Support for Functional Programming in Brew

Gerald Baumgartner, Martin Jansche, and Christopher D. Peisert

Dept. of Computer and Information Science
The Ohio State University
395 Dreese Lab., 2015 Neil Ave.
Columbus, OH 43210–1277, USA
`{gb,jansche,peisert}@cis.ohio-state.edu`

Abstract. Object-oriented and functional programming languages differ with respect to the types of problems that can be expressed naturally. We argue that a programming language should provide support for both programming styles so that the best style can be chosen for a given problem. We present the language support for functional programming in Brew, a successor language of Java we are currently developing. The salient features are closure object and multimethod dispatch, together with syntax for function types and function definitions. We demonstrate that these features can be smoothly integrated with the object-oriented features of the language and outline their implementation.

1 Introduction

Most mainstream object-oriented languages, including Java [9] and C++ [20], feature classes, objects as class instances, visibility constraints, and subtyping through a run-time dispatch on a designated receiver argument. These features make it easy to encapsulate implementation details, to abstract over implementation types, and to refine existing data structures by adding new subclasses.

However, object-oriented languages do not provide good support for abstracting over control, and single dispatch makes it difficult to define new operations on an existing data structure. The object-oriented style of defining a method at the root of the hierarchy and overriding it in subclasses requires classes to be modified when adding a new method. The Visitor pattern [8] simplifies adding new operations, but it must be anticipated when designing the class hierarchy and it precludes further refinements of the data structure.

By contrast, statically typed functional languages, such as ML [19, 18] and Haskell [10], make it easy to abstract over control through higher-order functions and to define new operations on an existing data structure.

Functional languages, however, do not provide adequate support for encapsulation and abstraction over implementation types and make it difficult to refine existing data structures. Adding a new variant to an existing data type requires all functions operating on this data type to be modified. As a consequence, ML must treat the exception type `exn` specially to allow new exception constructors to be added.

Many programming problems are more naturally solved in either a functional or in an object-oriented language and may be awkward to solve in a language from the other family. Also, the way in which data structures evolve over time, by adding new variants or new operations, is dictated by the application. It would be desirable if both sets of language mechanisms were provided in the same programming language.

The need for functional language mechanisms in object-oriented languages is demonstrated by the variety of design patterns and C++ frameworks for implementing closures. The design patterns Strategy, State, and Command [8] work around the lack of closures by encapsulating functional behavior in classes with a single public method. Läufer designed a framework for implementing higher order functions in C++ [13]. The FC++ library [16, 15] builds on Läufer's framework and implements a large part of the Haskell Standard Prelude by using templates in C++. FACT! [23] and the Lambda Library [11] use the preprocessor and templates to add lambda expression syntax to C++.

Because of the lack of templates, these approaches would not work for Java; it is necessary to modify the language. On the other hand, language support is desirable since it simplifies the syntax and provides better type-checking. The Pizza language [21] extends Java with syntax for function types and anonymous functions as well as templates. However their syntax for function types does not fit well into the language and, like the C++ library approaches, they provide no mechanism for defining functions by cases as in ML or Haskell. Also, like functional languages, the functional subsets of these approaches do not allow the data structure to be refined without modifications to existing code.

We are currently developing Brew as a successor language to Java. In previous research, we have analyzed object-oriented design patterns for gaining insight into what language features would be needed in better object-oriented languages [2]. Since the solutions of design patterns are influenced by the chosen implementation language, an analysis of the solutions indicates possible improvements to the language. We are designing Brew based on Java syntax but with an object model derived from this analysis of design patterns. In addition to the support for functional programming, Brew will provide a separation of subtyping from code reuse and a representation of classes as first-class objects.

In this paper, we present how object closures and multimethods allow support for functional programming to be included seamlessly in an object-oriented language. The notion of multimethods presented here is from Half & Half [1], an extension of Java (and predecessor of Brew) with retroactive abstraction and multimethods. We do not yet provide support for parametric polymorphism, but are planning to add it once the proposed extension of Java with generics [3] stabilizes.

Section 2 discusses in more detail why support for functional programming is desirable. The functional aspects of Brew's design are discussed in Section 3. Section 4 demonstrates the support for functional programming in Brew through examples. Following is an outline of the implementation in Section 5, and Section 6 provides conclusions.

2 Motivation

2.1 Functions

The class construct is designed for creating multiple instances of a class that contain state. Several design patterns employ classes for different purposes, though. E.g., the Singleton pattern [8] ensures that only a single instance of a class gets created by hiding the constructor and by letting the class maintain its own single instance. The complexity of the Singleton pattern suggests that language support for defining a singleton object may be desirable [2]. We propose to use an object construct of the form

```
object O implements I {
    int x = 42;
    int foo(int i) { ... }
}
```

Like instances of a class, singleton objects can be assigned to interface references or passed to methods and can be defined by inheritance. Singleton objects can be considered instances of class `Object` but do not have a class as implementation type.

The design patterns Strategy, State, and Command [8] employ the classes for encapsulating behavior, typically a single method without state. Again, since this is not the intended use of the class construct, and since these design patterns are fairly common and complex, this suggests that a language construct for defining functions outside of classes may be desirable.

Function would also be desirable for abstracting over behavior. For example, the higher-order function `map` applies a functional argument to every element of a list and returns the list of results. Similarly, `foldr` applies a binary function successively to all the elements of a list:

```
fun map f nil = nil
  | map f (h::t) = (f h) :: (map f t)

fun foldr f b nil = b
  | foldr f b (h::t) = f (h, foldr f b t)
```

While iterators in an object-oriented language can be used for traversing a collection data structure such as a list and performing an operation on every list element, they are not as flexible and as succinct as higher-order functions.

2.2 Closures

In addition to a function construct, it would be desirable to have static scoping and a closure mechanism that allows functions to capture their statically enclosing environment at the time the function is defined. For example, using static scoping and the above function `foldr`, the cross product of two lists can be computed as follows:

```
fun crossProduct (l1, l2) =
  foldr (fn (x, p) => foldr (fn (y, q) => (x, y) :: q) p l2)
        []
        l1
```

For a given x, the inner foldr starts with the list of pairs p and successively adds the pairs (x, y) for all y in l2. The parameter q of the anonymous function is used for accumulating the resulting list of pairs. The outer foldr starts with the empty list and calls the inner foldr for each x in l2 to add all the pairs with first element x to its accumulator p. It is difficult to express such algorithms as succinctly without the use of closures and higher-order functions.

Closures also allow functions to be defined by currying functions with higher arity. The function add below takes an integer x as argument and returns a function that adds x to its argument. This way add3 can be defined as a unary function that adds 3 to its argument.

```
fun add x =
  fn y => x + y

val add3 = add 3
```

Finally, closures allow the definition of infinite and lazy data structures. For example, a lazy list or stream can be defined as either the empty stream Nil or as a stream Cons(h,f) consisting of a first element h and a function f that computes the rest of the stream on demand:

```
datatype 'a Stream = Nil
                   | Cons of 'a * (unit -> 'a Stream)
```

The infinite list even of all even integers can then be defined using a helper function evenFrom that constructs the stream:

```
fun evenFrom n = Cons (n, fn () => evenFrom (n + 2))

val even = evenFrom 0
```

The elements of this stream only get constructed when trying to access them. E.g., using the function take,

```
fun take (Nil, n) = nil
  | take (Cons (h, t), n) =
      if n = 0 then nil else h :: (take (t (), n-1))
```

the call take (even, 5) constructs the first five even integers and returns them as the list [0,2,4,6,8].

A more elaborate example that uses streams to test whether two trees have the same fringe, i.e., the same leaves in a pre-order traversal, is used as a motivating example for Läufer's C++ framework [13] and for FC++ [15].

Unlike Haskell, we do not advocate making lazy evaluation the default evaluation mechanism, since it can result in unpredictable memory usage. If lazy data structures are needed, they are straightforward to implement using closures.

2.3 Multimethods

Another useful feature of modern functional languages is that they allow functions to be defined by an enumeration of cases. E.g., the ML definitions of `map` and `foldr` above both contain one case for the empty list and one case for a non-empty list. Instead of a parameter name, a parameter pattern can be specified that is matched at run time against the argument. This programming style makes function definitions more readable and more succinct.

However, the semantics of pattern matching in ML would be undesirable for inclusion in an object-oriented language. If multiple patterns are applicable for matching against a given argument, ML chooses the case in textual order. E.g., in the factorial function

```
fun fac 0 = 1
  | fac n = n * fac (n - 1)
```

both cases would be applicable if the argument 0 is passed, but the first one will be selected. This sequential evaluation order has the disadvantage that functions operating on a data structure must be modified when a new variant is added to the data structure.

If the cases defining a function were disjoint and could be textually separated, then adding a variant to a data structure would only require adding a new case to the function definition without modifying existing code. This semantics can be captured very well with multimethods, except that multimethods only allow dispatching on an argument type instead of matching the argument against a pattern.

Multimethods provide run-time dispatch on multiple arguments, which allows for flexibility both in extending the type hierarchy and in extending the operations on the type hierarchy. This simplifies designs for which the Visitor pattern [8] was originally intended, as it allows operations on an existing type hierarchy to be added as multimethods without changing the existing type hierarchy.

For example, suppose we are given the following type hierarchy for arithmetic expressions:

```
abstract class Exp { }
class IntLiteral extends Exp { ... }
class PlusExp    extends Exp { ... }
```

we would like to define an evaluation function as a list of cases as follows:

```
int eval(IntLiteral x) { ... }
int eval(PlusExp x)    { ... }
```

When calling `eval` on an argument of type `Exp`, the appropriate case should be selected at run time based on the dynamic type of the argument. This is in contrast to overloading, which selects a method based on the static types of the arguments.

It should be possible, to add functions, e.g., a `print` function, without modifying the data structure. It should also be possible to add new variants to the data structure, e.g., a class `MinusExp`, together with a method `int eval(MinusExp)` without modifying the existing code of `eval`. Of course, it should be possible to type-check multimethods statically.

This problem is known as the *extensibility problem* or as the *expression problem* [22, 6, 12, 24, 7] A more detailed description of how this problem can be solved using multimethods can be found in [1].

The presence of both multiple dispatch and closures would allow us to add generic traversals, such as `map` or `fold` functions, to existing type hierarchies in a functional style.

3 Language Design

3.1 Singleton Objects

For declaring a singleton object that is not an instance of a class Brew uses an object construct that is similar to a Java class declaration, except that the new keyword `object` is used instead of `class` and that `static` constructs (including constructors) are not permitted. If an initializer is provided (a single block in the body of the object declaration), it is run exactly once immediately after the object has been created. The following example is a declaration that binds an object with members `counter` and `count` to the identifier `O`. This binding cannot be changed while the name `O` is in scope (i.e., it is implicitly `final` in Java terminology):

```
object O implements Serializable {
    private int counter = 0;
    int count() { return counter++; }
}
```

The object construct will be refined later in this section, but in its current form it is already useful since it makes the implementation of the Singleton pattern trivial.

3.2 Functions

We use syntax for function types that follows the C-family convention of putting the result type in front of a parenthesized list of argument types. Here are two example types:

```
int (float)             // 1
int (float) (int, int)  // 2
```

On the first line is the type of a function with a parameter of type `float` that returns a result of type `int`. The second example is the type of a higher-order function that takes two `int` arguments and returns a function from `float` to `int`. This is the reverse of the notation used in the ML-family, where the types in this example would be written as follows:

```
float -> int                  (* 1 *)
(int * int) -> float -> int   (* 2 *)
```

Function types are treated like interface types, where each interface defines a method named with the keyword apply — analogous to operator() in C++— of the appropriate type. The types shown above are equivalent to the following explicit interface declarations:

```
interface int(float) { int apply(float); }                 // 1
interface int(float)(int) { int(float) apply(int, int); }  // 2
```

The converse is true, too: an object implementing any interface containing a method called apply can be assigned to a variable of the corresponding function type. This allows programmers to declare type synonyms:

```
interface StringOp extends String(String) {}
```

A variable of a function type can be assigned either (i) a static method qualified by the name of the class or singleton object that contains it; or (ii) the partial application of a non-static method to an object (passed to the implicit this argument of that method); or — the most general case — (iii) a singleton object or class instance that implements the function type:

```
int (float) f = Math.round;            // (i)
char (int) g = "Hello, world!".charAt;  // (ii)
String () h = anObject.toString;       // (ii)

object O implements int(float) {
    int apply(float x) {
        return Math.round(x);
    }
}
f = O.apply;                           // (i)
f = O;                                 // (iii)
```

We introduce syntactic sugar to make case (iii) more palatable. Brew provides the programmer with the illusion that all functions are objects of an appropriate interface type (though the compiler may choose to represent them differently as we will discuss in Section 5). The declaration of object O in the last example can be written equivalently as

```
int O(float x) {
    return Math.round(x);
}
```

An anonymous function (lambda abstraction) simply omits the function name:

```
int (float x) { return Math.round(x); }
```

Higher-order functions, especially operations on homogeneous collections, in a statically typed functional language usually go together with a powerful type system that at the very least allows for parametric polymorphism. So far, we did not include genericity in our design, but we are planning to adopt the proposed genericity model for Java [3] once it stabilizes and to extend it to our function syntax.

3.3 Closures

For providing closures, we simply allow objects to be arbitrarily nested and let them capture their lexically enclosing environment. For example, if an object is defined inside a method and returned by the method, the object still has access to the local variables and the parameters of the method after the method returns. Unlike with inner classes in Java, we also allow objects to access non-local variables of a built-in type. Because functions are treated as objects, function closures are simply a special case of object closures.

As a simple example, consider currying of addition. The following code uses syntactic sugar for functions, including an anonymous function:

```
int(int) add(int x) {
    return
        int (int y) { return x+y; };    // anonymous function
}
int answer = add(11)(31);
```

Recall that this is equivalent to the following piece of code, which uses object syntax instead of function syntax and introduces a type synonym:

```
interface Adder {
    int apply(int y);
}

object add {
    Adder apply(int x) {
        object addx implements Adder {  // object closure
            public int apply(int y) {
                return x+y;             // capture non-local x
            }
        }
        return addx;
    }
}
int answer = add.apply(11).apply(31);
```

3.4 Multiple dispatch

An abstract method in a base class provides a uniform interface to all concrete implementations in derived classes. For run-time dispatch based on additional arguments we need a similar notion in order to provide a single entry-point that then dispatches through to the appropriate special case. This is commonly referred to as the *generic function*, which is implemented by one or more *multimethods.* A generic function is declared via a function header modified by the keyword `generic`. Its multimethods must share its name and provide implementations for all possible combinations of arguments passed to the generic function. At run time, dispatch proceeds in two steps: single-argument dispatch on the designated receiver argument is carried out first, followed by symmetric dispatch on the remaining arguments, which selects the most specific multimethod for the run-time types of the arguments.

As an example, consider the following class hierarchy for implementing lists. Since we do not yet have support for templates, list elements pointed to by the heads of `Cons` nodes are declared to be of type `Object`.

```
abstract class List {}
class Nil extends List {}
class Cons extends List {
    private Object hd;
    private List   tl;
    Cons(Object h, List t) { hd = h; tl = t; }
    Object getHd() { return hd; }
    List   getTl() { return tl; }
}
```

Multimethods allow us to define functions operating on this data structure in a similar style as in a functional language.

```
object ListOps {
    generic List append(List, List);  // generic function
    List append(Nil l1, List l2)  {   // first multimethod
        return l2;
    }
    List append(Cons l1, List l2) {   // second multimethod
        return new Cons(l1.getHd(), append(l1.getTl(), l2));
    }
}
```

For guaranteeing run-time type safety, the compiler must ensure that for any combination of arguments of the generic function there is exactly one most-specific applicable multimethod. For allowing this type-check to be performed statically, previous approaches to multimethods restricted the parameter types of generic functions to be class types [5, 17].

However, since function types are interface types, we need to allow interface types as parameter types of generic functions and of multimethods so that higher-order functions can be defined by cases. For avoiding a global type-check we need to introduce constraints on the visibility of these types (for details see [1]). A sufficient, but slightly too restrictive, condition is to demand that no interface type that appears as parameter type of a generic function or a multimethod can be `public`. A non-public interface type is visible only within its package, hence the set of its subtypes can be determined by package-level program analysis, which, depending on the compilation model, can happen at compile time or at link-time.

What this effectively means is that non-public interface types are enumerated types, much like ML datatypes. This has been used in the above example, which does not contain a multimethod to deal with the case where the first argument has run-time type `List`, but is neither a subtype of `Nil` nor of `Cons`, because `List` is partitioned into `Nil` and `Cons` and cannot be extended outside its package.

4 Further Examples

4.1 Higher-Order Functions

Suppose we implemented lists as a class hierarchy with an abstract superclass `List` and two subclasses `Cons` and `Nil` as above.

Using the proposed syntax for function types and function definitions it is straightforward to define the higher-order functions `map` and `foldr` as multi-methods on the `List` hierarchy.

```
generic List map(Object(Object), List);
List map(Object(Object) f, Nil l)  { return l; }
List map(Object(Object) f, Cons l) {
    return new Cons(f(l.getHd()), map(f, l.getTl()));
}

generic Object foldr(Object(Object, Object), Object, List);
Object foldr(Object(Object, Object) f, Object b, Nil l)  {
    return b;
}
Object foldr(Object(Object, Object) f, Object b, Cons l) {
    return f(l.getHd(), foldr(f, b, l.getTl()));
}
```

Using `foldr`, our syntax for anonymous functions, and static scoping, the `crossProduct` function can be defined as follows:

```
List crossProduct(List l1, List l2) {
    return
        foldr(List (Object x, List p) {
                  return
                      foldr(List (Object y, List q) {
                                return
                                    new Cons(new Pair(x,y), q);
                            }, p, l2);
              }, new Nil(), l1);
}
```

4.2 Internal Iterators

Since any object with an `apply` method can be used as a function, we can write accumulator objects that maintain state across function calls. For example, given an integer list class `IntList` with an internal iterator method `void foreach(void(int))`, we can sum the elements of the list by passing the following object `add` as argument to `foreach`:

```
IntList il;
object add implements void(int) {
    private int total = 0;
    public void apply(int x) { total += x; }
    public int getTotal() { return total; }
}
il.foreach(add);
int sum = add.getTotal();
```

In the absence of templates and, therefore, parametric polymorphic higher-order functions, such internal iterators can be used for type-safe iteration over the elements of a collection class.

4.3 The Expression Problem

Suppose we are given the following class hierarchy for arithmetic expressions:

```
abstract class Exp { }
class IntLiteral extends Exp {
    int value;
    // etc.
}
class PlusExp extends Exp {
    Exp left;
    Exp right;
    // etc.
}
```

It is straightforward to add operations on the data structure without modifications to existing code by defining these operations as multimethods. For example, we might add a function for evaluating expression trees:

```
object Evaluator {
    generic int eval(Exp x);
    int eval(IntLiteral x) {
        return x.value;
    }
    int eval(PlusExp x) {
        return eval(x.left) + eval(x.right);
    }
}
```

If later we extend the data structure by adding a new subclass

```
class MinusExp extends Exp {
    Exp left;
    Exp right;
    // etc.
}
```

we do not need to modify the existing operations on the data structure. We can simply extend these operations through inheritance:

```
object ExtendedEvaluator extends Evaluator {
    int eval(MinusExp x) {
        return eval(x.left) - eval(x.right);
    }
}
```

5 Implementation

Work on the implementation of a Brew compiler is in progress. The compilation process can best be conceptualized as a translation of Brew into Java. This idea is borrowed from Pizza [21], and in fact Pizza's approach for implementing function closures will be extended to object closures.

For the `object` construct the compiler constructs a Java class with an appropriately mangled name and ensures that exactly one instance of that class is created. For closure objects that have access to the local variables of the lexically surrounding method, the compiler creates an additional field in the object for each method variable that the object accesses.

If the object does not access any variables in the enclosing method, it can be allocated outside the method. Under the same conditions a nested function is translated to a Java method, which is more efficient if the function is never assigned to a variable of function type.

In case a method or a Brew function represented as a method is assigned to a variable of function type (or if assignment conversion from a method to a function type is required in function calls, returns, or casts), the method must first be wrapped by an adapter object. This is similar to the generation of wrappers for implementing structural subtyping [14]. For example, the following piece of Brew code

```
int (float) f = Math.round;
```

is, conceptually, first transformed into

```
object _o42 implements int(float) {
    public int apply(float x) { return Math.round(x); }
}
int (float) f = _o42;
```

and ends up in Java as

```
interface brew_funtype_17 {
    int apply(float x);
}
class brew_obj_o42 implements brew_funtype_17 {
    public static instance = new brew_obj_o42;
    public int apply(float x) { return Math.round(x); }
}
brew_funtype_17 f = brew_obj_o42.instance;
```

For implementing multiple dispatch, multimethods are translated into protected methods with a mangled name. A generic function is then translated into an if-then-else chain that tests the argument types using `instanceof` and dispatches to one of the appropriate multimethods. The optimal if-then-else chain is generated using the algorithm by Chambers and Chen [4]. Further details on our multimethod implementation can be found in [1].

6 Conclusions

We have described the language support for functional programming in the programming language Brew. We are developing Brew as a successor language of Java. The support for functional programming, consisting of syntax for function types and function definitions, closures, and multimethods, is seamlessly integrated with the object-oriented features of the language.

Brew's object model is being designed based on an analysis of design patterns [2] and will feature support for retroactive abstraction over existing type hierarchies [14], a separation of subtyping from code reuse through inheritance, closure objects, multimethods [1], and an object representation of classes. In this paper, we have highlighted closure objects and multimethod dispatch and have demonstrated using examples that this combination provides good support for functional programming. By adopting the proposed extension of Java with generics [3] once it stabilizes, we will also be able to support parametric polymorphism for functions.

References

[1] Gerald Baumgartner, Martin Jansche, and Konstantin Läufer. Half & Half: Multiple dispatch and retroactive abstraction for Java. Technical Report OSU-CIRC-5/01-TR08, Department of Computer and Information Sciences, The Ohio State University, May 2001.

[2] Gerald Baumgartner, Konstantin Läufer, and Vincent F. Russo. On the interaction of object-oriented design patterns and programming languages. Technical Report CSD-TR-96-020, Department of Computer Sciences, Purdue University, February 1996.

[3] Gilad Bracha, Norman Cohen, Christian Kemper, Steve Marx, Martin Odersky, Sven-Eric Panitz, David Stoutamire, Kresten Thorup, and Philip Wadler. Adding generics to the Java programming language: Participant draft specification. Draft for Public Review JSR-000014, Sun Microsystems, Inc., May 2001.

[4] Craig Chambers and Weimin Chen. Efficient multiple and predicate dispatching. In *Proceedings of the OOPSLA '99 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 238–255. Association for Computing Machinery, October 1999. *ACM SIGPLAN Notices*, 34(10), October 1999.

[5] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the OOPSLA '00 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 130–145, Minneapolis, Minnesota, 15–19 October 2000. Association for Computing Machinery.

[6] William R. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489, pages 151–178. Springer-Verlag, New York, NY, 1991.

[7] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1), pages 94–104, 1998.

[8] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1995.

[9] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, Massachusetts, 2nd edition, 2000.

[10] Paul Hudak (ed.), Simon Peyton Jones (ed.), Philip Wadler (ed.), Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell: A non-strict, purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27(5):Section R, May 1992.

[11] Jaakko Järvi and Gary Powell. The Lambda Library: Lambda abstraction in C++. TUCS Technical Report No. 378, Turku Center for Computer Science, University of Turku, Turku, Finnland, November 2000.

[12] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *ECOOP*, pages 91–113, 1998.

[13] Konstantin Läufer. A framework for higher-order functions in C++. In *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 103–116, Monterey, California, 26–29 June 1995. USENIX Association.

[14] Konstantin Läufer, Gerald Baumgartner, and Vincent F. Russo. Safe structural conformance for Java. *Computer Journal*, 43, 2001. In press.

[15] Brian McNamara and Yannis Smaragdakis. Functional programmming in C++. In *Proceedings of the International Conference on Functional Programming*, pages 118–129, Montreal, Canada, 18–21 September 2000. Association for Computing Machinery.

[16] Brian McNamara and Yannis Smaragdakis. Functional programming in C++ using the FC++ library. *ACM SIGPLAN Notices*, 36(4):25–30, April 2001.

[17] Todd D. Millstein and Craig Chambers. Modular statically typed multimethods. In *Proceedings of the 1999 European Conference for Object-Oriented Programming (ECOOP '99)*, volume 1628 of *Lecture Notes in Computer Science*, pages 279–303, Lisbon, Portugal, 14–18 June 1999. Springer-Verlag.

[18] Robin Milner and Mads Tofte. *Commentary on Standard ML*. The MIT Press, Cambridge, Massachusetts, 1991.

[19] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Massachusetts, 1990.

[20] National Committee for Information Technology Standards. *International Standard 14882 — Programming Language C++*. American National Standards Institute, 1998.

[21] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, 15–17 January 1997. Association for Computing Machinery.

[22] J. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Stephen A. Schuman, editor, *New Directions in Algorithmic Languages*, pages 157–168. Institut de Recherche d'Informatique et d'Automatique, Le Chesnay, France, 1975.

[23] Jörg Striegnitz. FACT! — Multiparadigm programming with C++. `http://www.kfa-juelich.de/zam/FACT/start/`.

[24] Philip Wadler. The expression problem. Posted to the Java-Genericity Mailing List, 12 November 1999.