

Pizza into Java: Translating theory into practice

Martin Odersky
University of Karlsruhe

Philip Wadler
University of Glasgow

Abstract

Pizza is a strict superset of Java that incorporates three ideas from the academic community: parametric polymorphism, higher-order functions, and algebraic data types. Pizza is defined by translation into Java and compiles into the Java Virtual Machine, requirements which strongly constrain the design space. Nonetheless, Pizza fits smoothly to Java, with only a few rough edges.

1 Introduction

There is nothing new beneath the sun.
— Ecclesiastes 1:10

Java embodies several great ideas, including:

- strong static typing,
- heap allocation with garbage collection, and
- safe execution that never corrupts the store.

These eliminate some sources of programming errors and enhance the portability of software across a network.

These great ideas are nothing new, as the designers of Java will be the first to tell you. Algol had strong typing, Lisp had heap allocation with garbage collection, both had safe execution, and Simula combined all three with object-oriented programming; and all this was well over a quarter of a century ago. Yet Java represents the first widespread industrial adoption of these notions. Earlier attempts exist, such as Modula-3, but never reached widespread acceptance.

Clearly, academic innovations in programming languages face barriers that hinder penetration into industrial practice. We are not short on innovations, but we need more ways to translate innovations into practice.

Pizza is a strict superset of Java that incorporates three other ideas from the academic community:

Revised from a paper presented in Proc. 24th ACM Symposium on Principles of Programming Languages, Paris, France, January 1997.

© 1997 ACM. Permission to copy without fee all or part of this material is granted, provided that the copies are not made for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

- parametric polymorphism,
- higher-order functions, and
- algebraic data types.

Pizza attempts to make these ideas accessible by translating them into Java. We mean that both figuratively and literally, because Pizza is defined by translation into Java. Our requirement that Pizza translate into Java strongly constrained the design space. Despite this, it turns out that our new features integrate well: Pizza fits smoothly to Java, with relatively few rough edges.

Promoting innovation by extending a popular existing language, and defining the new language features by translation into the old, are also not new ideas. They have proved spectacularly successful in the case of C++.

Pizza and Java. Our initial goal was that Pizza should compile into the Java Virtual Machine, or JVM. We considered this essential because the JVM will be available across a wide variety of platforms, including web browsers and special-purpose chips. In addition, we required that existing code compiled from Java should smoothly inter-operate with new code compiled from Pizza. Among other things, this would give Pizza programmers access to the extensive Java libraries that exist for graphics and networking.

We did not originally insist that Pizza should translate into Java, or that Pizza should be a superset of Java. However, it soon became apparent that the JVM and Java were tightly coupled, and any language that compiles into the JVM would lose little in efficiency — and gain much in clarity — by translating into Java as an intermediate stage. A corollary was that making Pizza a superset of Java imposed few additional constraints on the design. The requirement of smooth inter-operability amounts to insisting that the translation of Pizza into Java is the identity for the Java subset of Pizza.

Heterogenous and homogenous translations. We like translations so much that we have two of them: a *heterogenous* translation that produces a specialised copy of code for each type at which it is used, and a *homogenous* translation that uses a single copy of the code with a universal representation. Typically the heterogenous translation yields code that runs faster, while the homogenous translation yields code that is more compact.

The two translations correspond to two common idioms for writing programs that are generic over a range of types. The translations we give are surprisingly natural, in that they are close to the programs one would write by hand

in these idioms. Since the translations are natural, they help the programmer to develop good intuitions about the operational behaviour of Pizza programs.

Mixtures of heterogenous and homogenous translation are possible, so programmers can trade size for speed by adjusting their compilers rather than by rewriting their programs. We expect the best balance between performance and code size will typically be achieved by using the heterogenous translation for base types and the homogenous translation for reference types.

Related work. Pizza's type system is based on a mixture of Hindley-Milner type inference [DM82] and F-bounded polymorphism [CCH⁺89], closely related to type classes [WB89, Jon93]. There is also some use of existential types [CW85, MP88].

Superficially, Pizza types appear similar to the template mechanism of C++ [Str91]. Both allow parameterized types, both have polymorphic functions with implicit instantiation, and both have similar syntax. However, the similarity does not run deep. C++ templates are implemented by macro expansion, such that all type checking is performed only on the function instances, not on the template itself. In the presence of separate compilation, type checking in C++ must be delayed until link time, when all instance types are known. In contrast, Pizza types allow full type checking at compile time.

Bank, Liskov, and Myers [BLM96] describe a polymorphic type system for Java, broadly similar to ours. The key difference is that we translate our language into the existing JVM, while they extend the JVM to support polymorphism. We believe these two approaches have complementary advantages, and are both worthy of pursuit.

Ideas for translating higher-order functions into classes belong to the folklore of the object-oriented community. A codification similar to ours has been described by Laufer [Läu95]. A facility similar to higher-order functions is provided by the anonymous classes proposed for the next version of Java [Sun96a]. Our observations about visibility problems appear to be new.

Status. We have a complete design for Pizza, including type rules, as sketched in this paper. We consider the design preliminary, and subject to change as we gain experience.

We have implemented EspressoGrinder, a compiler for Java written in Java [OP95]. Interestingly, it was a commercial client who approached us to add higher-order functions to EspressoGrinder. We added this feature first, to the client's satisfaction, and are now at work adding the other features of Pizza.

Structure of this report. To read this paper, you will need a passing acquaintance with parametric polymorphism, higher-order functions, and algebraic types (see, e.g. [BW88, Pau91, CW85]); and a passing acquaintance with Java (see, e.g. [AG96, GJS96]).

This paper is organised as follows. Section 2 introduces parametric polymorphism, Section 3 introduces higher-order functions, and Section 4 introduces algebraic data types. Each Pizza feature is accompanied by a description of its translation into Java. Section 5 explores further issues connected to the type system. Section 6 describes rough edges we encountered in fitting Pizza to Java. Section 7 concludes. Appendix A summarises Pizza syntax. Appendix B discusses formal type rules.

Example 2.1 Polymorphism in Pizza

```
class Pair<elem> {
    elem x; elem y;
    Pair (elem x, elem y) {this.x = x; this.y = y;}
    void swap () {elem t = x; x = y; y = t;}
}

Pair<String> p = new Pair("world!", "Hello,");
p.swap();
System.out.println(p.x + p.y);

Pair<int> q = new Pair(22, 64);
q.swap();
System.out.println(q.x - q.y);
```

Example 2.2 Heterogenous translation of polymorphism into Java

```
class Pair_String {
    String x; String y;
    Pair_String (String x, String y) {this.x = x; this.y = y;}
    void swap () {String t = x; x = y; y = t;}
}

class Pair_int {
    int x; int y;
    Pair_int (int x, int y) {this.x = x; this.y = y;}
    void swap () {int t = x; x = y; y = t;}
}

Pair_String p = new Pair_String("world!", "Hello,");
p.swap(); System.out.println(p.x + p.y);

Pair_int q = new Pair_int(22, 64);
q.swap(); System.out.println(q.x - q.y);
```

2 Parametric polymorphism

A set of integers is much the same as a set of characters, sorting strings is much the same as sorting floats. *Polymorphism* provides a general approach to describing data or algorithms where the structure is independent of the type of element manipulated.

As a trivial example, we consider an algorithm to swap a pair of elements, both of the same type. Pizza code for this task appears in Example 2.1.

The class `Pair` takes a type parameter `elem`. A pair has two fields `x` and `y`, both of type `elem`. The constructor `Pair` takes two elements and initialises the fields. The method `swap` interchanges the field contents, using a local variable `t` also of type `elem`. The test code at the end creates a pair of integers and a pair of strings, and prints

```
Hello,world!
42
```

to the standard output. (Here `+` is string concatenation.)

We consider two ways in which Java may simulate polymorphism. The first method is to macro expand a new version of the `Pair` class for each type at which it is instantiated.

Example 2.3 Homogenous translation of polymorphism into Java

```
class Pair {
    Object x; Object y;
    Pair (Object x, Object y) {this.x = x; this.y = y;}
    void swap () {Object t = x; x = y; y = t;}
}

class Integer {
    int i;
    Integer (int i) { this.i = i; }
    int intValue() { return i; }
}

Pair p = new Pair((Object)"world!", (Object)"Hello,");
p.swap();
System.out.println((String)p.x + (String)p.y);

Pair q = new Pair((Object)new Integer(22),
                 (Object)new Integer(64));
q.swap();
System.out.println(((Integer)(q.x)).intValue() -
                  ((Integer)(q.y)).intValue());
```

We call this the *heterogenous* translation, and it is shown in Example 2.2.

The appearance of parameterised classes `Pair<String>` and `Pair<int>` causes the creation of the expanded classes `Pair_String` and `Pair_int`, within which each occurrence of the type variable `elem` is replaced by the types `String` and `int`, respectively.

The second method is to replace the type variable `elem` by the class `Object`, the top of the class hierarchy. We call this the *homogenous* translation, and it is shown in Example 2.3.

The key to this translation is that a value of any type may be converted into type `Object`, and later recovered. Every type in Java is either a reference type or one of the eight base types, such as `int`. Each base type has a corresponding reference type, such as `Integer`, the relevant fragment of which appears in Example 2.3.

If `v` is a variable of reference type, say `String`, then it is converted into an object `o` by widening `(Object)s`, and converted back by narrowing `(String)o`. If `v` is a value of base type, say `int`, then it is converted to an object `o` by `(Object)(new Integer(v))`, and converted back by `((Integer)o).intValue()`. (In Java, widening may be implicit, but we write the cast `(Object)` explicitly for clarity.)

Java programmers can and do use idioms similar to the heterogenous and homogenous translations given here. Given the code duplication of the first, and the lengthy conversions of the second, the advantages of direct support for polymorphism are clear.

2.1 Bounded parametric polymorphism

Subtyping plays a central role in object-oriented languages, in the form of inheritance. In Java, single inheritance is indicated via subclassing, and multiple inheritance via interfaces. Thus, *Pizza* provides bounded polymorphism, where a type variable may take on any class that is a subclass of a given class, or any class that implements a given interface.

Example 2.4 Bounded polymorphism in *Pizza*

```
interface Ord<elem> {
    boolean less(elem o);
}

class Pair<elem implements Ord<elem>> {
    elem x; elem y;
    Pair (elem x, elem y) { this.x = x; this.y = y; }
    elem min() {if (x.less(y)) return x; else return y; }
}

class OrdInt implements Ord<OrdInt> {
    int i;
    OrdInt (int i) { this.i = i; }
    int intValue() { return i; }
    public boolean less(OrdInt o) { return i < o.intValue(); }
}

Pair<OrdInt> p = new Pair(new OrdInt(22),
                        new OrdInt(64));
System.out.println(p.min().intValue());
```

Pizza also allows interfaces to be parameterised, just as classes are. Parameterised interfaces allow one to express precisely the type of operations where both arguments are of the same type, a notoriously hard problem for object-oriented languages [BCC⁺96].

To demonstrate, we modify our previous example to find the minimum of a pair of elements, as shown in Example 2.4.

The interface `Ord` is parameterised on the type `elem`, which here ranges over all types, and specifies a method `less` with an argument of type `elem`.

The class `Pair` is also parameterised on the type `elem`, but here `elem` is constrained so to be a type that implements the interface `Ord<elem>`. Note that `elem` appears both as the bounded variable and in the bound: this form of recursive bound is well known to theorists of object-oriented type systems, and goes by the name of *F-bounded* polymorphism [CCH⁺89]. The method `min` is defined using the method `less`, which is invoked for object `x` of type `elem`, and has argument `y` also of type `elem`. The class `OrdInt` is similar to the class `Integer`, except that it also implements the interface `Ord<OrdInt>`. Hence, the class `OrdInt` is suitable as a parameter for the class `Pair`. The test code at the end creates a pair of ordered integers, and prints the minimum of the two to the standard output. The exercise of defining `OrdInt` is unavoidable, because Java provides no way for a base type to implement an interface. This is one of a number of points at which the Java designers promote simplicity over convenience, and *Pizza* follows their lead.

Again, we consider two ways in which Java may simulate bounded polymorphism. The heterogenous translation, again based on macro expansion, is shown in Example 2.5.

The appearance of the parameterised class `Pair<OrdInt>` and interface `Ord<OrdInt>` causes the creation of the expanded class `Pair_OrdInt` and interface `Ord_OrdInt`, within which each occurrence of the type variable `elem` is replaced by the type `OrdInt`. Once the code is expanded, the interface `Ord_OrdInt` plays no useful role, and all references to it may be deleted.

The homogenous translation, again based on replacing

Example 2.5 Heterogenous translation of bounded polymorphism into Java

```
interface Ord_OrdInt {
    boolean less(OrdInt o);
}

class Pair_OrdInt {
    OrdInt x; OrdInt y;
    Pair_OrdInt (OrdInt x, OrdInt y) { this.x = x; this.y = y; }
    OrdInt min() {if (x.less(y)) return x; else return y; }
}

class OrdInt implements Ord_OrdInt {
    int i;
    OrdInt (int i) { this.i = i; }
    int intValue() { return i; }
    boolean less(OrdInt o) { return i < o.intValue(); }
}

Pair_OrdInt p = new Pair_OrdInt(new OrdInt(22),
                               new OrdInt(64));
System.out.println(p.min().intValue());
```

elem by some fixed type, is shown in Example 2.6. The unbounded type variable `elem` in the interface `Ord` is replaced by the class `Object`, the top of the class hierarchy. The bounded type variable `elem` in the class `Pair` is replaced by the interface `Ord`, the homogenous version of its bound.

In the homogenous version, there is a mismatch between the type of `less` in the interface `Ord`, where it expects an argument of type `Object`, and in the class `OrdInt`, where it expects an argument of type `OrdInt`. This is patched in the interface `Ord` renaming `less` to `less_Ord`; and in the class `OrdInt` by adding a ‘bridge’ definition of `less_Ord` in terms of `less`. The ‘bridge’ adds suitable casts to connect the types. Suitable casts are also added to the test code.

Again, Java programmers can and do use idioms like the above. The idiomatic Java programs are slightly simpler: the useless interface `Ord_OrdInt` can be dropped from the heterogenous translation, and `less` and `less_Ord` can be merged in the homogenous translation. Nonetheless, the original `Pizza` is simpler and more expressive, making the advantages of direct support for bounded polymorphism clear.

2.2 Arrays

Arrays are the classic example of parametric polymorphism: for each type `elem` there is the corresponding array type `elem[]`. Nonetheless, fitting Java arrays with `Pizza` polymorphism poses some problems.

As a trivial example, we consider rotating the entries of an array, the `Pizza` code for which appears in Example 2.7. The method `rotate` applies to arrays of any element type. Whereas before `<elem>` was used as an (explicitly instantiated) parameter to a class, here it is used as an (implicitly instantiated) parameter to a method. The two calls to `rotate` instantiate it to `String` and `int`, respectively, and the test code prints `Hello world!` and `2 3 4 1`. (In Java, a `static` method corresponds to a traditional function call, and is invoked by naming the class rather than by referencing an instance.)

Example 2.6 Homogenous translation of bounded polymorphism into Java

```
interface Ord {
    boolean less_Ord(Object o);
}

class Pair {
    Ord x; Ord y;
    Pair (Ord x, Ord y) { this.x = x; this.y = y; }
    Ord min() {if (x.less_Ord(y)) return x; else return y; }
}

class OrdInt implements Ord {
    int i;
    OrdInt (int i) { this.i = i; }
    int intValue() { return i; }
    boolean less(OrdInt o) { return i < o.intValue(); }
    boolean less_Ord(Object o) {
        return this.less((OrdInt)o); }
}

Pair p = new Pair (new OrdInt(22), new OrdInt(64));
System.out.println(((OrdInt)(p.min())).intValue());
```

The heterogenous translation is straightforward and won’t be further treated here, but the homogenous translation raises some interesting issues.

Naively, we might expect to translate the type `elem[]` as `Object[]`. This works fine when `elem` is a reference type like `String`, since it is easy to coerce from `String[]` to `Object[]` and back. But it falls flat when `elem` is a base type like `int`, since the only way to coerce from `int[]` to `Object[]` is to copy the whole array. We reject coercion based on copying for two reasons. First, copying is too expensive: `Pizza`, like Java, follows the principle that all coercions should occur in constant time, irrespective of the size of the coerced data. Second, copying loses the semantics of arrays: an update of a formal array parameter must also affect the actual array.

The solution adopted is to introduce a new type `Array`, shown in Example 2.8, to act as a wrapper around the Java array. The abstract class `Array` supports operations to find the length of the array, to get the object at index `i`, and to set index `i` to contain object `o`. Two subclasses are given, one where the array is implemented as an array of objects, and one where the array is implemented as an array of integers. Not shown are the six other subclasses corresponding to the six other base types of Java.

The homogenous translation of Example 2.7 is shown in Example 2.9. Occurrences of the type `elem[]` are replaced by the type `Array`, and expressions to find the length, or get or set elements, of arrays of type `elem[]` are replaced by the corresponding method calls of class `Array`. Occurrences of the type `elem` on its own are replaced by `Object`, just as in the naive translation. In the test code, the arrays of types `String[]` and `int[]` are coerced to type `Array` by calls to the constructors `Array_Object` and `Array_int` respectively.

Arrays, subtyping, and polymorphism. The Java designers took some pains to mimic parametric polymorphism for arrays via subtyping. For instance, since `String` is a subtype of `Object` they make `String[]` a subtype of `Object[]`. This requires extra run-time checks to ensure soundness: an ar-

Example 2.7 Polymorphic arrays in Pizza.

```

class Rotate {
    static <elem> void rotate(elem[] x) {
        elem t = x[0];
        for (int i = 0; i < x.length-1; i++) { x[i] = x[i+1]; }
        x[x.length] = t;
    }
}

String[] u = {"world!", "Hello"};
Rotate.rotate(u);
System.out.println(u[0]+" "+u[1]);

int[] v = {1,2,3,4}
Rotate.rotate(v);
System.out.println(v[0]+" "+v[1]+" "+v[2]+" "+v[3]);

```

Example 2.8 Utility arrays in Java.

```

abstract class Array {
    int length();
    Object get(int i);
    void set(int i, Object o);
}

class Array_Object extends Array {
    Object[] x;
    Array_Object(Object[] x) { this.x = x; }
    int length() { return x.length; }
    Object get(int i) { return x[i]; }
    void set(int i, Object o) { x[i] = o; }
}

class Array_int extends Array {
    int[] x;
    Array_int(int[] x) { this.x = x; }
    int length() { return x.length; }
    Object get(int i) { return new Integer(x[i]); }
    void set(int i, Object o) { x[i] = ((Integer)o).intValue(); }
}

```

ray is labeled with its run-time type, and assignment to an array checks compatibility with this type. Thus, a `String[]` array actual may be passed to an `Object[]` formal, and any assignment within the procedure is checked at run-time to ensure that the assigned element is not merely an `Object` but in fact a `String`.

Pizza uses instantiation, not subtyping, to match `String[]` to `elem[]`, and so it is tempting to eliminate any notion of subtyping between arrays, and hence eliminate the need for run-time checks. Alas, since Pizza compiles into the Java Virtual Machine, the checks are unavoidable. We therefore retain the subtyping relation between arrays in Pizza, which is necessary for Pizza to be a superset of Java. This mismatch between languages is ironic: Java supports polymorphic arrays as best it can, and as a result Pizza must offer less good support for them.

Array creation. The homogenous translation for array creation is problematic. Consider the phrase `new elem[size]`, where `elem` is a type variable and `size` is an integer. At first

Example 2.9 Homogenous translation of polymorphic arrays into Java.

```

class Rotate {
    static void rotate(Array x) {
        Object t = x.get(0);
        for (int i = 0; i < x.length()-1; i++) { x.set(i,x.get(i+1)); }
        x.set(x.length(),t);
    }
}

String[] u = {"world!", "Hello"};
Rotate.rotate(new Array_Object(u));
System.out.println(u[0]+" "+u[1]);

int[] v = {1,2,3,4}
Rotate.rotate(new Array_int(v));
System.out.println(v[0]+" "+v[1]+" "+v[2]+" "+v[3]);

```

blush, the translation `new Object[size]` might seem sensible, but a moment of thought shows this fails for base types, since they are allocated differently. An additional moment of thought shows that it also fails for reference types. For instance, if `elem` is `String`, then the result is an array of objects that all happen to be strings, and there is no way in Java to cast this to an array of strings.

The remaining alternative is to pass explicit type information around at run-time. The current version of Java makes this depressingly difficult: one may at run-time extract a representation of the type of an object, but one may not use this to create an array of that type. New ‘reflection’ facilities for the next version of Java solve this problem, and may cause us to reconsider our choice [Sun96b].

For now, we simply prohibit polymorphic array creation. In its place, one may use higher-order functions, passing a function that explicitly creates the array. This is analogous to the passing of type parameters, where instead of implicitly passing a parameter representing the type we explicitly pass a function to create an array of that type. Higher-order functions are the subject of the next section.

3 Higher-order functions

It can be convenient to treat functions as data: to pass them as arguments, to return them as results, or to store them in variables or data structures. Such a feature goes by the name of *higher-order functions* or *first-class functions*.

The object-oriented style partly supports higher-order functions, since functions are implemented by methods, methods are parts of objects, and objects may themselves be passed, returned, or stored. Indeed, we will implement higher-order functions as objects. But our translation will be rather lengthy, making clear why higher-order functions are sometimes more convenient than objects as a way of structuring programs.

The body of a function abstraction may refer to three sorts of variables:

- *formal parameters* declared in the abstraction header,
- *free variables* declared in an enclosing scope, and
- *instance variables* declared in the enclosing class.

Example 3.1 Higher-order functions in Pizza.

```
class Radix {
  int n = 0;
  (char)→boolean radix(int r) {
    return fun (char c)→boolean {
      n++; return '0' <= c && c < '0'+r;
    };
  }
  String test () {
    (char)→boolean f = radix(8);
    return f('0')+ " "+f('9')+ " "+n;
  }
}
```

All three sorts of variable appear in Example 3.1.

In the body of the abstraction, *c* is a formal parameter, *r* is a free variable, and *n* is an instance variable. The body returns `true` if the character *c* represents a digit in radix *r*, and increments *n* each time it is called. Thus, calling `new Radix().test()` returns `"true false 2"`.

In Java, the variable `this` denotes the receiver of a method (it is called `self` in some other object-oriented languages). The *receiver* of an abstraction is the same as the receiver of the method within which it appears; so `this` and instance variables follow a static scope discipline with regard to abstractions.

In general, the function type

$$(t_1, \dots, t_n) \rightarrow t_0$$

denotes a function with a result of type t_0 and argument of types t_1, \dots, t_n . Here t_0 , but not t_1, \dots, t_n , may be `void`, and $n \geq 0$. The function abstraction

$$\text{fun } (t_1 \ x_1, \dots, t_n \ x_n) \rightarrow t_0 \ s$$

denotes a function of the above type with variables x_1, \dots, x_n as its formals, and statement *s* as its body.

Given Java's syntactic tradition, one might expect the notation $t_0(t_1, \dots, t_n)$ for function types. This alternate notation was rejected for two reasons. First, functions that return functions become unnecessarily confusing. Consider a function call `f(i)(c)` where *i* is an `int` and *c* is a `char`. What is the type of *f*? In our notation it is simply $(\text{int}) \rightarrow (\text{char}) \rightarrow \text{boolean}$. In the alternate notation, it is $(\text{boolean}(\text{char}))(\text{int})$ and the reader must decode the reversal of order. Second, an omitted semicolon can lead to a confused parse and a perplexing error message. Compare `f(a); x=y;` where `f(a)` is a method call and `x=y` is an assignment, with `f(a) x=y` where `f(a)` is a type, *x* is a newly declared variable and `=y` is an initialiser.

Formal parameters are passed by value, so any updates to them are not seen outside the function body; but instance variables are accessed by reference, so any updates are seen outside the function body. This is just as in Java methods. What about free variables? Because Java provides no reference parameters, it would be most convenient to treat these as passed by value, just as formal parameters. Passing free variables by reference is also possible, but requires that the variables be implemented as single-element arrays. Our current implementation of closures conceptually passes variables by reference, but contains a conservative analysis that

Example 3.2 Heterogenous translation of higher-order functions into Java.

```
abstract class Closure_CB {
  abstract boolean apply_CB (char c);
}

class Closure_1 extends Closure_CB {
  Radix receiver;
  int r;
  Closure_1 (Radix receiver, int r) {
    this.receiver = receiver; this.r = r;
  }
  boolean apply_CB (char c) {
    return receiver.apply_1(r, c);
  }
}
```

```
class Radix {
  int n = 0;
  boolean apply_1 (int r, char c) {
    n++; return '0' <= c && c < '0'+r;
  }
  Closure_CB radix(int r) {
    return new Closure_1 (this, r);
  }
  String test () {
    Closure_CB f = radix(8);
    return f.apply_CB('0')+ " "+f.apply_CB('9')+ " "+n;
  }
}
```

determines whether variables might possibly be assigned to while captured in a closure. If a free variable is known to be immutable for the duration of a closure the more efficient value passing scheme is used. This choice was one of the more finely balanced, however, as there are also good reasons to pass all free variables by value.

3.1 Heterogenous translation

We have found that while some Java programmers have difficulty understanding the notion of higher-order functions, they find it easier to follow once the translation scheme has been explained. We explain the heterogenous translation scheme by example, but it should be clear how it works in the general case.

The heterogenous translation introduces one abstract class for each function type in a program, and one new class for each function abstraction in a program. The abstract class captures the type of a higher-order function, while the new class implements a closure.

The heterogenous translation of Example 3.1 is shown in Example 3.2.

First, each function type in the original introduces an abstract class in the translation, specifying an `apply` method for that type. Thus, the function type $(\text{char}) \rightarrow \text{boolean}$ in the original introduces the abstract class `Closure_CB` in the translation. This specifies a method `apply_CB` that expects an argument of type `char` and returns a result of type `boolean`.

Second, each function abstraction in the original introduces a class in the translation, which is a subclass of the

class corresponding to the function type. Thus, the one function abstraction in `radix` introduces the class `Closure_1` in the translation, which is a subclass of `Closure_CB`. The `apply` method for the type `apply_CB` calls the `apply` method for the abstraction `apply_1`.

It is necessary to have separate `apply` methods for each function type (e.g., `apply_CB`) and for each abstraction (e.g., `apply_1`). Each function type defines an `apply` method in the closure (so it is accessible wherever the function type is accessible), whereas each abstraction defines an `apply` method in the original class (so it may access private instance variables).

3.2 Homogenous translation

Functions exhibit parametric polymorphism in their argument and result types, and so the function translation has both heterogenous and homogenous forms. Whereas the heterogenous translation introduces one class for each closure, the homogenous translation represents all closures as instances of a single class. While the heterogenous translation represents each free variable and argument and result with its correct type, the homogenous translation treats free variables and arguments as arrays of type `Object` and results as of type `Object`. Thus, the homogenous translation is more compact, but exploits less static type information and so must do more work at run time. We omit an example of this translation, as it is lengthy but straightforward.

4 Algebraic types

The final addition to `Pizza` is algebraic types and pattern matching. Object types and inheritance are complementary to algebraic types and matching. Object types and inheritance make it easy to extend the set of constructors for the type, so long as the set of operations is relatively fixed. Conversely, algebraic types and matching make it easy to add new operations over the type, so long as the set of constructors is relatively fixed. The former might be useful for building a prototype interpreter for a new programming language, where one often wants to add new language constructs, but the set of operations is small and fixed (evaluate, print). The latter might be useful for building an optimising compiler for a mature language, where one often wants to add new passes, but the set of language constructs is fixed.

An algebraic type for lists of characters is shown in Example 4.1. The two `case` declarations introduce constructors for the algebraic type: `Nil` to represent the empty list; and `Cons` to represent a list cell with two fields, a character `head`, and a list `tail`. The method `append` shows how the `switch` statement may be used to pattern match against a given list. Again there are two cases, for `Nil` and for `Cons`, and the second case binds the freshly declared variables `x` and `xs` to the head and tail of the list cell. The test code binds `zs` to the list `Cons('a',Cons('b',Cons('c',Nil)))`.

The label for a case contains type declarations for each of the bound variables, as in `Cons(char head, list tail)`. We might have chosen a syntax that omits these types, since they can be inferred from the type of the case selector, but we decided to retain them, to maintain the convention that every newly introduced variable is preceded by its type. Further, doing so eliminates any possible confusion between a variable and a constructor of no arguments. (This possible confusion is a well-known problem in functional languages. Miranda and

Example 4.1 Algebraic types in `Pizza`

```
class List {
  case Nil;
  case Cons(char head, List tail);
  List append (List ys) {
    switch (this) {
      case Nil:
        return ys;
      case Cons(char x, List xs):
        return Cons(x, xs.append(ys));
    }
  }
}
```

```
List zs = Cons('a',Cons('b',Nil)).append(Cons('c',Nil));
```

Haskell treat it by requiring variables and constructors to be lexically distinct, the former beginning with a small letter and the latter with a capital.)

In Java, a missing `break` causes control to fall through to the following case, which is a source of frequent errors. `Pizza`, being a superset of Java, is stuck with this design. However, it makes no sense to fall into a case that introduces new bound variables, so we have a good excuse to do the right thing and flag such cases as erroneous at compile time.

The translation from `Pizza` to Java is shown in Example 4.2. The translated class includes a tag indicating which algebraic constructor is represented. Each `case` declaration introduces a subclass with a constructor that initialises the tag and the fields. The `switch` construct is translated to a `switch` on the tag, and each case initialises any bound variables to the corresponding fields.

If `xs` is a list, the notations `xs instanceof Nil`, `xs instanceof Cons`, `((Cons)xs).head` and `((Cons)xs).tail` are valid in `Pizza`, and they require no translation to be equally valid in Java.

As a final demonstration of the power of our techniques, Example 4.3 demonstrates a polymorphic algebraic type with higher-order functions. Note the use of nested pattern matching in `zip`, and the use of the phrase `` to introduce `B` as a type variable in `map`. We invite readers unconvinced of the utility of `Pizza` to attempt to program the same functionality directly in Java.

5 Typing issues

`Pizza` uses a mixture of explicit and implicit polymorphism, in that types of variables and methods are given explicitly where they are declared, but instantiated implicitly where they are used, in a way that corresponds to the Hindley-Milner type system [DM82]. Type variables range over subtypes of one or more types, giving us bounded polymorphism [CCH⁺89, BTCGS91], and since the type variable may recursively appear in its own bound we have F-bounded polymorphism [CCH⁺89, BTCGS91]. The mixture of implicit instantiation with F-bounded polymorphism resembles type classes as used in Haskell, and may be implemented using similar techniques [WB89, Oho92, Jon93].

Building on this work, the major remaining difficulties in the design of `Pizza`'s type system are to integrate subtyping with parametric polymorphism, and to integrate static and dynamic typing.

Example 4.2 Translation of algebraic types into Java

```
class List {
  final int Nil_tag = 0;
  final int Cons_tag = 1;
  int tag;
  List append (List ys) {
    switch (this.tag) {
      case Nil_tag:
        return ys;
      case Cons_tag:
        char x = ((Cons)this).head;
        List xs = ((Cons)this).tail;
        return new Cons(x, xs.append(ys));
    }
  }
}

class Nil extends List {
  Nil() {
    this.tag = Nil_tag;
  }
}

class Cons extends List {
  char head;
  List tail;
  Cons(char head, List tail) {
    this.tag = Cons_tag;
    this.head = head; this.tail = tail;
  }
}

List zs = new Cons('a',new Cons('b',new Nil()))
    .append(new Cons('c',new Nil()));
```

5.1 Integrating Subtyping and Parametric Polymorphism

It is not entirely straightforward to combine subtyping with implicit polymorphism of the sort used in Pizza.

Subtyping should not extend through constructors. To see why, consider Example 5.1. Since `String` is a subtype of `Object`, it seems natural to consider `Cell<String>` a subclass of `Cell<Object>`. But this would be unsound, as demonstrated in the test code, which tries to assign an `Integer` to a `String`. Pizza avoids the problem by making the marked line illegal: `Cell<String>` is not considered a subtype of `Cell<Object>`, so the assignment is not allowed.

A common approach to subtyping is based on the principle of *subsumption*: if an expression has type A , and A is a subtype of B , then the expression also has type B . Subsumption and implicit polymorphism do not mix well, as is shown by considering the expression

```
new Cell("Hello")
```

Clearly, one type it might have is `Cell<String>`. But since `String` is a subtype of `Object`, in the presence of subsumption the expression `"Hello"` also has the type `Object`, and so the whole expression should also have the type `Cell<Object>`. This type ambiguity introduced by subsumption would be fine if `Cell<String>` were a subtype of `Cell<Object>`, but we've already seen why that should not be the case.

Example 4.3 Polymorphism, higher-order functions, and algebraic types

```
class Pair<A,B> {
  case Pair(A fst,B snd);
}

class List<A> {
  case Nil;
  case Cons(A head, List<A> tail);
  <B> List<B> map ((A)→B f) {
    switch (this) {
      case Nil:
        return Nil;
      case Cons(A x, List<A> xs):
        return Cons(f(x), xs.map(f));
    }
  }
  <B> List<Pair<A,B>> zip (List<B> ys) {
    switch (Pair(this,ys)) {
      case Pair(Nil,Nil):
        return Nil;
      case Pair(Cons(A x, List<A> xs),
                Cons(B y, List<B> ys)):
        return Cons(Pair(x,y), xs.zip(ys));
    }
  }
}
```

Moreover, there is no F-bounded type which subsumes both `Cell<String>` and `Cell<Object>`. To be sure, it is possible to generalize F-bounded polymorphism so that such a type would exist. For instance, if we allow lower as well as upper bounds for type variables we would find that the type scheme

$$\forall a. \text{String extends } a \Rightarrow \text{Cell}\langle a \rangle$$

contains both `Cell<String>` and `Cell<Object>` as instances. It is possible to come up with type systems that allow these more general constraints [AW93, ESTZ95], but there remain challenging problems in the areas of efficiency of type checking and user diagnostics. It seems that more research is needed until systems like these can be applied in mainstream languages.

So, we eschew subsumption and require that type variables always match types exactly. To understand the consequences of this design, consider the class in Example 5.2 and the following ill-typed expression:

```
Subsume.choose(true, "Hello", new Integer(42))
```

One might assume that since `String` and `Integer` are subtypes of `Object`, this expression would be well-typed by taking `elem` to be the type `Object`. But since Pizza uses exact type matching, this expression is in fact ill-typed. It can be made well-typed by introducing explicit widening casts.

```
Subsume.choose(true, (Object)"Hello",
               (Object)(new Integer(42)))
```

Interestingly, Java does not have subsumption either. For a simple demonstration, consider the following conditional expression.

```
b ? "Hello" : new Integer(42)
```

Example 5.1 Subtyping and parameterised types

```
class Cell<elem> {
    elem x;
    Cell (elem x) { this.x = x; }
    void set(elem x) { this.x = x; }
    elem get() { return x; }
}

Cell<String> sc = new Cell("Hello");
Cell<Object> oc = sc; // illegal
oc.set(new Integer(42));
String s = sc.get();
```

Example 5.2 Examples related to subsumption

```
class Subsume {
    static <elem> elem choose(boolean b, elem x, elem y) {
        if (b) return x; else return y;
    }
    static Object choose1(boolean b, Object x, Object y) {
        if (b) return x; else return y;
    }
    static <elemx extends Object, elemy extends Object>
    Object choose2(boolean b, elemx x, elemy y) {
        if (b) return x; else return y;
    }
}
```

By subsumption, this expression should have type `Object`, since both `Integer` and `String` are subtypes of `Object`. However, in Java this expression is ill-typed: one branch of the conditional must be a subtype of the other branch.

Java does however allow a limited form of subsumption for method calls, in that the type of the actual argument may be a subtype of the type of the formal, and therefore `Pizza` must also allow it. Fortunately, this limited form of subsumption can be explained in terms of bounded polymorphism. Consider the following expression:

```
Subsume.choose1(true, "Hello", new Integer(42))
```

This is well-typed in both Java and `Pizza`, because the actual argument types `String` and `Integer` are implicitly widened to the formal argument type `Object`. Note that the behaviour of `choose1` is mimicked precisely by `choose2`, which allows two actual arguments of any two types that are subtypes of `Object`, and which always returns an `Object`. Thus, the methods `choose1` and `choose2` are equivalent to each other in that a call to one is valid exactly when a call to the other is; but neither is equivalent to `choose`.

This equivalence is important, because it shows that bounded polymorphism can be used to implement the form of subsumption found in `Pizza`. All that is necessary is to *complete* each function type, by replacing any formal argument type `A` which is not a type variable by a new quantified variable with bound `A`. A similar technique to model subtyping by matching has been advocated by Bruce [Bru97].

(Details of completion are described in the appendix.)

Example 5.3 Existentially quantified type variables in patterns.

```
class List<A> {
    ...
    boolean equals(Object other) {
        if (other instanceof List) {
            switch Pair(this, (List)other) {
                case Pair(Null, Null):
                    return true;
                case <B> Pair(Cons(A x, List<A> xs),
                            Cons(B y, List<B> ys)):
                    return x.equals(y) && xs.equals(ys);
                default:
                    return false;
            }
        } else return false;
    }
}
```

5.2 Integrating Dynamic Typing

Java provides three expression forms that explicitly mention types: creation of new objects, casting, and testing whether an object is an instance of a given class. In each of these one mentions only a class name; any parameters of the class are omitted.

For example, using the polymorphic lists of Example 4.3, the following code fragment is legal, though it will raise an error at run-time.

```
List<String> xs = Cons("a", Cons("b", Cons("c", Nil)));
Object obj = (Object)xs;
int x = ((List)obj).length();
```

Why say `(List)` rather than `(List<String>)`? Actually, the latter is more useful, but it would be too expensive to implement in the homogenous translation: the entire list needs to be traversed, checking that each element is a string.

What type should be assigned to a term after a cast? In the example above, `(List)obj` has the existential type

$$\exists a. \text{List}\langle a \rangle.$$

Existential types were originally introduced to model abstract types [CW85, MP88], but there is also precedence to their modeling dynamic types [LM91].

Figure 5.3 illustrates how existentially qualified patterns can be used to define an `equals` method for lists. (In Java, all classes inherit an `equals` method from class `Object`, which they may override.) The method is passed an argument `other` of type `Object`, and a run-time test determines whether this object belongs to class `List`. If so, then it has type `List` for an arbitrary type `B`. Hence the code must be valid for any type `B`, which is indicated by adding the quantifier `` to the pattern match. Note that the syntax exploits the logical identity between $(\exists X.P) \rightarrow Q$ and $\forall X.(P \rightarrow Q)$, so that angle brackets can always be read as universal quantification.

Type checking existential types is straightforward [LO94]. The type checker treats existentially bound variables as fresh type constants, which cannot be instantiated to anything.

(Details of existentials are described in the appendix.)

6 Rough edges

There are surprisingly few places where we could not achieve a good fit of Pizza to Java. We list some of these here: casting, visibility, dynamic loading, interfaces to built-in classes, tail calls, and arrays.

Casting. Java ensures safe execution by inserting a runtime test at every narrowing from a superclass to a subclass. Pizza has a more sophisticated type system that renders some such tests redundant. Translating Pizza to Java (or to the Java Virtual Machine) necessarily incurs this modest extra cost.

Java also promotes safety by limiting the casting operations between base types. By and large this is desirable, but it is a hindrance when implementing parametric polymorphism. For instance, instantiations of a polymorphic class at types `int` and `float` must have separate implementations in the heterogenous translation, even though the word-level operations are identical.

These modest costs could be avoided only by altering the Java Virtual Machine (for instance, as suggested by [BLM96]), or by compiling Pizza directly to some other portable low-level code, such as Microsoft’s Omnicode or Lucent’s Dis.

Visibility. Java provides four visibility levels: *private*, visible only within the class; *default*, visible only within the package containing the class; *protected*, visible only within the package containing the class and within subclasses of the class; *public*, visible everywhere. Classes can only have default or public visibility; fields and methods of a class may have any of the four levels.

A function abstraction in Pizza defines a new class in Java (to represent the closure) and a new method in the original class (to represent the abstraction body). The constructor for the closure class should be invoked only in the original class, and the body method should be invoked only within the closure class. Java provides no way to enforce this style of visibility. Instead, the closure class and body method must be visible at least to the entire package containing the original class. One cannot guarantee appropriate access to closures, unless one relies on dynamically allocated keys and run-time checks.

For similar reasons, all fields of an algebraic type must have default or public visibility, even when private or protected visibility may be desirable.

Dynamic loading. Java provides facilities for dynamic code loading. In a native environment the user may specify a class loader to locate code corresponding to a given class name. The heterogenous translation can benefit by using a class loader to generate on the fly the code for an instance of a parameterised class.

Unfortunately, a fixed class loader must be used for Java code executed by a web client. With a little work, it should be possible to allow user-defined class loaders without compromising security.

Interfaces for built-in classes. In Section 2.1 we needed to declare a new class `OrdInt` that implements the interface `Ord`. We could not simply extend the `Integer` class, because for efficiency reasons it is final, and can have no subclasses. A similar problem arises for `String`. This is a problem for Java proper, but it may be aggravated in Pizza, which enhances the significance of interfaces by their use for bounded polymorphism.

Tail calls. We would like for Pizza to support tail calls [SS76], but this is difficult without support in the Java Vir-

tual Machine. We hope for such support in future versions of Java.

Arrays. As discussed previously, there is a rather poor fit between polymorphic arrays in Pizza and their translation into Java.

7 Conclusion

Pizza extends Java with parametric polymorphism, higher-order functions, and algebraic data types; and is defined by translation into Java. The lessons we learned include the following:

- Our requirement that Pizza translate into Java strongly constrained the design space. Despite this, it turns out that our new features integrate well: Pizza fits smoothly to Java, with relatively few rough edges.
- It is useful to isolate two idioms for translating polymorphism, which we call *heterogenous* and *homogenous*. Adding bounded polymorphism proved surprisingly easy, while polymorphic arrays proved surprisingly difficult. Ironically, the use of subtyping in Java to mimic parametric polymorphism prohibits adopting the best model of arrays in Pizza.
- Standard notions from type theory help us model much of the behaviour of the Java and Pizza type systems. We made use of F-bounded polymorphism and existential types. Subsumption was not so helpful, and we introduced a notion of completion instead.

We are now adding Pizza features to our EspressoGrinder compiler for Java, and look forward to feedback from experience with our design.

Acknowledgments

We’d like to thank the anonymous referees for their thorough and helpful comments.

A Syntax extensions

Figure 1 sketches syntax extensions of Pizza with respect to Java in EBNF format [Wir77].

B Pizza’s Type System

Since full Pizza is an extension of Java, and Java is too complex for a concise formal definition, we concentrate here on a subset of Pizza that reflects essential aspects of our extensions. The abstract syntax of Mini-Pizza programs is given in Figure 2.

Preliminaries: We use vector notation \bar{A} to indicate a sequence A_1, \dots, A_n . If $\bar{A} = A_1, \dots, A_n$ and $\bar{B} = B_1, \dots, B_n$ and \oplus is a binary operator then $\bar{A} \oplus \bar{B}$ stands for $A_1 \oplus B_1, \dots, A_n \oplus B_n$. If \bar{A} and \bar{B} have different lengths then $\bar{A} \oplus \bar{B}$ is not defined. Each predicate p is promoted to a predicate over vectors: $p(A_1, \dots, A_n)$ is interpreted as $p(A_1) \wedge \dots \wedge p(A_n)$. We use ‘ \leq ’ to express class extension, rather than the ‘extends’ keyword that Java and Pizza source programs use.

Overview of Mini Pizza: A program consists of a sequence of class declarations K ; we do not consider packages.

<i>expression</i>	= ... fun <i>type</i> ([<i>params</i>]) [throws <i>types</i>] <i>block</i>
<i>type</i>	= ... ([<i>types</i>]) [throws <i>types</i>] \rightarrow <i>type</i> <i>qualident</i> [< <i>types</i> >]
<i>typevardcl</i>	= <i>ident</i> <i>ident</i> implements <i>type</i> <i>ident</i> extends <i>type</i>
<i>typeformals</i>	= < <i>typevardcl</i> {, <i>typevardcl</i> } >
<i>classdef</i>	= class <i>ident</i> [<i>typeformals</i>] [extends <i>type</i>] [implements <i>types</i>] <i>classBlock</i>
<i>interfacedef</i>	= interface <i>ident</i> [<i>typeformals</i>] [extends <i>types</i>] <i>interfaceBlock</i>
<i>methoddef</i>	= <i>modifiers</i> [<i>typeformals</i>] <i>ident</i> ([<i>params</i>]) { [] } [throws <i>types</i>] (<i>block</i> ;)
<i>classBlock</i>	= ... <i>casedef</i>
<i>casedef</i>	= <i>modifiers</i> [<i>typeformals</i>] case <i>ident</i> [([<i>params</i>])];
<i>case</i>	= case [<i>typeformals</i>] <i>pattern</i> : <i>statements</i> default : <i>statements</i>
<i>pattern</i>	= <i>expression</i> <i>type ident</i> <i>qualident</i> (<i>apat</i> {, <i>apat</i> })

Figure 1: Pizza’s syntax extensions.

Each class definition contains the name of the defined class, c , the class parameters $\bar{\alpha}$ with their bounds, the type that c extends, and a sequence of member declarations. We require that every class extends another class, except for class **Object**, which extends itself. For space reason we omit interfaces in this summary, but adding them would be straightforward.

Definitions in a class body define either variables or functions. Variables are always object fields and functions are always object methods. We do not consider static variables or functions, and also do not deal with access specifiers in declarations. To keep the presentation manageable, we do not consider overloading and assume that every identifier is used only once in a class, and that identifiers in different classes with the same name correspond to methods that are in an “override” relationship.

As Pizza statements we have expression statements E ; function returns **return** E ; statement composition $S_1 S_2$, and conditionals **if** (E) S_1 **else** S_2 . As Pizza expressions we have identifiers x , selection $E.x$, (higher-order) function application $E(E_1, \dots, E_n)$, assignment $E_1 = E_2$, object creation **new** $c()$ and type casts $(c)E$.

Types are either type variables α , or parameterized classes $c\langle\bar{A}\rangle$, or function types $(\bar{A}) \rightarrow B$. For simplicity we leave out Java’s primitive types such as **int** or **float**. Func-

Variables	x, y
Classids	c, d
Typevars	α, β
Program	$P = \bar{K}$
Classdcl	$K = \mathbf{class} \ c\langle\bar{\alpha} \leq \bar{\mathcal{B}}\rangle \ \mathbf{extends} \ C \ \{\bar{M}\}$
Memberdcl	$M = A \ x = E;$ $\langle\bar{\alpha} \leq \bar{\mathcal{B}}\rangle \ A \ x(\bar{B} \ \bar{y}) \ \{S\}$
Statement	$S = E;$ return E ; $S_1 \ S_2$ if (E) S_1 else S_2
Expression	$E = x$ $E.x$ $E(\bar{E})$ new $c()$ $E_1 = E_2$ $(c)E$
Classtype	$C = c\langle\bar{A}\rangle$
Type	$A, B = C$ $(\bar{A}) \rightarrow B$ α
Typescheme	$U = A$ $\forall\bar{\alpha} \leq \bar{\mathcal{B}}.U$ var A
Typesum	$X = A$ $\exists\bar{\alpha} \leq \bar{\mathcal{B}}.X$
Typebound	$\mathcal{B} = \bar{\mathcal{C}}$
Constraint	$\Sigma = \bar{\alpha} \leq \bar{\mathcal{B}}$
Typothesis	$\Gamma = \bar{x} : \bar{U}$
Classdcl	$D = c : \mathbf{class}(\bar{\alpha} \leq \bar{\mathcal{B}}, \Gamma, A)$
Classenv	$\Delta = \bar{D}$

Figure 2: Abstract Syntax of Mini Pizza.

tions may have F-bounded polymorphic type $\forall\bar{\alpha} \leq \bar{\mathcal{B}}.A$ and the type of a mutable variable is always of the form **var** A . These two forms are not proper types but belong to the syntactic category of typeschemes U .

In some statement and expression contexts we also admit existential types $\exists\bar{\alpha} \leq \bar{\mathcal{B}}.A$. Like typeschemes, these *typesums* have no written representation in Pizza programs; they are used only internally for assigning types to intermediate expressions. Quantifiers \forall and \exists bind lists $\bar{\alpha}$ of type variables; hence mutual recursion between type variable bounds is possible, as in the following example:

$$\exists\alpha \leq c\langle\beta\rangle, \beta \leq d\langle\alpha\rangle.A$$

Type judgements contain both a subtype constraint Σ and a typothesis Γ . For both environments we denote environment extension by new variables with an infix dot, i.e. $\Sigma.\alpha \leq \mathcal{B}$, $\Gamma.x : U$. Since subtyping in Java and Pizza is by declaration, subtyping rules depend on a class environment Δ , which is generated by the program’s class declarations.

Well-formedness of Programs: Type checking a pizza program proceeds in three phases.

1. Generate a class environment Δ ,
2. check that Δ is well-formed, and
3. check that every class is well-formed under Δ .

Phase 1: Generating Δ is straightforward. For each class definition

$$\mathbf{class} \ c\langle\bar{\alpha} \leq \bar{\mathcal{B}}\rangle \ \mathbf{extends} \ A \ \{\bar{D}\}$$

we make a binding that associates the class name c with an entry **class**($\bar{\alpha} \leq \bar{\mathcal{B}}, \Gamma, A$). The entry consists of the class parameters with their bounds, a local environment Γ that

$$\begin{array}{l}
\text{(Top)} \quad X \leq \text{Object} \\
\text{(Refl)} \quad \Sigma \vdash X \leq X \\
\text{(Trans)} \quad \frac{\Sigma \vdash X_1 \leq X_2 \quad \Sigma \vdash X_2 \leq X_3}{\Sigma \vdash X_1 \leq X_3} \\
(\alpha \leq) \quad \frac{\alpha \leq \mathcal{B} \in \Sigma \quad A \in \mathcal{B}}{\Sigma \vdash \alpha \leq A} \\
(c \leq) \quad \frac{c : \mathbf{class}(\overline{\alpha} \leq \overline{\mathcal{B}}, \Gamma, C) \in \Delta \quad \Sigma \vdash \overline{A} \leq \overline{\mathcal{B}}[\overline{\alpha} := \overline{A}]}{\Sigma \vdash c\langle \overline{A} \rangle \leq C[\overline{\alpha} := \overline{A}]} \\
(\exists \leq) \quad \frac{\Sigma, \overline{\alpha} \leq \overline{\mathcal{B}} \vdash X \leq X' \quad \alpha \notin tv(X')}{\Sigma \vdash (\exists \overline{\alpha} \leq \overline{\mathcal{B}}. X) \leq X'} \\
(\leq \exists) \quad \frac{\Sigma \vdash X \leq X'[\overline{\alpha} := \overline{A}] \quad \Sigma \vdash \overline{A} \leq \overline{\mathcal{B}}[\overline{\alpha} := \overline{A}]}{\Sigma \vdash X \leq (\exists \overline{\alpha} \leq \overline{\mathcal{B}}. X')}
\end{array}$$

Figure 3: The subtype relation $\Sigma \vdash X \leq X'$.

records the declared types of all class members, and the supertype of the class.

A class environment generates a subtype logic

$$\Sigma \vdash A \leq B$$

between types and typesums, which is defined in Figure 3. Following Java, we take function types to be non-variant in their argument and result types.

In the following, we want to restrict our attention to *well-formed* types, that satisfy each of the following conditions:

- Every free type variable is bound in Σ
- Every class name is bound in Δ .
- If a class has parameters, then the actual parameters are subtypes of the typebounds of the formal parameters.

It is straightforward to formalize these requirements in a **wf** predicate for types and typeschemes. For space reasons such a formalization is omitted here.

Phase 2: A class environment Δ is *well-formed*, if it satisfies the following conditions:

1. \leq is a partial order on ground types with a top element (i.e. **Object**).
2. For all ground types A, B , if A is well-formed and $\vdash A \leq B$ then B is well-formed.
3. Field or method declarations in different classes with the same name have the same type.

If the class environment is well-formed, the subtype relation over typesums is a complete upper semilattice:

Proposition 1 *Let Σ be a subtype environment, and let \mathcal{X} be a set of typesums. Then there is a least typesum $\sqcup \mathcal{X}$ such that*

$$\Sigma \vdash X' \leq \sqcup \mathcal{X} \quad \text{for all } X' \in \mathcal{X} .$$

$$\begin{array}{l}
\text{(Taut)} \quad \frac{x : U \in \Gamma}{\Sigma, \Gamma \vdash x : \mathbf{cpl}(U)} \\
\text{(var Elim)} \quad \frac{\Sigma, \Gamma \vdash E : \mathbf{var} A}{\Sigma, \Gamma \vdash E : \mathbf{cpl}(A)} \\
(\forall \text{ Elim}) \quad \frac{\Sigma, \Gamma \vdash E : \forall \overline{\alpha} \leq \overline{\mathcal{B}}. U}{\Sigma, \Gamma \vdash \overline{A} \leq \overline{\mathcal{B}}[\overline{\alpha} := \overline{A}] \quad \Sigma, \Gamma \vdash E : U[\overline{\alpha} := \overline{A}]} \\
(\exists \text{ Elim}) \quad \frac{\Sigma, \Gamma \vdash E : \exists \overline{\alpha} \leq \overline{\mathcal{B}}. X}{\Sigma, \overline{\alpha} \leq \overline{\mathcal{B}}, \Gamma \vdash E : X}
\end{array}$$

where

$$\begin{array}{l}
\mathbf{cpl}(\alpha) = \alpha \\
\mathbf{cpl}(C) = C \\
\mathbf{cpl}((\overline{\alpha}, C, \overline{A}) \rightarrow B) = \forall \beta \leq C. \mathbf{cpl}((\overline{\alpha}, \beta, \overline{A}) \rightarrow B) \\
\quad \text{where } \beta \text{ fresh} \\
\mathbf{cpl}((\overline{\alpha}) \rightarrow A) = \forall \Sigma. (\overline{\alpha}) \rightarrow B \\
\quad \text{where } \forall \Sigma. B = \mathbf{cpl}(A) \\
\mathbf{cpl}(\forall \Sigma. U) = \forall \Sigma. \mathbf{cpl}(U)
\end{array}$$

Figure 4: Second order rules.

Proof idea: Every typesum X has only finitely many super-types A . Let $\mathit{super}(X)$ be the set of super-types of X and take $\sqcup \mathcal{X} = \exists \alpha \leq (\bigcap_{X \in \mathcal{X}} \mathit{super}(X)). \alpha$.

The type checking problem for Pizza expressions can be reduced to the problem of finding a most general substitution that solves a set of subtype constraints. Let θ_1 and θ_2 be substitutions on a given set of type variables, V . We say θ_1 is more general than θ_2 if there is a substitution θ_3 such that $\alpha \theta_1 \leq \alpha \theta_2 \theta_3$, for all $\alpha \in V$.

Proposition 2 *Let Σ be a subtype environment, let \mathcal{C} be a system of subtype constraints $(X_i \leq X'_i)_{i=1, \dots, n}$, and let V be a set of type variables. Then either*

- *there is no substitution θ such that $\text{dom}(\theta) \subseteq V$ and $\Sigma \vdash \mathcal{C}\theta$, or*
- *there is a most general substitution θ such that $\text{dom}(\theta) \subseteq V$ and $\Sigma \vdash \mathcal{C}\theta$.*

Phase 3: In the following, we always assume that we have a well-formed class environment Δ . We start with the rule for typing variables, followed by rules for eliminating typeschemes and type sums in typing judgements. We then give typing rules for all remaining Mini-Pizza constructs: expressions, statements, member- and class-declarations.

Figure 4 presents typing rules for variables and elimination rules for quantified types. The type of a variable x is the *completion* of the declared type of x , which is recorded in the typohesis. Completion extends the range of an argument type C to all subtypes of C . Rule (**var Elim**) implements Java's implicit dereferencing of mutable variables. Rule (\forall Elim) is the standard quantifier elimination rule of F-bounded polymorphism. Finally, rule (\exists Elim) eliminates existential quantifiers by skolemisation.

$$\begin{array}{c}
\text{(Select)} \quad \frac{\Sigma, \Gamma \vdash E : A \quad \Sigma \vdash A \leq c \langle \overline{B} \rangle \quad c : \mathbf{class}(\overline{\alpha} \leq \overline{B}, \Gamma_c, C) \in \Delta \quad \overline{\alpha} \leq \overline{B}, \Gamma_c \vdash x : U}{\Sigma, \Gamma \vdash E.x : U[\overline{\alpha} := \overline{B}]} \\
\text{(Apply)} \quad \frac{\Sigma, \Gamma \vdash E : (\overline{A}) \rightarrow B \quad \Sigma, \Gamma \vdash \overline{E} : \overline{A}}{\Sigma, \Gamma \vdash E(\overline{E}) : B} \\
\text{(New)} \quad \Sigma, \Gamma \vdash \mathbf{new} \ c() : c \langle \overline{A} \rangle \\
\text{(Assign)} \quad \frac{\Sigma, \Gamma \vdash E_1 : \mathbf{var} \ A \quad \Sigma, \Gamma \vdash E_2 : X \quad \Sigma \vdash X \leq A}{\Sigma, \Gamma \vdash E_1 = E_2 : A} \\
\text{(Widen)} \quad \frac{\Sigma, \Gamma \vdash E : A \quad \Sigma \vdash A \leq c \langle \overline{B} \rangle}{\Sigma, \Gamma \vdash (c)E : c \langle \overline{B} \rangle} \\
\text{(Narrow)} \quad \frac{\Sigma, \Gamma \vdash E : A}{\Sigma, \Gamma \vdash (c)E : \sqcup \{X \mid X = \exists \Sigma'. c \langle \overline{A} \rangle, \Sigma \vdash X \leq A\}}
\end{array}$$

Figure 5: Typing rules for expressions.

Figure 5 presents typing rules for expressions. Most of these rules are straightforward; note in particular that the function application rule (*Apply*) is the standard Hindley/Milner rule without any widening of argument types. The two rules for typecasts are more subtle. When an expression E with static type A is cast to a class c then one of two situations must apply. Either c is the name of a superclass of A . In that case A is widened to a class c , possibly completed with parameters such that the resulting type is a supertype of A . Otherwise, c must be the name of a subclass of A . In that case, we narrow A to the largest (wrt \leq) typesum $X \leq A$ generated from class c . The typesum X might contain existential quantifiers.

Figure 6 presents typing rules for statements. The type of a statement is the least upper bound of the types of all expressions returned from that statement. By Proposition 1 the least upper bound always exists. If no expression is returned, then the type of the statement is arbitrary (that is, we assume that checking that every non-void procedure has a return statement is done elsewhere).

Figure 7 presents rules that determine whether a class- or member-declaration is well-formed. Obviously, all types written in a declaration must be well-formed. For variable declarations we require that the initializing expression must be a subtype of the variable's declared type. Analogously, for function declarations we require that the return type of a function body must be a subtype of the function's declared return type.

Finally, a class declaration K is well-formed if all of its member declarations are well-formed in a context consisting of K 's formal type parameters and K 's class hypothesis, plus appropriate bindings for the two standard identifiers **this** and **super**.

$$\begin{array}{c}
\text{(Expr)} \quad \frac{\Sigma, \Gamma \vdash E : A}{\Sigma, \Gamma \vdash E; : B} \\
\text{(Seq)} \quad \frac{\Sigma, \Gamma \vdash S_1 : X_1 \quad \Sigma, \Gamma \vdash S_2 : X_2}{\Sigma, \Gamma \vdash S_1 S_2 : X_1 \sqcup X_2} \\
\text{(Return)} \quad \frac{\Sigma, \Gamma \vdash E : A}{\Sigma, \Gamma \vdash \mathbf{return} \ E; : A} \\
\text{(Cond)} \quad \frac{\Sigma, \Gamma \vdash E_0 : \mathbf{boolean} \quad \Sigma, \Gamma \vdash S_1 : X_1 \quad \Sigma, \Gamma \vdash S_2 : X_2}{\Sigma, \Gamma \vdash \mathbf{if} \ (E_0) \ E_1 \ \mathbf{else} \ E_2 : X_1 \sqcup X_2}
\end{array}$$

Figure 6: Typing rules for statements.

$$\begin{array}{c}
\text{(Vardcl)} \quad \frac{\Sigma \vdash A \ \mathbf{wf} \quad \Sigma, \Gamma \vdash E : X \quad \Sigma \vdash X \leq A}{\Sigma, \Gamma \vdash A \ x = E; \ \mathbf{wf}} \\
\text{(Fundcl)} \quad \frac{\Sigma \vdash \overline{A} \ \mathbf{wf} \quad \Sigma \vdash B \ \mathbf{wf} \quad \Sigma \vdash X \leq B \quad \Sigma, \Sigma', \Gamma, \overline{y} : \overline{A} \vdash S : X}{\Sigma, \Gamma \vdash \langle S \rangle B \ x(\overline{A} \ \overline{y}) : \{S\} \ \mathbf{wf}} \\
\text{(Classdcl)} \quad \frac{c : \mathbf{class}(\overline{\alpha} \leq \overline{B}, \Gamma, C) \in \Delta \quad \overline{\alpha} \leq \overline{B}, \Gamma, \mathbf{this} : c \langle \overline{\alpha} \rangle, \mathbf{super} : C \vdash \overline{M} \ \mathbf{wf}}{\vdash \mathbf{class} \ c \langle \overline{\alpha} \leq \overline{B} \rangle \ \mathbf{extends} \ C \ \{ \overline{M} \} \ \mathbf{wf}}
\end{array}$$

Figure 7: Typing rules for declarations.

References

- [AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proc. Functional Programming Languages and Computer Architecture*, pages 31–41. ACM, June 1993.
- [AG96] Ken Arnold and James Gosling. *The Java Programming Language*. Java Series, Sun Microsystems, 1996. ISBN 0-201-63455-4.
- [BTCGS91] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunther, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.
- [BCC⁺96] Kim Bruce, Luca Cardelli, Guiseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3), 1996.
- [BLM96] J. A. Bank, B. Liskov, and A. C. Myers. Parameterised types and Java. Technical Report MIT LCS TM-553, Laboratory for Computer Science, Massachusetts Institute of Technology, May 1996.

- [Bru97] Kim Bruce. Typing in object-oriented languages: Achieving expressibility and safety. *Computing Surveys*, to appear.
- [BW88] Richard S. Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice-Hall, 1988.
- [CCH⁺89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proc. Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.
- [CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.
- [DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proc. 9th ACM Symposium on Principles of Programming Languages*, January 1982.
- [ESTZ95] Jonathan Eifrig, Scott Smith, Valery Trifonov, Amy Zwarico. An interpretation of typed OOP in a language with state. *Lisp and Symbolic Computation* 8(4):357–397, December 1995.
- [GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Java Series, Sun Microsystems, 1996. ISBN 0-201-63451-1.
- [Jon93] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proc. Functional Programming Languages and Computer Architecture*, pages 52–61. ACM, June 1993.
- [Läu95] K. Läufer. A framework for higher-order functions in C++. In *Proc. Conf. Object-Oriented Technologies (COOTS)*, Monterey, CA, June 1995. USENIX.
- [LO94] Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Trans. on Programming Languages and Systems*, 16(5): 1411–1430, 1994.
- [LM91] Xavier Leroy and Michel Mauny. Dynamics in ML. *Proc. Functional Programming Languages and Computer Architecture*, pages 406–426. Springer-Verlag, August 1991. Lecture Notes in Computer Science 523.
- [MP88] J. Mitchell and G. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988.
- [Oho92] Atsushi Ohori. A compilation method for ML-style polymorphic record calculi. In *Proc. 19th ACM Symposium on Principles of Programming Languages*, pages 154–165, January 1992.
- [OP95] Martin Odersky and Michael Philippsen. EspressoGrinder distribution. <http://wwwipd.ira.uka.de/~espresso>.
- [Pau91] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991. ISBN 0-521-39022-2.
- [SS76] Guy Steele and Gerald Jay Sussman. Lambda: The ultimate imperative. AI Memo 353, MIT AI Lab, March 1976.
- [Str91] Bjarne Stroustrup. *The C++ Programming Language, Second Edition*. Addison-Wesley, 1991.
- [Sun96a] Sun Microsystems. Inner classes in Java. <http://java.sun.com/products/JDK/1.1/designspecs/innerclasses>.
- [Sun96b] Sun Microsystems. Java core reflection. <http://java.sun.com/products/JDK/1.1/designspecs/reflection>.
- [WB89] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, pages 60–76, January 1989.
- [Wir77] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Comm. ACM*, 20, pages 822–823, November 1977.