Leftover Curry and reheated Pizza: How functional programming nourishes software reuse*

Martin Odersky
University of Southern Australia
www.cis.unisa.edu.au/~cismxo/

Philip Wadler
Bell Labs, Lucent Technologies
www.cs.bell-labs.com/~wadler/

Abstract

Functional programmers and reuse engineers dine at the same table. Delicacies like type abstraction and higher-order functions are meat and potatoes for those who need to reuse code parameterised by types and operations.

The talk will start with a review of modern functional languages. Isolation has given way to systems that interact with C and COM components. Code quality can rival C. Functional programs deliver calls in Brussels, route planes through Paris, and play CDs over networks at Cornell.

The talk will then describe Pizza, an attempt to make functional ideas accessible to a wider community by embedding them in Java. Pizza contains Java as a subset, so its easy to learn, and it compiles to the Java Virtual Machine, so it runs anywhere Java runs, including web browsers. We will focus on how Pizza is designed to add parametric types on top of existing Java libraries, enhancing reuse.

Applications of functional languages have been described elsewhere [Wad98]; this paper describes salient features of the latest version of Pizza.

1 Introduction to Pizza

Say you wish to process collections. Some may be collections of bytes, others collections of strings, and yet others collections of collections of strings. Java supports such variation by allowing you to form a collection of `Object`, so the elements may have any reference type. In order to keep Java simple, you are forced to do some of the work yourself: you must keep track of the fact that you have a collection of bytes, and when you extract an element from the collection you must cast it to class `Byte` before further processing.

This common situation is becoming more common as Java evolves, notably with the addition of collection classes to JDK 1.2. Other languages provide additional support for this situation: in C++, it is supported with templates; in Ada, it is supported with generics; and in ML and Haskell, it is supported with parametric polymorphism. Pizza is an extension to Java that supports types with parameters, derived from the functional programming community's experience with parametric polymorphism.

The Pizza compiler (itself written in Pizza) has been freely available on the web since 1996; and the features and theoretical underpinnings of Pizza have been described elsewhere [OW97]. A new element, described here, is an adjustment in the design so that new Pizza code will work with existing Java libraries, even when the libraries are available only in binary class file form. This paper introduces the most salient features of the new design. Companion papers are in preparation which provide full details, and a new implementation is under construction.

Pizza programs look much like their Java equivalents, except they have more type information and fewer casts. The semantics of Pizza is given by a translation back into ordinary Java. The translation erases type parameters, replaces type variables by their bounding type (typically `Object`), adds casts, and inserts bridge methods so that overriding works properly. The resulting program is pretty much what you would write in ordinary Java if generics weren't available.

In pathological cases, the translation requires bridge methods that cannot be coded in ordinary Java, but can be encoded directly in JVM byte codes. Thus Pizza extends the expressive power of Java, while remaining compatible with the JVM.

Pizza is backward and forward compatible with Java and the Java Virtual Machine (JVM).

- *Compatible with Java.* Every Java program is still legal and retains the same meaning in Pizza.

- *Compatible with JVM.* Pizza compiles into JVM code. No change to the JVM is required. The

*Keynote address, *IEEE Fifth International Conference on Software Reuse*, Vancouver, BC, June 1998.

code is verifiable and can be executed on any Java compliant browser.

- *Compatible with existing libraries.* One can compile a program that uses a parameterised type against existing class file libraries for an unparameterised version of that type. For instance, one can use parameterised collections with the existing collections library.
- *Transparent translation.* There is a straightforward translation from Pizza into Java. The translation leaves field and method names unchanged, and introduces bridge methods according to a simple rule, so it remains easy to use reflection with Pizza.
- *Efficient.* Pizza is translated into Java by erasure: no information about type parameters is carried at run-time. This means Pizza code is pretty much identical to Java code for the same purpose, and equally efficient.
- *Futureproof.* Pizza is designed to extend smoothly to implementations that do carry type information at run-time.

Pizza implements parametric types by erasure. A similar idea has been proposed for Oberon [RS97]. There are a number of other proposals for adding parameterised types to Java, all based on carrying type information at run-time [AFM97, CS97, MBL97]. Run-time types may be less efficient to implement than erasure, and may be harder to interface with legacy code. Pizza is designed to extend smoothly if in future it is judge practicable to carry type information at run-time.

Virtual types have been suggested as an alternative to parametric types for increasing the exprssiveness of types in Java [Tho97, Tor98]. A comparison of the relative strengths of parameteric and virtual types appears elsewhere [BOW98]. It may be possible to merge virtual and parametric types [BOW98, TT98], but it is not clear whether the benefits of the merger outweigh the increase in complexity.

This paper is structured as follows. Sections 2–4 introduce the basic features of Pizza, using a running example based on collections and linked lists. Section 5 describes how bridge methods that cannot be expressed in Java, but can be expressed by direct translation into the JVM.

2 Generic classes

The first example demonstrates the fundamentals of generic classes. For this example and the following ones, we first consider Java code for a task, then see how it is rewritten in Pizza.

Figure 1 shows interfaces for collections and iterators, and a linked list class (all much simplified from the collections library) in Java.

The collection interface provides a method to add an element to a collection, and a method to return an iterator for the collection. In turn, the iterator interface provides a method to determine if the iteration is done, and (if it is not) a method to return the next element and advance the iterator. The linked list class implements the collections interface, and contains a nested class for list nodes and an anonymous class for the list iterator. Each element has type `Object`, so one may form linked lists with elements of any reference type, including `Byte`, `String`, or `LinkedList` itself.

The test class utilizes linked lists in Java. The user must recall what type of element is stored in a list, and cast to the appropriate type when extracting an element from a list. Extracting an element from a list of lists requires two casts. If the user accidentally attempts to extract a byte from a list of strings, this raises an exception at run-time.

Now, Figure 2 shows collections, iterators, and linked lists rewritten in Pizza.

The interfaces and class take a type parameter `A`, written in angle brackets, representing the element type. Each place where `Object` appeared in the previous code, it is now replaced by `A`. Each place where `Collection`, `Iterator`, or `LinkedList` appeared in the previous code, it is now replaced by `Collection<A>`, `Iterator<A>`, or `LinkedList<A>` (the one exception being the declaration of the class constructor). The nested class `Node` has `A` as an implicit parameter inherited from the scope, the full name of the class being `LinkedList<A>.Node`.

In the test class, instead of relying on the user's memory, parameters document the type of each list's elements, and no casts are required. The code to extract an element from a list of lists is more perspicuous. Now an attempt to extract a byte from a list of strings indicates an error at compile-time.

To translate from Pizza back to Java, one replaces each type by its *erasure*. The erasure of a parametric type is obtained by deleting the parameter (so `LinkedList<A>` erases to `LinkedList`), the erasure of a non-parametric type is the type itself (so `Byte` erases to `Byte`) and the erasure of a type parameter is `Object` (so `A` erases to `Object`). A suitable cast is inserted around each method call where the return type is a type parameter. Translating the Pizza code for lists yields the Java code we started with (except

```
interface Collection {
  public void add (Object x);
  public Iterator iterator ();
}
interface Iterator {
  public Object next ();
  public boolean hasNext ();
}
class NoSuchElementException extends RuntimeException {}
class LinkedList implements Collection {
  protected class Node {
    protected Object elt;
    protected Node next = null;
    protected Node (Object elt) { this.elt=elt; }
  }
  protected Node head = null, tail = null;
  public LinkedList () {}
  public void add (Object elt) {
    if (head==null) { head=new Node(elt); tail=head; }
    else { tail.next=new Node(elt); tail=tail.next; }
  }
  public Iterator iterator () {
    return new Iterator () {
      protected Node ptr=head;
      public boolean hasNext () { return ptr!=null; }
      public Object next () {
        if (ptr!=null) { Object elt=ptr.elt; ptr=ptr.next; return elt; }
        else throw new NoSuchElementException ();
      }
    };
  }
}
class Test {
  public static void main (String[] args) {
    LinkedList xs = new LinkedList();        // byte list
    xs.add(new Byte((byte)0)); xs.add(new Byte((byte)1));
    Byte x = (Byte)xs.iterator().next();
    LinkedList ys = new LinkedList();        // string list
    ys.add("zero"); ys.add("one");
    String y = (String)ys.iterator().next();
    LinkedList zss = new LinkedList();       // string list list
    zss.add(ys);
    String z = (String)((LinkedList)zss.iterator().next()).iterator().next();
    Byte w = (Byte)ys.iterator().next();     // string list as byte list: run-time exception
  }
}
```

Figure 1: Collections in Java

for the line that was in error). Thus, a new program compiled against the Pizza code could be used with an old library compiled against the Java code.

The scope of a type parameter is the entire class, excluding static members and static initialisers. This is required since different instances of a class may have different type parameters, but access the same static members. Parameters are irrelevant when using a class name to access a static member, and must be omitted.

```
interface Collection<A> {
  public void add(A x);
  public Iterator<A> iterator();
}
interface Iterator<A> {
  public A next();
  public boolean hasNext();
}
class NoSuchElementException extends RuntimeException {}
class LinkedList<A> implements Collection<A> {
  protected class Node {
    protected A elt;
    protected Node next = null;
    protected Node (A elt) { this.elt=elt; }
  }
  protected Node head = null, tail = null;
  public LinkedList () {}
  public void add (A elt) {
    if (head==null) { head=new Node(elt); tail=head; }
    else { tail.next=new Node(elt); tail=tail.next; }
  }
  public Iterator<A> iterator () {
    return new Iterator<A> () {
      protected Node ptr=head;
      public boolean hasNext () { return ptr!=null; }
      public A next () {
        if (ptr!=null) { A elt=ptr.elt; ptr=ptr.next; return elt; }
        else throw new NoSuchElementException ();
      }
    };
  }
}
class Test {
  public static void main (String[] args) {
    LinkedList<Byte> xs = new LinkedList<Byte>();      // byte list
    xs.add(new Byte(0)); xs.add(new Byte(1));
    Byte x = xs.iterator().next();
    LinkedList<String> ys = new LinkedList<String>();  // string list
    ys.add("zero"); ys.add("one");
    String y = ys.iterator().next();
    LinkedList<LinkedList<String>> zss                 // string list list
      = new LinkedList<LinkedList<String>>();
    zss.add(ys);
    String z = zss.iterator().next().iterator().next();
    Byte w = ys.iterator().next();                     // string list as byte list: compile-time error
  }
}
```

Figure 2: Collections in Pizza

Angle brackets were chosen for type parameters since they are familiar to C++ users, and since they avoid confusion that may otherwise arise. Each of the other form of brackets may lead to confusion. If round brackets are used, it is difficult to distinguish type and value parameters. If square brackets are used, it is

```
interface Comparator {
  public int compare (Object x, Object y);
}
class ByteComparator implements Comparator {
  public int compare (Object x, Object y) {
    return ((Byte)x).byteValue() - ((Byte)y).byteValue();
  }
}
class Collections {
  public static Object max (Collection xs, Comparator c) {
    Iterator xi = xs.iterator();
    Object w = xi.next();
    while (xi.hasNext()) {
      Object x = xi.next();
      if (c.compare(w,x) < 0) w = x;
    }
    return w;
  }
}
class Test {
  public static void main (String[] args) {
    LinkedList xs = new LinkedList();                           // byte list with byte comparator
    xs.add(new Byte(0)); xs.add(new Byte(1));
    Byte x = (Byte)Collections.max(xs, new ByteComparator());
    LinkedList ys = new LinkedList();                           // string list with byte comparator
    ys.add("zero"); ys.add("one");
    String y = (String)Collections.max(ys, new ByteComparator());  // run-time exception
  }
}
```

Figure 3: Comparators in Java

difficult to distinguish type parameters and array dimensions. If curly brackets are used, it is difficult to distinguish type parameters from class bodies.

Phrases like `LinkedList<LinkedList<String>>` pose a problem to the parser, since `>>` is treated as a single lexeme. (Similarly for `>>>`.) In C++, users are required to add extra spaces to avoid this problem. In Pizza, there is no worry for the user, the problem is instead solved by a slight complication to the grammar.

3 Parametric methods and bridges

The second example demonstrates a parametric method and illustrates the need for bridges.

Figure 3 shows an interface for comparators, a comparator for bytes, and a class with a static method to find the maximum of a collection (all based on the collections library) in Java.

The comparator interface specifies a method that takes a pair of objects, and returns an integer that is negative, zero, or positive if the first object is less than, equal to, or greater than the second. In the byte comparator class, the two objects are cast to bytes, and then subtracted to generate a comparison value. The last class defines a static method that accepts a collection and a comparator and returns the maximum element in the collection (if it is non-empty).

The test class utilises comparators in Java. As before, the user must keep track of the result type and cast the results as appropriate. Using a string collection with a byte comparator raises an exception at run-time.

Now, Figure 4 shows comparators and maximum rewritten in Pizza.

The interface now has a type parameter `A`, and the compare method takes two arguments of type `A`. The byte comparator class implements the comparator interface with the type parameter instantiated to `Byte`, and so the compare method takes two arguments of

```
interface Comparator<A> {
  public int compare (A x, A y);
}
class ByteComparator implements Comparator<Byte> {
  public int compare (Byte x, Byte y) {
    return x.byteValue() - y.byteValue();
  }
}
class Collections {
  public static <A> A max (Collection<A> xs, Comparator<A> c) {
    Iterator<A> xi = xs.iterator();
    A w = xi.next();
    while (xi.hasNext()) {
      A x = xi.next();
      if (c.compare(w,x) < 0) w = x;
    }
    return w;
  }
}
class Test {
  public static void main (String[] args) {
    LinkedList<Byte> xs = new LinkedList<Byte>();          // byte list with byte comparator
    xs.add(new Byte(0)); xs.add(new Byte(1));
    Byte x = Collections.max(xs, new ByteComparator());
    LinkedList<String> ys = new LinkedList<String>();      // string list with byte comparator
    ys.add("zero"); ys.add("one");
    String y = Collections.max(ys, new ByteComparator());  // compile-time error
  }
}
```

Figure 4: Comparators in Pizza

```
class ByteComparator implements Comparator {
  public int compare (Byte x, Byte y) {
    return x.byteValue() - y.byteValue();
  }
  // bridge
  public int compare (Object x, Object y) {
    return this.compare((Byte)x, (Byte)y);
  }
}
```

Figure 5: Comparator translated from Pizza to Java

type `Byte`, and no casts are required. In the last class, the static method `max` takes a type parameter, indicated by writing `<A>` before the method signature; the type parameter `A` appears in the result type, argument types, and method body.

In the test class, as before, no casts are required. Now using a string collection with a byte comparator indicates an error at compile-time.

Type parameters are inferred for a polymorphic method as follows: the smallest type parameter that yields a valid call is chosen. In the example above, the call to `max` with a byte collection and a byte comparator infers that the type parameter `A` is `Byte`. If there is no unique smallest type, inference fails, as in the call to `max` with a string collection and a byte comparator. (As described here, inference will not work for `null`, empty lists, and other empty data structures; we are contemplating an extension that will work in these cases.)

For purposes of type comparison, subtyping is invariant for parameterised types: for instance, even though the class `Byte` is a subtype of `Object`, the parameterised type `Collection<Byte>` is not a subtype of `Collection<Object>`. In comparison, arrays use

```
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;
  }
}
class Test {
  public static void main (String[] args) {
    LinkedList xs = new LinkedList();                      // byte collection
    xs.add(new Byte(0)); xs.add(new Byte(1));
    Byte x = (Byte)Collections.max(xs);
    LinkedList ys = new LinkedList();                      // boolean collection
    ys.add(new Boolean(false)); ys.add(new Boolean(true));
    Boolean y = (Boolean)Collections.max(ys);              // run-time exception
  }
}
```

Figure 6: Comparable objects in Java

covariant subtyping, so the array type `Byte[]` is a subtype of `Object[]`.

To translate from Pizza to Java one erases types and adds casts as before. The `Comparator` interface and `Collections` class in Pizza translate back to the Java versions above. The `ByteComparator` requires a little more cleverness, as shown in Figure 5. A *bridge* method must be introduced, since overriding occurs only when types match exactly. Overloading allows the bridge and the original method to share the same name. In general, a bridge is introduced whenever a class implements a parameterised interface or extends a parameterised class at an instantiated type. A problematic case arises if the overridden method has a parameterised type but no arguments, as discussed in Section 5.

A class can implement an interface in only one way. Thus, a class cannot implement the parameterised interfaces `I<S>` and `I<T>` unless `S` is identical to `T`. Similarly for the superinterfaces of an interface.

4 Bounds

The third example demonstrates that type parameters may be bounded.

As an alternative to comparators, one may specify that the object itself contains a suitable comparison method. Figure 6 shows an interface for comparable objects, the standard byte class that implements this

```
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 implements 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;
  }
}
class Test {
  public static void main (String[] args) {
    LinkedList<Byte> xs = new LinkedList<Byte>();           // byte collection
    xs.add(new Byte(0)); xs.add(new Byte(1));
    Byte x = Collections.max(xs);
    LinkedList<Boolean> ys = new LinkedList<Boolean>();     // boolean collection
    ys.add(new Boolean(false)); ys.add(new Boolean(true));
    Boolean y = Collections.max(ys);                        // compile-time error
  }
}
```

Figure 7: Comparable objects in Pizza

interface, and a variant method to find the maximum of a collection (all based on the collections library), in Java.

Whereas a comparator defines a method `compare` that takes two objects, a comparable class defines a method `compareTo` that takes one object, which is compared with the method's receiver. The standard `Byte` class implements the `Comparable` interface (as of JDK 1.2). It defines two version of `compareTo`, one which expects a `Byte`, and one which expects an `Object`; the second of these overrides the method in the interface. The last class defines a static method that returns the maximum element in the collection (if it is non-empty and all the elements in it are comparable.

The test class utilises comparable objects in Java. As before, the user must keep track of the result type and cast the results as appropriate. Booleans do not implement the comparable interface, so an attempt to find the maximum of a collection of booleans raises an exception at run-time.

Now, Figure 7 shows comparable objects rewritten in Pizza.

In `Byte`, the `compareTo` method now only needs to be defined at type `Byte`, the definition for `Object` being automatically derived by the insertion of a bridge method. In `max` the type parameter is constrained by a bound, `A implements Comparable<A>`, indicating that the type variable should only be instantiated by a class that is comparable with itself. It is legal to apply `max` at type `Byte` because `Byte implements Comparable<Byte>`. In the body of `max` it is permitted to call the `compareTo` method since the receiver and argument are both of type `A`, and `A` implements

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

Figure 8: Intervals in Pizza

```
interface Iterator {
  public boolean hasNext ();
  public Object next ();
}
class Interval implements Iterator {
  private int i;
  private int n;
  public Interval (int l, int u) { i = l; n = u; }
  public boolean hasNext () { return (i <= n); }
  public Integer next/*1*/ () { return new Integer(i++); }
  public Object next/*2*/ () { return next/*1*/(); }  // bridge
}
```

Figure 9: Intervals translated from Pizza to Java

`Comparable<A>`.

In the test class, now an attempt to find the maximum of a collection of booleans indicates an error at compile-time.

In general, a bound is introduced by following the parameter with `extends` and a class, or `implements` and an interface. The bounding class or interface may itself be parameterised, and may include type variables appearing elsewhere in the parameter section; recursion or mutual recursion between parameters is allowed. Omitting a bound is equivalent to using the bound `Object`.

The definition of erasure is extended so that the erasure of a type variable is the erasure of its bound (so `A` erases to `Comparable` in `max`). The Pizza versions of `Comparable`, `Byte`, and `Collections` translate back to the Java versions given above.

5 A bridge too far

A problematic case of bridging may arise if the overridden method has a parameterised type but no arguments.

Figure 8 shows the iterator interface in Pizza (adopted from the collections library), and a class that implements that interface. Here the `next` method of the interval class returns an `Integer`, to match the instantiation of the type parameter.

Translating from Pizza to Java yields the code in Figure 9. As one would expect, a bridge must be added to the `Interval` class. Unfortunately, the result is not legal Java, as the two versions of `next` cannot be distinguished because they have identical arguments. The code above distinguishes our intention by suffixing the declarations and calls with `/*1*/` and `/*2*/` as appropriate.

Fortunately, the two versions of `next` can be distinguished in the JVM, which identifies methods using a signature that includes the result type. (Tests confirm this technique works on both Javasoft and Microsoft implementations of the JVM.) This situation represents the one place where Pizza cannot translate into Java, but must be defined by translation directly into the JVM.

This loophole points out an underspecification in Java and the JVM. Since legal JVM code may in fact contain more than one implementation of a method with the same name and the same argument types, it needs to be specified which of these is returned by `Class.getMethod`. (Tests show the Javasoft and Microsoft implementations differ.)

6 Conclusions

Java already incorporates a number of ideas familiar to the functional programming (FP) community. Java, like FP, uses garbage collection collection to free the user from details of storage allocation. Java, like FP, makes strings an immutable type, and encourages (via `final`) immutable fields and parameters. Java's inner classes closely resemble FP's higher-order functions, and Java's anonymous classes closely resemble FP's lambda expressions. In some ways, Java is closer to FP than to C: for instance, C programmers expect to distinguish between passing a record or passing a pointer to a record, a distinction not possible in Java or in FP.

Pizza forwards this trend by adding to Java another feature of functional languages, parametric polymorphism. On the surface, parametric polymorphism resembles type templates in C++, and indeed we have chosen a syntax that reflects this. However, at a deeper level parametric polymorphism differs profoundly from C++ templates. Templates are essentially syntactic in nature, corresponding to macros; parametric polymorphism has deep foundations in logic and semantics [Rey83, Wad89]. Templates are implemented by expansion, with special steps required to avoid code bloat; parametric types are implemented by erasure, with no possibility of code bloat. Templates leave constraints on type variables implicit in the context; parametric types make constraints on type variables explicit via bounds. In C++, templates are type checked at link-time; in Pizza, parametric types are checked at compile-time.

We believe that functional programming has much to offer to conventional programmers, and that Java provides a good foundation on which to build a bridge between these communities.

Pizza is available from the following sites.

```
http://www.cs.bell-labs.com/~wadler/pizza/
http://www.cis.unisa.edu.au/~pizza
http://wwwipd.ira.uka.de/~pizza/
```

References

[AFM97] Ole Agesen, Stephen Freund, and John C. Mitchell. Adding parameterized types to Java. *Object-Oriented Programming: Systems, Languages, and Applications*, ACM, 1997.

[BOW98] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. *Foundations of Object-Oriented Languages*, January 1998.

[CS97] Corky Cartwright and Guy Steele. Yet another parametric types proposal. Message to Java genericity mailing list, August, 1997.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. *Principles of Programming Languages*, pp. 132–145, ACM, January 1997.

[Rey83] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing*, pp. 513–523, North-Holland, Amsterdam, 1983.

[RS97] P. Roe and C. A. Szyperski. Lightweight Parametric Polymorphism for Oberon. *Joint Modular Languages Conference*, Hagenberg Austria, March 1997

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. *of Programming Languages*, pp. 146–159, ACM, January 1997.

[Tho97] Kresten Krab Thorup. Genericity in Java with virtual types. In *European Conference on Object-Oriented Programming*, pp. 444–471, LNCS 1241, Springer-Verlag, 1997.

[Tor98] Mads Togersen. Virtual types are statically safe. *Foundations of Object-Oriented Languages*, January 1998.

[TT98] Kresten Krab Thorup and Mads Togersen. Structural virtual types. Informal session on types for Java, *Foundations of Object-Oriented Languages*, January 1998.

[Wad89] Philip Wadler. Theorems for free! *Functional Programming and Computer Architecture*, London, ACM, September 1989.

[Wad98] Philip Wadler. An angry half dozen. *SIGPLAN Notices* 33(2):25–30, February 1998. [NB. Table of contents on cover of this issue is wrong.]