (L+R)/2; ...
                        if (! val.lt(items[m]))
                          L = m; else R = m; }
                return (items[m].equals(val)); }
}
```

- Can use *only* where–clause operations
- Separate compilation, no code duplication

Subtype Interactions

```
interface SortedList[ T ]
        where T {  bool lt(T); bool equals(T); }
        extends Bag[ T ]
interface Bag[ T ] where T { bool equals(T); }
```

Means: $\forall$T satisfying SL_impl where clauses

SortedList[T] $<$ Bag[T]

Legal if: All where clauses of extended type (Bag[T]) are satisfied

More Subtype Interactions

```
interface Map[ K, V ] extends Set [ K ]
class IntSet implements Set [ int ]
class Vector [T] implements Cloneable
```

- Allowed:

  SortedList[T] < Bag[T]
- Unsound (argument contravariance):

  $S < T \longrightarrow Bag[\ S\ ] < Bag[\ T\ ]$
- Java arrays: run–time checks

Binding Where–Routines

```
class SL_impl [ T ]
        where T { bool lt(T);
                  bool equals(T); }
interface Num {
        bool equals(Num b);
        bool equals(Object o);
}
```

```
Num x,y;
x.equals(y);
```

- Same rule as Java invocation: tightest match else compile–time error

Summary

- Classes, interfaces can be generic over a type parameter
- Parameters constrained by where clauses, which permit separate compilation
- Primitive types (e.g., int) have methods
- Subtype relation only when actual parameters match: A[T] < B[T], but not A[S] < A[T]
- Binding to overloaded methods resolved identically to Java calls

Efficient Implementation

1. Template translation to Java, ala C++
   – duplicates, re–verifies code
2. Wrapper translation to Java
   – translate code once
   – use wrapper classes to provide where–routines
   – extra indirection, run–time casts
3. Extend Java Virtual Machine
   – space, time efficient
   – still verifiable

Extend Virtual Machine

- Extended JVM format — virtual machine directly supports parametric polymorphism
- Added two bytecodes: invokewhere, invokestaticwhere
- Still verifiable
- Modified Sun interpreter: reasonable performance
- Result: Better target language

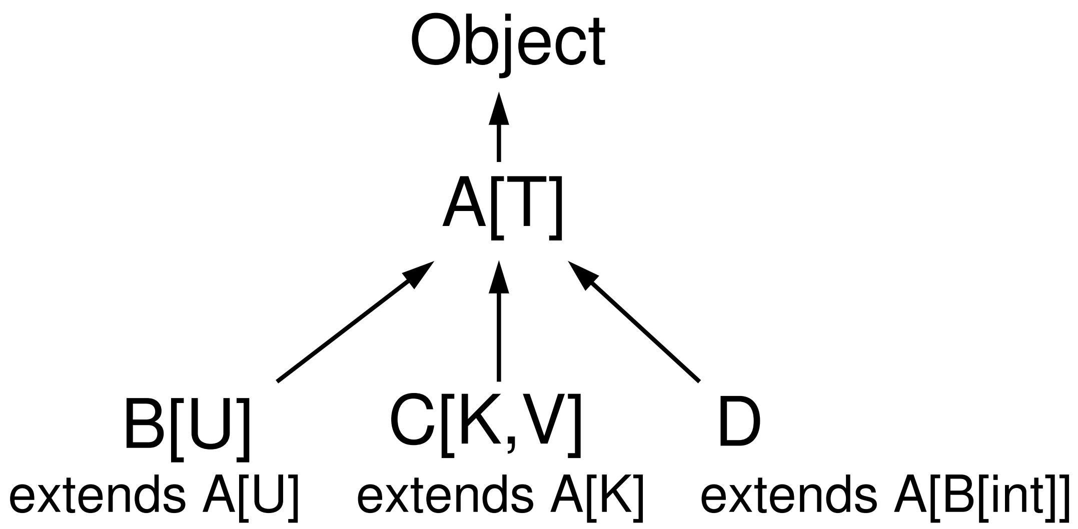
Verifying JVM

- JVM model : untyped stack & registers
- Most operations (e.g., method calls) typed
- Dataflow assigns types for each PC
- Fundamental operation: LUB

x := new A()	*typeof(x) = A*
goto L1	
x := new B()	*typeof(x) = B*
L1: y := x	*typeof(x) = LUB(A,B)*

$$\begin{tikzcd} & C & \\ A \arrow[ur] & & B \arrow[ul] \end{tikzcd}$$

LUB of Instantiations



LUB(B[X], C[X,Y]) = LUB(A[X], A[X]) = A[X]

LUB(B[X], C[Y,X]) = LUB(A[X], A[Y]) = Object

Performance Results

- Sample micro–benchmark tests collection class speed.

		speedup
0. Using Object:	15.1 sec	0%
1. Template translation:	13.4 sec	13%
2. Wrapper translation:	20.2 sec	–25%
3. Extended JVM:	13.4 sec	13%

- Disparity larger with JIT compilation
- Object, wrapper (0, 2) contain runtime type checks: relatively slower than 1, 3

Summary

- Carefully designed backward–compatible extension to Java for param. polymorphism
  - intuitive model, static type checking
  - separate compilation
  - translates to standard Java
- Extended JVM w/ param. polymorphism
  - space and time efficient
  - implemented extended interpreter
  - still verifiable

Conclusions

- Java needs parametric polymorphism
- Constrained parameters essential — allow separate compilation, libraries
- Where clauses are concise, modular, work well with explicit subtype declarations
- No subtyping between instantiations of same type (unlike arrays)
- Verifiable intermediate code can support parametric polymorphism efficiently

Parameterized Types for Java

Andrew Myers
Joseph Bank
Barbara Liskov

MIT Lab for Computer Science

http://www.pmg.lcs.mit.edu/
{andru, jbank, liskov}@lcs.mit.edu