Adding Type Parameterization to the Java Language

Ole Agesen
Sun

Stephen Freund
John Mitchell
Stanford University

Outline

- Generic data structures
  - Type parameters
  - Constraints on object types
- Java type system and run-time architecture
- Language design alternatives
- Implementation by load-time expansion
- Conclusions and directions for future work

Example generic construct: Lists

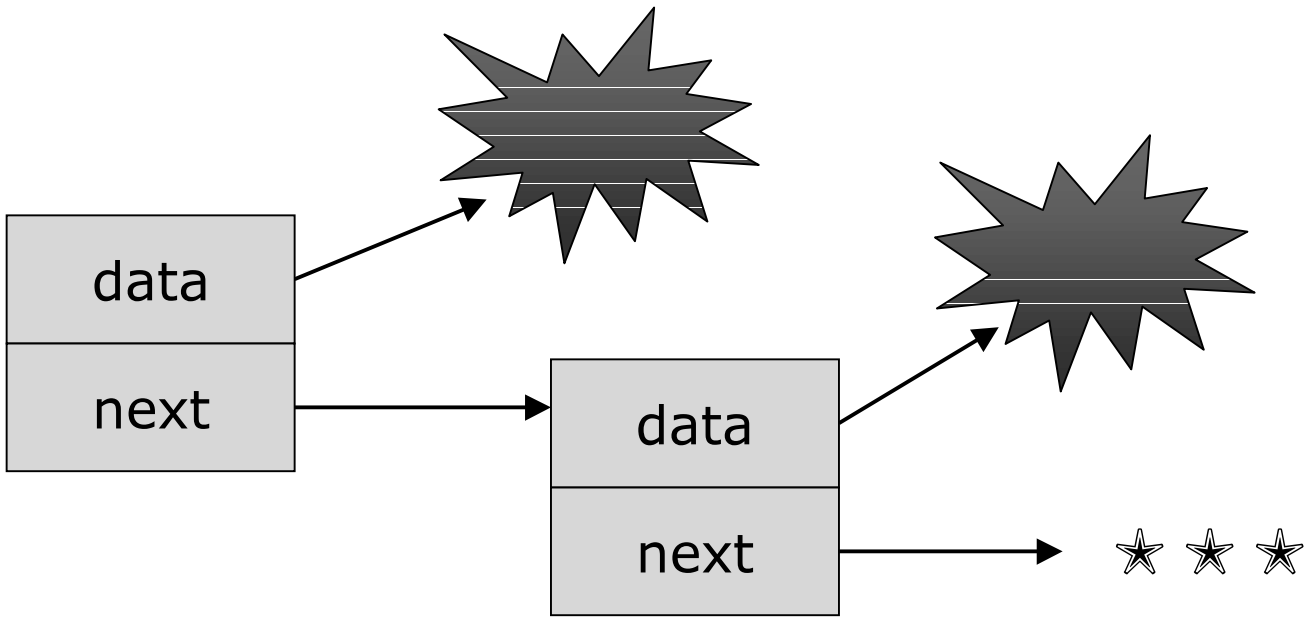
- Lists possible for any type of object
  - For any type t, can have type list_of_t
  - Operations cons, head, tail work for any type
- Define generic list class

```
template <type t> class List {
        private: t* data; List<t> * next;
        public: void      Cons (t* x) { … }
                t*        Head (    ) { … }
                List<t>   Tail  (    ) { … }
};
```

Implementation Issues

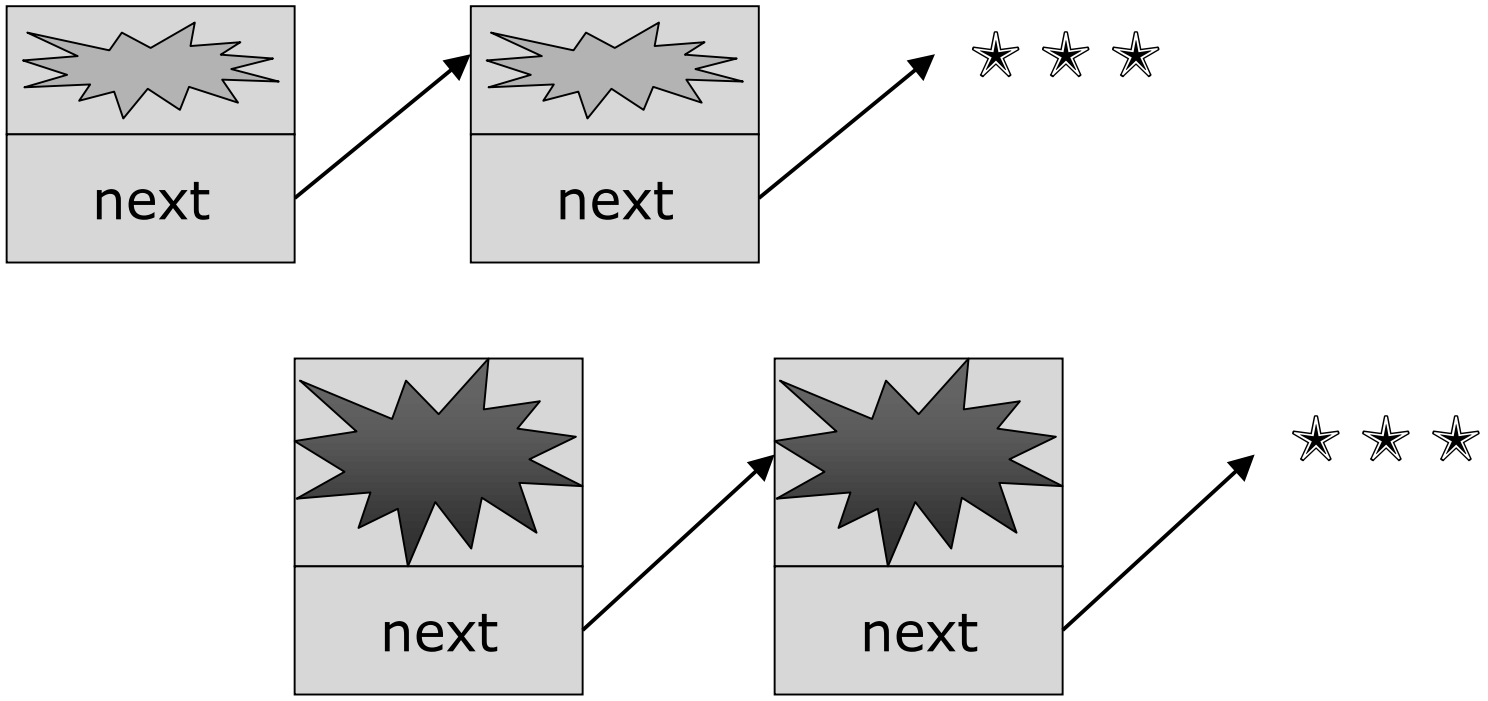
- Data on heap, manipulated by pointer
  - Every list cell has two pointers, `data` and `next`
  - All pointers are same size
  - Can use same representation, code for all types
- Data stored in local variables
  - List cell must have space for data
  - Different representation for different types
  - Different code if offset built into code

“Homogeneous Implementation”



Same representation, code for all types of data

“Heterogeneous Implementation”



Specialize representation, code according to type

Some details that matter

- Allocation of static variables
  - Homogenous: one copy shared by all instances
  - Heterogeneous: separate copy for each instance
- When is template instantiated?
  - Compile- or link-time (C++)
  - Run-time also possible
- Constructor of actual class parameter
- Resolve overloading, …

Constraints on object types

- Need operations on parameter type
- In OO language, these are methods of objects
- Ways to guarantee existence of methods:
  - Expand at compile time and check
  - Compile assuming constraints, check constraints at instantiation time
- This project:

  Separate compilation, with compile-time type checking.

  Programmers write constraints on parameter types.

Example: Hash Table

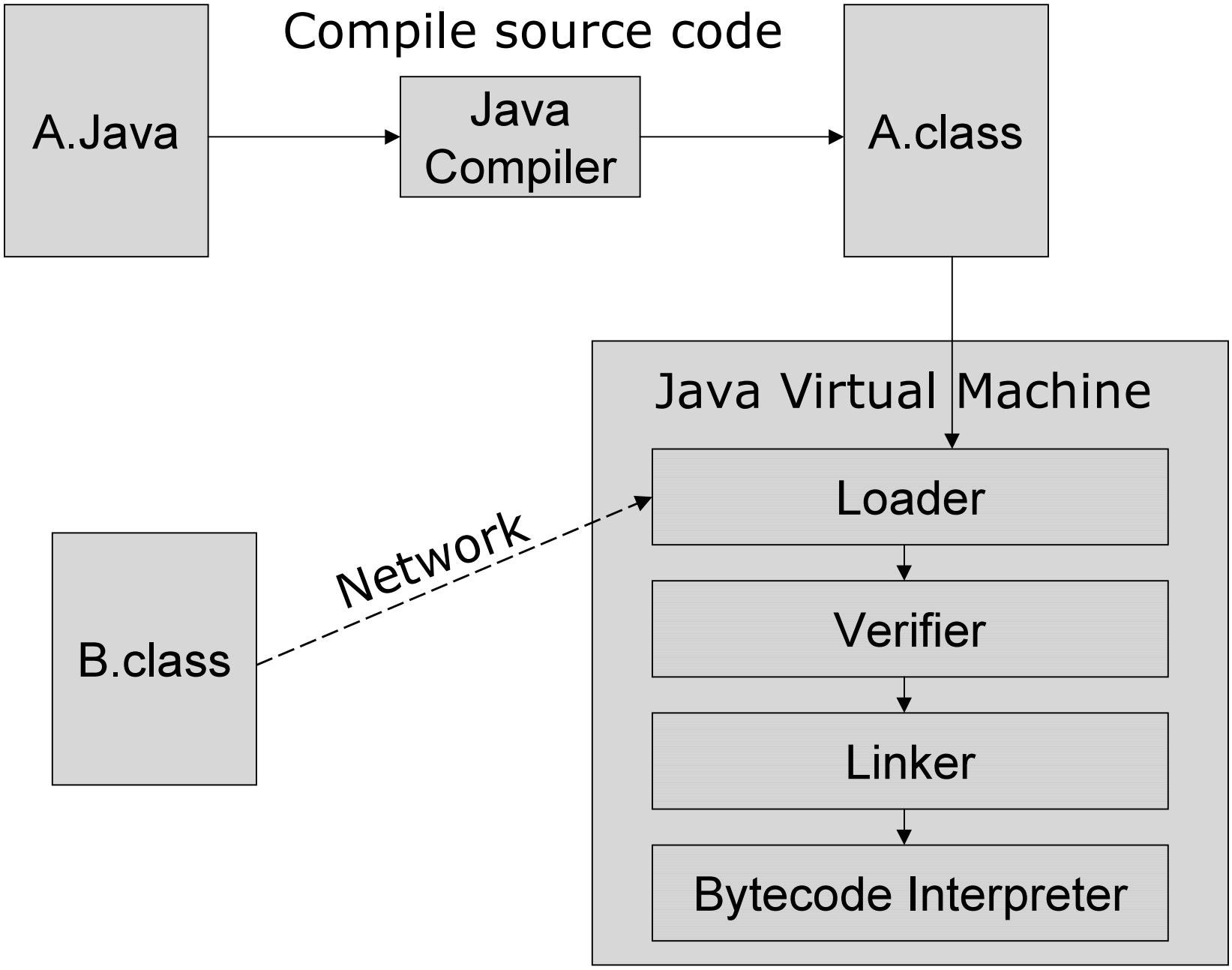
```
interface Hashable {
        int      HashCode ();
};

class HashTable < Key implements Hashable, Value> {
        void    Insert (Key k, Value v) {
                        int bucket = k.HashCode();
                        InsertAt (bucket, k, v);
        }
        …
};
```

Graduated program examples

- List, vector, stack, queue, …
  - no operations needed on “element” objects
- Hash table
  - need `int HashCode()` method to insert object ...
- Priority queue
  - need `bool less_than(t)` method for `t` objects
- “Mixins”
  - Inherit from class passed as actual parameter
  - Need operations on param to define new op’s

Java Overview

- Object System
  - Classes and interfaces
  - Subclass extends superclass, assumed subtype
  - Explicitly declare class implements interface
- Implementation
  - Compiler produces bytecode
  - Virtual machine loads classes on demand, verifies bytecode properties, interprets code


A.Java
Compile source code
Java Compiler
A.class
Java Virtual Machine
Loader
Verifier
Linker
Bytecode Interpreter
B.class
Network

Our Proposal

- Extend Java language with type parameters
  - Generic classes, interfaces
- Use current Java concepts as constraints
  - Class `extends` superclass, `implements` interface
- Implement by expansion at load time
  - Heterogeneous implementation
  - Compile-time checking of generic classes
  - Verify expanded class using standard checks
  - *No change to existing virtual machine*

Competing Proposals

- Pizza [Wadler and Odersky]
  - Similar use of constraints on parameter types (F-bounds)
  - Homogenous implementation in current compiler
- MIT group [Bank, Liskov, Myers]
  - More generous constraints based on CLU `where` clause
  - Shared code with separate constant pool each instance
  - Require additional bytecodes (type depends on constant pool)
- Virtual types [Thorup]
  - Separate subclasses each instance; forced covariance

Comparison

	Language				Implementation			
	Parameterization			Inherit	Shared by instances		Separate each inst.	
	Constraints		None					
	Impl, Extend	Other			Con-stants	Code Only	Com-pile	Load
Pizza	•				•			
MIT		•				•		
C++			•				•	
Thorup				•			•	
AFM	•							•

Implement by load-time expansion

- Compile generic class C<param>
  - Check use of parameter type according to constraints
  - Produce extended form of bytecode class file
    - Store constraints, type parameter names in bytecode file
- Expand when class C<actual> is loaded
  - Replace parameter type by actual class
  - Result is ordinary class file
  - This is a preprocessor to the class loader:
    - No change to the virtual machine
    - No need for additional bytecodes

Example: Hash Table

```
interface Hashable {
        int      HashCode ();
};

class HashTable < Key implements Hashable, Value> {
        void    Insert (Key k, Value v) {
                        int bucket = k.HashCode();
                        InsertAt (bucket, k, v);
        }
        …
};
```

Compilation

- Check use of parameter type
  - Code `Insert (Key k, Value v) { … k.HashCode … }` requires that `k:Key` has `HashCode` method
  - This can be checked against `Key implements Hashable`
- Essentially same static analysis as Pizza
- Generate class file
  - Bytecode for `HashTable`
  - Extra information: constraints, parameter name, ...
  - Standard Java bytecode uses symbolic type names, ...

Generic bytecode with placeholders

```
void Insert (Key k, Value v) {
   int bucket = k.HashCode();
   InsertAt (bucket, k, v);
}
Method void Insert($1, $2)
   aload_1
   invokevirtual #6 <Method $1.HashCode()I>
   istore_3    aload_0   iload_3   aload_1   aload_2
   invokevirtual #7 <Method HashTable<$1,$2>.
                          InsertAt(IL$1;L$2;)V>
   return
```

Load parameterized class file

- Use of `HashTable <Name, Integer>` invokes loader
- Several preprocess steps
  - Locate bytecode for parameterized class, actual types
  - Check the parameter constraints against actual class
  - Substitute actual type name for parameter type
  - Proceed with verifier, linker as usual.
- Implemented with ~500 lines Java code
  - Portable, efficient, no need to change virtual machine
  - We have only implemented loader; Pizza has compiler

Instantiation of generic bytecode

```
void Insert (Key k, Value v) {
    int bucket = k.HashCode();
    InsertAt (bucket, k, v);
}
Method void Insert(Name, Integer)
    aload_1
    invokevirtual #6 <Method Name.HashCode()I>
    istore_3     aload_0    iload_3    aload_1   aload_2
    invokevirtual #7 <Method HashTable<Name,Integer>
                              InsertAt(ILName;LInteger;)V>
    return
```

Comparison with other approaches

- Pizza
  - Similar language, but we avoid limitation of homogeneous implementation (e.g, static variables, constructors)
- MIT
  - More expressive set of constraints (but we could use theirs)
  - We use load-time expansion instead of add'l bytecodes
- C++-style templates
  - Our constraints allow compile-time checking
  - Java load-time expansion possible because of heap objects
  - But we have complications with constructors, overloading, basic types

Comparison

	Language				Implementation			
	Parameterization			Inherit	Shared by instances		Separate each inst.	
	Constraints		None					
	Impl, Extend	Other			Con-stants	Code Only	Com-pile	Load
Pizza	•				•			
MIT		•				•		
C++			•				•	
Thorup				•			•	
AFM	•							•

Conclusions

- Generic classes, interfaces added to Java
- Useful extension of Java language

  container classes, hash table, priority queue, “mixins” ...
- Simple implementation
  - Preprocessor to Java class loader
  - Same technique can be used for other Java extensions
  - No change to verifier: important for security
  - No change to interpreter, bytecode language