Extensible Algebraic Datatypes with Defaults

Matthias Zenger
Swiss Federal Institute of Technology
INR Ecublens
1015 Lausanne, Switzerland
matthias.zenger@epfl.ch

Martin Odersky
Swiss Federal Institute of Technology
INR Ecublens
1015 Lausanne, Switzerland
martin.odersky@epfl.ch

ABSTRACT

A major problem for writing extensible software arises when recursively defined datatypes and operations on these types have to be extended simultaneously without modifying existing code. This paper introduces Extensible Algebraic Datatypes with Defaults which promote a simple programming pattern to solve this well known problem. We show that it is possible to encode extensible algebraic datatypes in an object-oriented language, using a new design pattern for extensible visitors. Extensible algebraic datatypes have been successfully applied in the implementation of an extensible Java compiler. Our technique allows for the reuse of existing components in compiler extensions without the need for any adaptations.

1. INTRODUCTION

1.1 Extensibility Problem

The extensibility problem has been extensively studied [5, 6, 10, 11, 16, 19, 22, 30]. It can be paraphrased as follows: Given a recursively defined set of data and operations on them, how can one add both new data variants and new operations on variants without changing or duplicating existing code? This is not only an academic question. Extending a system by modifying source code is an error-prone task. Furthermore, creating an extended system by duplicating source code also results in duplicated maintenance costs, if the old and the new system are both going to be used. In some cases, it is not even possible to apply source code modifications, because some parts of the system might only be available in binary form.

For example, consider a representation of lambda terms as trees, with variants `Lambda`, `Apply` and `Variable` and an `eval` operation on those trees. As one possible extension, consider adding variants `Number` and `Plus` to the term type. As another extension consider adding an operation that prints a term.

ICFP'01, September 3-5, 2001, Florence, Italy.

The traditional object-oriented and functional approaches both make extensions in one dimension easy, but extensions in the other dimension very hard. In the object-oriented approach, data is modelled by a set of classes, sharing a common interface. For the lambda term example, there would be an interface or abstract class `Term` specifying the `eval` method with subclasses `Lambda`, `Apply` and `Variable`. Each subclass defines its own implementation of `eval`. Whereas extending the datatype with new variants is simply done by creating new classes, adding new operations involves modifications of the abstract base class.

On the other hand, in the functional approach, the variants of a datatype are typically implemented as an algebraic type. Here, defining new operations is easy. One just writes a new function which matches against the data variants. But since ordinary algebraic datatypes cannot be extended without modifications to the source code, it would not be possible to add new variants.

Each of the two approaches can encode the other. In one direction, object-oriented languages can model the functional approach using the *Visitor* design pattern [14]. In the other direction, objects can be represented in functional languages as closures taking an algebraic datatype of messages as parameter. However, each of these encodings exchanges both the strengths and weaknesses of one approach with the strengths and the weaknesses of the other; neither encoding gains simultaneous extensibility of both data and operations.

1.2 Extensibility by Subclassing

We can make the object-oriented approach extensible without modifying source code if we allow type casts. For adding a new operation, we have to extend the common variant interface with the new operation. In our lambda example, one would define an extension `ExtendedTerm` of interface `Term` that adds a `print` method. As a consequence, all three variants have to be subclassed to provide an implementation for the new operation. Whenever we want to invoke the new operation, we first have to cast the receiver to the extended type. For instance to invoke the `print` method of a variable `t` of type `Term` one uses `((ExtendedTerm)t).print()`. Otherwise we would not be able to access the new operation. This approach supports extensibility of both data and functions. But adding a new function is very tedious, since it requires that all variants have to be subclassed.

Krishnamurthi, Felleisen and Friedman show that we can apply a similar coding scheme to the functional approach to make it also support variant extensions [19]. They describe the composite design pattern *Extensible Visitor* that keeps visitors open for later extensions. Whenever a new variant class is added, all existing visitors have to be subclassed in order to support this new variant. Otherwise a runtime error will appear as soon as an old visitor is applied to a new variant. Again, variants and operations are extensible. But this time, adding a new variant requires all visitors to be subclassed. Since the extensible visitor pattern is rather complex and error-prone to implement by hand, a special syntax is proposed for specifying extensible visitors. A preprocessor translates these specifications into an object-oriented programming language. A pattern comparable to the one of Krishnamurthi, Felleisen and Friedman is described by Gagnon and Hendren [13]. They explain how to add new variants and operations, but the extension or reuse of operations is not discussed.

In summary, subclassing techniques can provide extensibility in the previously missing dimension, but they rely on the use of type casts and require possibly extensive adaptation code.

1.3 Extensibility with Default Cases

In practice, it appears quite often that an operation defines a specific behaviour only for some variants and all other variants are subsumed by a default treatment. Such an operation could be reused without modifications for an extended type, if all new variants are properly treated by the existing default behaviour. In fact, our experience with the implementation of an extensible Java compiler shows that at least for this sort of application, the majority of the existing operations can be reused "as is" for extended types (Section 5.3 presents statistics). In this case, extending a system with the techniques of Section 1.2 would be very cumbersome, since for most cases we would just have to map an operation for all new variants to the default behaviour.

If we would be able to specify a default case for every function operating on an extensible type, a function would have to be adapted only in those situations where new variants require a specific treatment. This technique would improve "as is" code reuse significantly. Previously mentioned approaches were not able to handle default cases since datatype and function definitions were tightly coupled: one of the two always specified the interface of its partner completely.

In this paper we present a solution to the extensibility problem which is based on the new notion of extensible algebraic datatypes with defaults. We describe these extensible algebraic types in the context of an object-oriented language. The approach presented in this paper smoothly combines object-oriented extensibility through subclassing and overriding with type-safe pattern matching on algebraic datatypes, known from functional programming languages. Our work was inspired by the algebraic types offered by Pizza [24], a superset of Java [18].

From an extensible algebraic datatype one can derive extended types with new variants in addition to the ones defined in the original type. These types enable us to solve the extensibility problem in a functional fashion; i.e. the definition of the datatype and operations on that type are strictly separated. Extensions on the operation side are therefore completely orthogonal to extensions of the datatype.

In addition to adding new variants and operations, we also support extending existing variants of a datatype and modifying existing operations. Furthermore, applying existing operations to new variants is possible, since operations for extensible algebraic types define a default case. Extensibility is achieved without the need for modifying or recompiling the original program code or existing clients.

We show that it is possible to encode programs using extensible algebraic types in languages with just objects and subtyping such as Java. The encoding takes the form of a new design pattern for extensible visitors with default cases. Since this pattern is rather difficult to implement by hand, we decided to include direct support for extensible algebraic types in an extension of Java.

Extensible algebraic types have been used heavily in the design and implementation of our extensible Java compiler *JaCo*. JaCo has been used in several projects to rapidly implement language extensions for Java. The implementation work in these projects showed that the notational convenience afforded by default cases was very important. Statistics taken from several compiler extensions indicate that on average three quarters of all functions operating on algebraic types could be reused on extended types without any modifications. Extensible algebraic datatypes allowed us to reuse these functions without having to add adaptation code.

The rest of this paper is organized as follows. Section 2 presents a simple programming protocol that explains how extensible algebraic datatypes with defaults solve the extensibility problem. Section 3 discusses various aspects of extensible algebraic datatypes in more detail. Section 4 gives an encoding of extensible algebraic datatypes with visitors. Section 5 discusses the experience we gained from using extensible algebraic types in the design and implementation of an extensible Java compiler and presents statistics about the level of code reuse in this system. Related work is reviewed in Section 6. Section 7 concludes.

2. EXTENSIBILITY WITH ALGEBRAIC DATATYPES

A typical example where the extensibility problem plays a significant role is the implementation of extensible interpreters and compilers. A wrong design limits extensibility or complicates at least the task of extending the system. In this section we use algebraic datatypes to derive a programming pattern for writing extensible interpreters. We start with a small language consisting of variables, lambda abstractions and lambda applications.

For declaring algebraic types we use the syntax introduced by the programming language Pizza [24]. Here is an algebraic datatype defining abstract syntax tree nodes for our example language:

```
class Term {
    case Variable(String name);
    case Apply(Term fn, Term arg);
    case Lambda(String name, Term body);
}
```

The algebraic type Term declares constructors Variable, Lambda and Apply for the three language constructs. We now define a simple interpreter that evaluates terms based on a call-by-value evaluation strategy with dynamic scoping. Our Interpreter class contains a single method eval that implements the operation for evaluating a term. Pattern matching is used to distinguish the different variants of the Term type in the eval method. Pizza uses switch statements to perform pattern matching for objects of an algebraic type.

```
class Interpreter {
   Term eval(Term term, Env env) {
      switch (term) {
         case Variable(String n):
            return env.lookup(n);
         case Apply(Term fn, Term arg):
            switch (eval(fn, env)) {
               case Lambda(String n, Term body):
                  return eval(body,
                                 env.bind(n, eval(arg, env)));
               default:
                  throw new Error("function expected");
            }
         default:
            return term;
      }
   }
}
```

By using this approach, it is straightforward to add new operations over type Term to the interpreter simply by defining further methods. The following code adds a method that transforms a term into a string:

```
class Formatter {
   void toString(Term term) {
      switch (term) {
         case Variable(String n):
            return n;
         case Apply(Term fn, Term arg):
            return "(" + toString(fn) +
                       " " + toString(arg) + ")";
         case Lambda(String n, Term b):
            return "(" + n + "->" + toString(b) + ")";
         default:
            return "<unknown>";
      }
   }
}
```

We now come to the problem of adding new variants to the Term datatype. This paper proposes *extensible algebraic datatypes with defaults* to solve this problem. These types enable us to define a new algebraic datatype by adding additional variants to an existing algebraic type. Here is the declaration of an extended Term datatype, which defines two new variants Number and Plus:

```
class ExtendedTerm extends Term {
   case Number(int val);
   case Plus(ExtendedTerm left, ExtendedTerm right);
}
```

With this definition, we introduce a new algebraic datatype ExtendedTerm consisting of five constructors Variable, Apply, Lambda, Number and Plus. One can think of an extensible algebraic datatype as an algebraic type with an implicit default case. Extending an extensible algebraic type means refining this default case with new variants. For the example above, the new type ExtendedTerm inherits all variants from Term and defines two additional ones. With our refinement notion, these two new variants are subsumed by the implicit default case of Term. The next section shows that this notion turns ExtendedTerm into a subtype of Term. This subtype relationship is crucial for code reuse, since it allows us to apply all functions over the original type to terms containing nodes from the extended type. Since the existing functions perform a pattern matching only over the original variants, an extended variant is handled by the default clause of the switch statement.

For our current eval method, the default clause simply returns the same node, so numbers and additions would not get evaluated by the eval operation. To support evaluation of the new variants, we have to adapt our interpreter accordingly. We do this by subclassing the Interpreter class and overriding the eval method.

```
class ExtendedInterpreter extends Interpreter {
   Term eval(Term term, Env env) {
      switch (term) {
         case Plus(ExtendedTerm left, ExtendedTerm right):
            return ExtendedTerm.Number(
               evalNum(left, env) + evalNum(right, env));
         default:
            return super.eval(term, env);
      }
   }
   int evalNum(ExtendedTerm term, Env env) {
      switch (eval(term, env)) {
         case Number(int val): return val;
         default: throw new Error("number expected");
      }
   }
}
```

The example shows that we only have to provide an implementation for the Plus variant. For all the other variants, we delegate the method call to the overridden method. Even our freshly introduced Numbers are handled properly by the overridden method.

In the new interpreter we make use of an additional method evalNum which projects Number terms to integers and throws an exception if the given term is not a Number.

These code fragments demonstrate the expressiveness of extensible algebraic datatypes in the context of an object-oriented language like Java. We are able to extend datatypes and operations in a completely independent and uniform way. An extension in one dimension does not enforce any adaptations of the other dimension. Since in pattern matching statements new variants are simply subsumed by the default clause, existing operations can be reused for extended datatypes. Operations are defined locally in a single place. The conventional object-oriented approach would distribute a function definition over several classes, making it very difficult to understand the operation as a whole.

Our approach supports a modular organization of datatypes and operations with an orthogonal extensibility mechanism. With the technique presented in this section, extended interpreters are derived out of existing ones simply by subclassing. Only the differences have to be implemented in subclasses. The rest is reused from the original system, which itself is not touched at all. Roudier and Ichisugi refer to this form of software development as *programming by difference* [29].

3. PRINCIPLES OF EXTENSIBLE ALGEBRAIC DATATYPES

In this section, we review the type theoretic intuitions behind extensible algebraic datatypes with defaults. Usually, algebraic types are treated as sum types of variants. Classical sum types can be straightforwardly extended by adding new variants. However, such an extension yields a subtype relation which is the reverse of the extension relation, i.e. extensions become supertypes of the original type. In the following subsection, we review these concepts, and argue that the induced subtyping relation is not useful for writing extensible software. We then refine in Section 3.2 our model

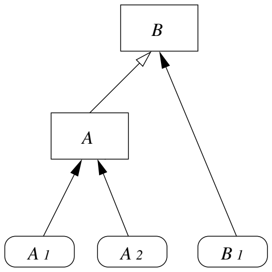

Figure 1: Subtyping for extensible sums

of algebraic types to include default cases. This has the effect of reversing the original subtype relation, bringing it in sync with the extension relation.

3.1 Extensible Sums

Algebraic datatypes can be modelled as sums of variants. Each variant constitutes a new type, which is given by a tag and a tuple of component types. For instance, consider the declaration:

```
class A {
   case A1(T1,1 x1,1, ... , T1,r1 x1,r1);
   case A2(T2,1 x2,1, ... , T2,r2 x2,r2);
}
```

This defines a sum type A consisting of two variant types A_1 and A_2, which have components $T_{1,1}\ x_{1,1}, \ldots, T_{1,r_1}\ x_{1,r_1}$ and $\mathsf{T}_{2,1}\ \mathsf{x}_{2,1}, \ldots, \mathsf{T}_{2,r_2}\ \mathsf{x}_{2,r_2}$, respectively.

Let $allcases(A)$ denote the set of all variants of the algebraic type A. For example, $allcases(A) = \{A_1, A_2\}$.

To describe extensions of algebraic types, we introduce a partial order $\preceq$ between algebraic types. $B \preceq A$ holds if B extends A by adding new variants to it. A priori the algebraic extension relation $\preceq$ is independent of the subtyping relation.

In our setting $\preceq$ is defined explicitly by type declarations. For example, the following code defines an algebraic datatype $B \preceq A$ that extends A with an additional variant B_1:

```
class B extends A {
   case B1(...);
}
```

The new type B is described by the set of its own variants $owncases(B) = \{B_1\}$ and the inherited variants $allcases(A)$. Thus, for the extended algebraic type B, we get $allcases(B) = allcases(A) \cup owncases(B) = \{A_1, A_2, B_1\}$.

The standard typing rules for sum types [3] make A a subtype of B if all variants of A are also variants of B. In our example, we have $allcases(A) \subseteq allcases(B)$, so our original type A is a subtype of the extended type B. Figure 1 summarizes the relationships between types. In this figure, algebraic datatypes are depicted as boxes, variants are displayed as round boxes. Arrows highlight subtype relationships. More specifically, outlined arrows represent algebraic type extensions, whereas all other arrows connect variants with the algebraic types to which they belong.

Unfortunately, the subtype relation between extensible sum types is often the opposite of what one would like to have in practice. Imagine we have the following Term type:

```
class Term {
   case Number(int val);
   case Plus(Term left, Term right);
}
```

Adding a new variant Ident would yield a new algebraic type ExtendedTerm.

```
class ExtendedTerm extends Term {
   case Ident(String name);
}
```

Since ExtendedTerm is a supertype of Term, we cannot represent the sum of two identifiers with the Plus variant. This variant expects two Terms as its arguments, but the variant Ident is not included in the Term type. In other words, extensible sums do not support open recursion in the definition of a datatype. So the classical way of describing algebraic types by a fixed set of variants does not provide extensibility in the way we need it.

3.2 Extensible Algebraic Types with Defaults

In order to turn extended types into subtypes, we have to keep the set of variants open for every extensible algebraic type. We achieve this by adding a default variant to every algebraic datatype, which subsumes all variants defined in future extensions of the type. The set of all variants of an extensible algebraic datatype is now given by the following equation.

$$allcases(Y) = inherited(Y) \cup owncases(Y) \cup default(Y)$$

$$\begin{array}{llcl} \text{where} & owncases(Y) & = & \bigcup_i \{Y_i\} \\ & inherited(Y) & = & \bigcup_{Y \preceq X, Y \neq X} owncases(X) \\ & default(Y) & = & \bigcup_{Z \preceq Y, Z \neq Y} owncases(Z) \end{array}$$

That is, every extensible algebraic type Y is defined by three disjoint variant sets $owncases(Y)$, $inherited(Y)$ and $default(Y)$. $inherited(Y)$ includes all inherited variants from the algebraic type Y is extending, $owncases(Y)$ denotes Y's new cases, and $default(Y)$ subsumes variants of future extensions.

With this understanding, our variant sets for types A and B from Section 3.1 now look like this: $allcases(A) = \{A_1, A_2\} \cup default(A)$, and $allcases(B) = \{A_1, A_2, B_1\} \cup default(B)$. Since $default(A)$ captures B_1 as well as $default(B)$, $\{B_1\} \cup default(B)$ is a subset of $default(A)$. Therefore $allcases(B) \subseteq allcases(A)$ and B is a subtype of A.

One might be tempted to believe now that one has even $allcases(A) = allcases(B)$. This would identify types A and B. But a closer look at the definition of $default$ reveals that $default(B)$ only subsumes variants of extensions of B. Variants of any other extension of A are contained in $default(A)$, but not covered by $default(B)$. This is illustrated by the following algebraic class declaration:

```
class C extends A {
   case C1(...);
}
```

C is another extension of algebraic type A, which is completely orthogonal to B. It's case C_1 is not included in $default(B)$, but is an element of $default(A)$. As a consequence, $\{B_1\} \cup default(B)$ is a proper subset of $default(A)$, and therefore the extended type B is a proper subtype of

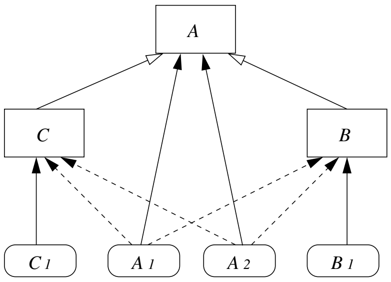

Figure 2: Subtyping for alternative extensions of algebraic types

A. C is a proper subtype of A for the same reasons, but the types B and C are incompatible.

The subtype relationships of our example are illustrated in Figure 2. Again, boxes represent extensible algebraic datatypes, round boxes represent variants. Subtype relationships are depicted with arrows. Extending an algebraic datatype means creating a new type which is a subtype of the old one and which inherits all the variants of the old one. Furthermore this new type may also define additional variants. Dashed arrows connect inherited variants with the algebraic type to which they get inherited.

With our approach, extended algebraic types are subtypes of the types they extend. Therefore existing functions can be applied to values of extended types. New variants are simply subsumed by the default clause of every pattern matching construct. Another interesting observation can be made when looking at two different extensions of a single algebraic type (like B and C in the example above). They are incompatible; neither of them is a supertype of the other one. This separation of different extensions is a direct consequence of single-inheritance: an extensible algebraic type can only extend a single other algebraic datatype.

Extending the same type more than once yields extended algebraic types that share some variants, but that are incompatible to each other. Of course, it is also possible to extend an extension of an algebraic type further:

```
class D extends B {
    case D1(...);
}
```

Here, the algebraic type D extends B and defines an additional variant D_1. Figure 3 shows the resulting subtype relations.

3.3 Compilation of Extensible Algebraic Types with Defaults

The previous section pointed out that extensible algebraic datatypes with defaults are subject to single-inheritance; i.e. an algebraic type can only extend a single other algebraic datatype. This restriction enables an efficient compilation of pattern matching for extensible algebraic types using the conventional technique for sealed datatypes. The conventional compilation scheme assigns unique tags to every variant of an algebraic type. Pattern matching can then simply be implemented with a switch over all tags.

For extensible algebraic datatypes with defaults, variants are tagged with subsequent numbers starting with the number which is equivalent to the number of inherited variants.

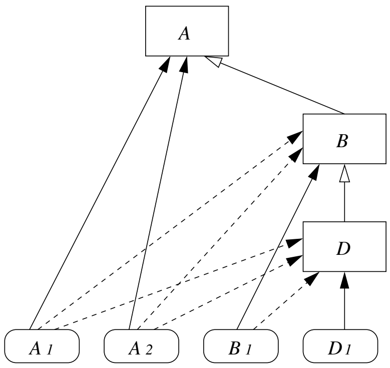

Figure 3: Subtyping for linear extensions of algebraic types

By doing this, we never tag two variants of any algebraic type with the same number. But we observe that two different extensions of an algebraic type (like B and C in Section 3.2) may share the same tags. Since two extensions of a single algebraic datatype always yield incompatible types, this issue usually does not cause any problems. Only pattern matching statements that mix variants of two incompatible extensions have to be split up in two separate pattern matching statements together with a dynamic typecheck that selects one of the two statements. We implemented an efficient version of this compilation scheme for our extensible Java compiler JaCo [33].

4. VISITOR ENCODING

In object-oriented languages, algebraic dataypes can be encoded with the visitor design pattern [14]. Krishnamurthi, Felleisen and Friedman extended this design pattern to enable extensibility [19]. In this section, we take their approach even further by adding support for default cases. Our programming pattern models extensible algebraic types in a purely object-oriented language (with some small differences which are discussed at the end of this section). We start with a review of the standard visitor pattern. All code fragments of this section are written in Java.

4.1 Design Pattern *Visitor*

The visitor design pattern models algebraic types with abstract classes. The variants of an algebraic type are represented by subclasses which define the fields of the variant and a corresponding constructor. Operations are encapsulated in visitor objects. A visitor object contains an overloaded method visit for every variant, which implements the operation specifically for this variant. To be able to apply a visitor, every variant defines an accept method. These methods take a visitor and invoke the appropriate visitor method. Here is a visitor framework for the example from Section 2. We only show the code for variant Variable. The other variants of the Term type are encoded analogously.

```
interface Visitor {
    void visit(Variable term);
    void visit(Apply term);
    void visit(Lambda term);
}
```

```
abstract class Term {
    abstract void accept(Visitor v);
}
class Variable extends Term {
    String name;
    Variable(String name) { this.name = name; }
    void accept(Visitor v) { v.visit(this); }
}
...
```

Concrete operations implement the visitor interface and provide a mechanism for passing arguments and returning results. Various solutions are possible here. We present a technique where for every visitor invocation a new visitor object gets created. The visitor constructor stores the arguments in fields inside the visitor. There is also a public field res in which the result of an operation is stored. The following implementation of Eval uses an anonymous visitor in the visit(Apply term) method for applying a lambda abstraction to an argument. Since we refer to the formal parameter term from inside the anonymous class, Java requires us to mark this parameter as final.

```
class Eval implements Visitor {
    private Env env;
    public Term res;
    Eval(Env env) { this.env = env; }
    void visit(Variable term) {
        res = term;
    }
    void visit(Lambda term) {
        res = term;
    }
    void visit(final Apply term) {
        Eval eval = new Eval(env);
        term.fn.accept(eval);
        eval.res.accept(new Visitor() {
            void visit(Variable t) { throw new Error(); }
            void visit(Apply t) { throw new Error(); }
            void visit(Lambda t) {
                Eval evArg=new Eval(env);
                term.arg.accept(evArg);
                Eval evFun=new Eval(env.bind(t.x, evArg.res));
                t.body.accept(evFun);
                res = evFun.res;
            }
        });
    }
}
```

The Visitor interface describes all the datatype's variants. Since this interface is fixed, it this not possible to add new variants. On the other side, adding new operations is easy by creating a new visitor implementation.

4.2 Extensible Visitors with Default Cases

We now show how to derive an extensible visitor design pattern with default cases. Operations with default cases can be applied to extended algebraic datatypes. They provide a generic treatment for variants of any future datatype extension. In our visitor framework we model default cases with an additional visit method in the Visitor interface.

```
interface Visitor {
    void visit(Term term);
    void visit(Variable term);
    void visit(Apply term);
    void visit(Lambda term);
}
```

In the code fragments of this section, new code is highlighted in italics.

The Term class now defines a generic implementation of the method accept which calls the default case of the visitor. Like in the standard visitor pattern, variants override this method by calling their own visit method.

```
abstract class Term {
    void accept(Visitor v) { v.visit(this); }
}
class Variable extends Term {
    String name;
    Variable(String name) { this.name = name; }
    void accept(Visitor v) { v.visit(this); }
}
...
```

Note that even though the accept methods of class Term and Variable are syntactically identical, a different visit method is called.

So far, except for the additional default case of visitors and the default implementation of the accept method, setting up the extensible visitor framework was identical to the standard pattern.

Now, we look into the implementation of concrete visitors. Since we want visitors to be extensible, it should be possible to subclass a visitor in order to override existing methods or to add further visit methods for new variants. In the standard visitor pattern, we created a new visitor for every visitor application. For example, in the code above, we implemented a recursive call of the visitor Eval with the following two lines:

```
Eval eval = new Eval(env);
term.fn.accept(eval);
```

For extensible visitors, this hard-coded visitor creation does not work anymore, since it does not consider the fact that we might use an extension of Eval. We use instead a scheme first proposed by Krishnamurti, Felleisen and Friedman [19]: Every visitor object has to provide factory methods [14] for creating all non-anonymous visitors that are being used in the visitor object. Extensions are supposed to override these factory methods.

Here is an extensible visitor that implements our Eval operation. Compared to the version from the previous section, we only added the default visit method and a factory method newEval for creating instances of Eval. This factory method gets invoked in the recursive operation call of the visit method for lambda applications.

```
class Eval implements Visitor {
    protected Env env;
    public Term res;
    Eval(Env env) { this.env = env; }
    Eval newEval(Env env) { return new Eval(env); }
    void visit(Term term) { res = term; }
    void visit(Variable term) { res = term; }
    void visit(Lambda term) { res = term; }
    void visit(final Apply term) {
        Eval eval = newEval(env);
        term.fn.accept(eval);
        eval.res.accept(new Visitor() {
            void visit(Term t) {
                throw new Error("function expected");
            }
            void visit(Variable t) {
                throw new Error("function expected");
            }
            void visit(Apply t) {
                throw new Error("function expected");
            }
```

```
        void visit(Lambda t) {
            Eval evArg=newEval(env);
            term.arg.accept(evArg);
            Eval evFun=newEval(env.bind(t.x, evArg.res));
            t.body.accept(evFun);
            res = evFun.res;
        }
    });
  }
}
```

Next, we are going to extend the system by adding new variants Number and Plus. We do this by first creating an abstract intermediate class XTerm which extends the old Term class. The new variant classes are direct subclasses of XTerm. With this approach, we have a type XTerm which subsumes our new variants. Now we have to update the Visitor interface accordingly. The extended XVisitor interface inherits all methods from the original Visitor interface and defines two additional visit methods for the Number and Plus variant. The following code shows that there is also an extra default visit method for XTerms. We explain later how the default methods for Term and XTerm variants are going to be used. Before, we focus our attention again on class XTerm. Like class Term, this class defines an accept method, which, this time, takes an XVisitor and calls the corresponding default case. Note that this method does not override the inherited accept method. The new accept simply overloads the existing one. As before, the variants override the new accept method in order to call their own specialized visitor method. The following code fragment summarizes the framework we described. Except for the additional extends clauses, this code for extending an algebraic type is completely analogous to the one for defining a new algebraic type.

```
interface XVisitor extends Visitor {
  void visit(XTerm term);
  void visit(Number term);
  void visit(Plus term);
}
abstract class XTerm extends Term {
  void accept(XVisitor v) {
    v.visit(this);
  }
}
class Number extends XTerm {
  int value;
  Number(int value) { this.value = value; }
  void accept(XVisitor v) { v.visit(this); }
}
...
```

We quickly verify that it is possible to apply an existing visitor like Eval to a term structure containing variants of the extended type. Since XTerm and variants of XTerm do not override the accept method of Term, the default visit method of the visitor gets invoked as desired in this case.

The rest of this section provides the last missing piece in the puzzle by looking at extensions of existing visitor classes like Eval. Here, we have to make sure that the visit methods of extended variants are called, rather than the default method of the original visitor. As explained before, for variants of extended types we automatically get into the original default method, so why not refining the dispatch here in order to incorporate the visitor's new visit methods? We do this by overriding the default visit method in the extended visitor. The overriding method performs a new dispatch with a more refined Term type. For this purpose, we have to insert a type cast to XTerm that tells the typechecker that our class is going to extend the Eval visitor specifically for XTerms. We defer a discussion about this type cast until the full pattern is explained.

Since we override the default visit(Term term) method from class Visitor, we have to provide a new default visit(XTerm term) method to keep the visitor open for further extensions. In most cases, this new method will just refer to the former default method via super.visit.

```
class XEval extends Eval implements XVisitor {
  void visit(Term term) {
    ((XTerm)term).accept(this);
  }
  void visit(XTerm term) { super.visit(term); }
  void visit(Number term) { super.visit(term); }
  void visit(Plus term) { ... }
}
```

Let us trace back the different method calls for the following visitor application to verify that our extended visitor is behaving in the desired way.

```
Term term = new Plus(new Number(1), new Number(2));
term.accept(new XEval(env));
```

The initial accept invocation refers to the accept method of class Term, since the dynamic type of term is Plus which does not override this accept method. Method accept of class Term calls the visit(Term term) method of our visitor. Here, the argument term gets cast to XTerm and a new dispatch on accept is performed. With Java's strategy for finding the best overloaded method, we refer to method accept(XVisitor v) this time. But now, this method is overridden in Plus, the dynamic type of our term, and we automatically end up in the visit method for Plus. Figure 4 illustrates the full sequence of method calls for the given visitor application.

A full implementation of the extended Eval visitor is given below. We used a slightly different approach for implementing the evaluation of Plus variants, compared to the version with extensible algebraic datatypes in Section 2.

```
class XEval extends Eval implements XVisitor {
  XEval(Env env) { super(env); }
  Eval newEval(Env env) { return new XEval(env); }
  void visit(Term term) {
    ((XTerm)term).accept(this);
  }
  void visit(XTerm term) { super.visit(term); }
  void visit(Number term) { super.visit(term); }
  void visit(Plus term) {
    class EvalNum implements XVisitor {
      int res = 0;
      void visit(Term t) { throw new Error(); }
      ...
      void visit(XTerm t) { throw new Error(); }
      void visit(Plus t) { throw new Error(); }
      void visit(Number t) { res += t.value; }
    }
    EvalNum evalNum = new EvalNum();
    Eval eval = newEval(env);
    term.left.accept(eval); eval.res.accept(evalNum);
    eval = newEval(env);
    term.right.accept(eval); eval.res.accept(evalNum);
    res = new Number(evalNum.res);
  }
}
```

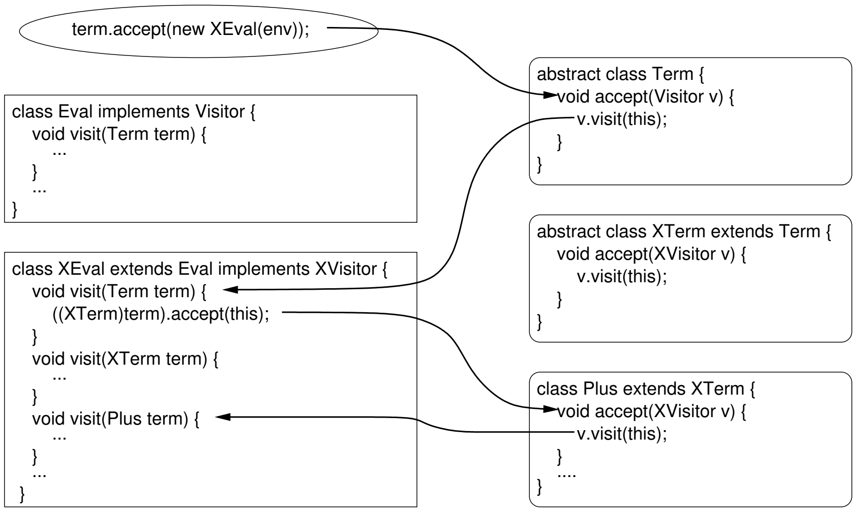

Figure 4: Sequence of method calls for a visitor invocation

With the protocol presented in this section, it is possible to implement extensible datatypes and visitors in an object-oriented language. Datatype extensions do not require any adaptations for existing visitors. They provide a default case for variants that are added later on. It is possible to extend existing visitors to modify the treatment of existing variants, but also to refine the default case for a new set of variants. This approach completely decouples datatype extensions from extensions of operations.

4.3 Extensible Algebraic Types in Comparison

In contrast to the source-level extensible algebraic types, the extensible visitor pattern opens the possibility of runtime errors because of the type cast in the extended concrete visitor class XEval. For programs adhering to the design pattern, this type cast only fails, if we try to apply an extended visitor to an incompatible datatype extension. For instance, if we would apply XEval to variants of another extension of Term, we would get a runtime error. XEval only handles variants of Term, XTerm, and any extension of XTerm. So, if we only extend a datatype linearly without introducing any branches, the type cast will never fail at runtime.

Pattern matching dispatch differs slightly between extensible algebraic types and extensible visitors. With the extensible visitor technique, the visit method for variants of Term is found after one double-dispatch. For variants of XTerm we need two double-dispatches until the proper visit method of the extended visitor is called. In general, for variants of the n-th datatype extension, we need $n+1$ double-dispatches.

Now, let us look back to the extensible algebraic datatypes. Here, we had an analogous situation. Every extension of an operation was implemented by overriding the old operation. The new operation matched against the new variants and forwarded the call to the old operation for the existing variants. So, for n extensions, we need in total $n+1$ single method dispatches and $n+1$ pattern matching statements. Only the order in which we dispatched on variants is now reversed. In the solution with extensible algebraic types, the variants from the latest extension are matched first. But since extensible algebraic datatypes can be used much more flexibly in practice, we do not need to observe a special implementation pattern, like the one presented in Section 2.

This is indeed one of the main weaknesses of the described design pattern. It is complicated to implement the full protocol by hand. Furthermore, even in the presence of anonymous local classes, visitors are rather heavy-weight constructs, imposing a lot of implementation work on the programmer.

For this reason, we decided to have direct language support for extensible algebraic datatypes in Java. This also allowed us to implement the more efficient pattern matching scheme that was briefly introduced in Section 3.3.

4.4 Extensibility Issues

Extensible algebraic datatypes as well as extensible visitors achieve simultaneous extensibility of types and operations. For both approaches, the main limitation is that we can extend a system only in a linear way, where each new extension explicitly refers to the previous one by name. Thus, it is not possible to merge orthogonal extensions.

The restriction to linear extensions is explicit in the extensible visitor approach, where a type cast fixes a visitor extension to be used only for a specific extension branch. The switch statement for extensible algebraic types does not have this restriction. It allows pattern matching against any variant of an extended type. For instance, in the following switch statement, we have a switch selector of type Term, but we can also match against variants of any extension of Term.

```
Term eval(Term term, Env env) {
   switch (term) {
      case Variable(String name): ...
      case Number(int val): ...
      default: ...
   }
}
```

This flexibility can be seen as an advantage or a disadvantage. One might regret the loss of type security, because it is now possible to apply the eval method to a datatype extension of Term for which it was not designed. But of course, one can model the behaviour of extensible visitors with the extensible algebraic type approach by introducing an explicit type cast.

```
Term eval(Term term, Env env) {
   switch ((ExtendedTerm)term) {
      ...
   }
}
```

In practice, we did not experience any problems with this issue. For the applications in which we used extensible algebraic types, namely extensible compilers and interpreters, we never happened to use two alternative extensions of a single algebraic datatype simultaneously. We also experienced that most extensions of our extensible compiler could not have been formulated in an orthogonal way. Therefore, the linearity restriction did not really limit the way we extended our system.

We emphasized already that in our approach, extensibility in one dimension does not require any adaptations in the other dimension due to the presence of default cases. This improves "as is" code reuse significantly, as Section 5.3 will show. On the other hand, the type system does not detect cases where an operation needs a refinement.

5. EXPERIENCE

5.1 Extensible Compilers

Originally, the work presented in this paper was motivated by a project which aimed at implementing an extensible Java compiler [33, 34]. At that time and still today, it was very popular to experiment with various language extensions of Java. Unfortunately, implementing these language extensions turned out to be a very difficult and time consuming task, since a suitable compiler infrastructure was missing. But even if a compiler prototype was available, implementation of new language features was rather done in an ad-hoc fashion by hacking a copy of the existing compiler. By doing this, the implementation of the new features and the original version get mixed. Thus, the extended compiler evolves into an independent system that has to be maintained separately.

Opposed to this destructive reuse of source code, we developed a framework where extended compilers reuse the components of their predecessors, and define new or extended components without touching any predecessor code. Extended compilers are derived out of existing ones simply by subclassing. Only the differences have to be implemented in subclasses. Thus, all extended compilers that are derived from an existing base compiler share the components of this base compiler. With this approach we created a compiler infrastructure which provides a basis for maintaining all compilers together.

One of the main difficulties of the implementation of this compiler was the representation of abstract syntax trees in an extensible fashion. The work with *EspressoGrinder* [23, 24, 27], a Java compiler with a rigorous object-oriented architecture, demonstrated the disadvantages of the classical object-oriented approach for implementing syntax trees and compiler phases. With an object-oriented approach, a compiler phase gets distributed over the whole code. This makes it very difficult to understand but also to extend. Furthermore, adding new passes is a pain, as explained in Section 1.1. Standard visitors do not provide an ideal solution either, because they do not allow extensibility for variants. They also tend to be rather heavy-weight syntactically.

Extensible algebraic datatypes offered what we needed: extensibility of the datatype itself, extensibility of operations and a very light-weight pattern matching construct which even supports nested patterns. They allowed us to apply a functional programming style in an object-oriented language. Object-oriented features were mainly used for providing extensibility by subclassing and overriding. Furthermore they were used to implement an extensible component framework for gluing the different components of the compiler together in a flexible and extensible manner.

5.2 JaCo: an Extensible Java Compiler

The implementation of our extensible Java compiler JaCo [32] makes extensive use of extensible algebraic datatypes. Since JaCo is a plain Java 2 compiler without support for extensible algebraic datatypes, it is not possible to compile JaCo with itself. Instead, JaCo is compiled with an extension of itself, supporting extensible algebraic types. This complicated the bootstrapping process quite a bit. The first version of JaCo was implemented in Pizza, which already offers algebraic types. We then modified the Pizza compiler to allow a very restricted form of extensible algebraic types. With this modified Pizza compiler we extended JaCo so that this extension supported our restricted form of extensible algebraic types introduced in Pizza. After this step, we were able to compile JaCo with an extension of itself and we finally implemented full support for extensible algebraic types as described in [34].

Throughout the last two years we utilized our extensible Java compiler successfully in various projects. Several language extensions have been implemented and are still being maintained. Among the implementations is a compiler for Java with synchronous active objects, proposed by Petitpierre [26]. Another extension introduces Büchi and Weck's compound types together with type aliases [2]. In addition, we added operator overloading to the Java programming language in the style proposed by Gosling [17]. Eugster, Guerraoui and Damm implemented a domain specific language extension supporting publish/subscribe primitives on top of JaCo [8]. A rather exotic extension of JaCo is an implementation of a small language based on join calculus [12]. It replaces the syntactic analyzer pass with a full compiler for join calculus that generates a Java syntax tree as output. This tree is then fed into the remaining Java compiler to generate Java bytecodes. In this extension, JaCo is basically used as a backend for a compiler of a language, which has nothing in common with Java.

During the implementation of the extensions mentioned before, we did not have to modify the base compiler a single time. Its architecture was open enough to support all sorts of extensions we needed so far. Changes of the base compiler were all related to minor modifications in the specification of the Java programming language or to bugs found in the compiler. These changes can usually be elaborated in such a way that binary compatibility of Java classfiles is not broken. As a consequence, all compilers derived from the base compiler benefit immediately from the changes, since they inherit them. Because of Java's late binding mechanism it is not even necessary to recompile derived compilers.

		JaCo	PiCo	CJavaC	SJavaC
1.	Lines of code	25590	5462	6972	2335
2.	Classes	134	48	58	34
3.	Algebraic types	5	2	3	1
4.	Algebraic type extensions	0	1	3	1
5.	Visitors	135	46	54	13
6.	Visitor extensions	4	17	39	11
7.	Visitors reused "as is"	–	118	96	124
8.	Visitors for extended types in JaCo	–	63	108	63
9.	Visitors for extended types	–	30	54	11
10.	Visitor extensions for extended types	–	14	39	10
11.	Visitors for extended types reused "as is"	–	49 (78%)	69 (64%)	53 (84%)

Table 1: Statistics for the extensible compiler *JaCo* and derived compilers

5.3 Code Reuse in JaCo Extensions

Table 1 shows some data regarding the base compiler *JaCo* and three compiler extensions: *PiCo*, written by Zenger [33], extends JaCo by adding extensible algebraic types, *CJavaC*, written by Zermatten [35], extends JaCo with compound types and type aliases, and *SJavaC*, written by Cavin [4], is the synchronous active objects compiler.

The first row in Table 1 shows the code size of the base compiler and all extensions (given in lines of code including comments and whitespaces). These data show that all extensions reuse large parts of the base compiler unchanged. The next three rows state the number of classes, algebraic types and algebraic type extensions of the four compilers under consideration.

The base compiler JaCo contains five algebraic types. These types are used to represent the abstract syntax, symbols, types and constants. Furthermore, the implementation of the backend uses *items* to enable delayed code generation [31]. Two of the three extensions of JaCo extend only the abstract syntax tree type. CJavaC also extends the symbol and the type representation.

The data presented in the following rows are supposed to give an indication how the different compiler extensions benefit from the use of extensible algebraic types. The row labelled *Visitors* gives the number of methods that contain at least one pattern matching construct. This analogy between visitors and methods is a relatively conservative approximation, because it counts methods with multiple `switch` statements as single visitors. The row labelled *Visitor extensions* shows the number of methods that override a method with a pattern matching construct in a supertype. The row labelled *Visitors reused "as is"* lists the number of visitors in the base compiler that are reused in an extension without being overridden. This number is for each extension the difference between the number of visitors in JaCo (135) and the number in row 6.

For instance, the PiCo extension defines 46 visitors, but only 17 of them override one of the 135 visitors of JaCo. So PiCo incorporates 118 visitors from the base compiler without changing them.

Most derived compilers extend only some of the algebraic types in JaCo, whereas others are left unchanged. To assess the usefulness of default cases in visitors, we need to disregard algebraic data types which are unchanged. Line 8 of Table 1 shows the number of visitors in the base compiler that match over types which are subsequently extended in the derived compiler. Line 9 shows the number of visitors in the derived compiler which match over extended types. Line 10 gives the number of methods that override a visitor method for an extended type. Line 11 gives the difference between lines 8 and 10, and therefore shows the number and percentage of visitors for extended types that are reused "as is". Note that this is a lower bound for code reuse since the data does not show whether a method counted in line 10 overrides a method in the base compiler (counted in line 8) or a method defined in the derived compiler (counted in line 9).

The data show that PiCo defines 30 visitors for the extended abstract syntax tree type, 14 of them override one of the 63 tree visitors of the base system. So more than 75% of the tree visitors are reused without refining their default case for the new variants. For CJavaC we do not have to refine the default case of more than 63% of visitors for extended types. For SJavaC, the numbers are even better. This compiler reuses 84% of the tree visitors from the base system without adapting them.

These statistics indicate that for our extensible compiler project, operations with default cases help to improve the "as is" code reuse enormously. Without default cases we would have to update every single operation if the corresponding datatype was extended.

6. RELATED WORK

For the language ML, several proposals have been made to support extensibility for algebraic types. With Garrigue's *polymorphic variants*, an algebraic type constructor does not belong to any algebraic datatype in particular [15]. So there is no need anymore to define an algebraic type before using a specific variant. Type inference is used to infer admissible variants according to their use. With polymorphic variants, a programming style like the one presented in this paper is possible, but results in less precise types. For functions that are defined by a pattern matching clause with a default case, it is possible to pass any existing variant — not necessarily one, the function was intended for. This is due to the fact that it is not possible to declare to which algebraic types a variant constructor belongs. In [16] Garrigue presents an alternative solution to the extensibility problem based on polymorphic variants. This solution has more precise types, but does not provide default cases for operations. It supports a modular organization of datatype extensions. But when combining two datatypes, it is necessary to redefine every function in order to forward the call to one of the previous implementations. OCaml supports polymorphic variants from version 3.00 on [20].

The proposal for ML2000 contains a generalization of SML's exception types [1]. These *extensible types* are introduced in order to tag objects to support runtime dispatch and type-safe downcasting. Therefore they can also be seen as an extensible form of algebraic types. An extensible type is described by an initial variant constructor. Extending this type means refining the variant. The extended variant is a subtype of the previous one. In a pattern matching construct it is possible to match against variants that have a common supertype, thus giving the programmer the ability to define a typecase facility. It is possible to implement our programming protocol with these types, but ML2000's extensible types provide a slightly weaker typing, since they do not support deriving extended datatypes from existing types. It is only possible to add new variants to an existing type by specializing a variant. So, a type like ExtendedTerm from Section 2 could not be formulated. ML2000's extensible types are a refinement of the *Object ML* design of Reppy and Riecke [28].

In the literature, extensibility of algebraic datatypes was mostly discussed in the context of building modular interpreters in functional programming languages. Existing approaches like [21] and [9] allow a restricted form of extensibility: algebraic types are extensible, but the final datatype has to be closed before being used. Furthermore, extensions of datatypes always require updates of all existing functions to support the new variants due to the lack of default cases in pattern matching constructs. On the other hand, these approaches support the combination of orthogonal extensions. Basically the same holds for *mixin modules* proposed by Duggan and Sourelis [7].

Open classes, proposed by Clifton, Leavens, Chambers and Millstein, offer extensibility with default cases [5] for the object-oriented approach. Open classes allow the user to add new methods to existing classes without modifying existing code and without breaking encapsulation properties. With open classes, a datatype is modelled by an abstract superclass, variants are concrete subclasses. New operations are specified as external top-level methods. The default case is given in the form of a method for the abstract superclass. If a specific behaviour for a variant has to be provided, this method has to be overridden for the variant. With open classes, we get extensibility for data and operations where extensions in one dimension do not require modifications of the other one. But in practice, open classes suffer from several drawbacks. Whereas a new operation is typically defined as an external top-level method in a single compilation unit, extending or modifying an existing operation can only be done by explicitly subclassing all affected variants and overriding the corresponding methods. This leads to an inconsistent distribution of code, making it very difficult to group related operations and to separate unrelated ones. Furthermore, extending or modifying an operation always entails extensions of the datatype. This restricts and complicates reuse. For instance, accessing an extended operation in one context and using the original operation in another one cannot be implemented in a straightforward way.

Languages equipped with multiple dispatch offer this functionality. For instance, *MultiJava* introduces multimethods for Java [5]. For type-safety, MultiJava requires "default implementations" for generic methods as well. Therefore, a programming protocol similar to ours could be used to solve the extensibility problem. On the other hand, multimethods do not support deep pattern matching, and they are syntactically more heavy-weight and less flexible to apply than our switch statements. In the context of Java it is furthermore difficult to compile multiple dispatch efficiently. Dispatch costs are linear with the number of methods that dynamically overload a method.

Palsberg and Jay's *Generic Visitor* design pattern offers a way to completely decouple datatype and function definitions [25]. Therefore, their generic visitors are very flexible to use and to extend. Since generic visitors rely on reflective capabilities of the underlying system, this approach lacks static type-safety and is subject to substantial runtime penalties.

7. CONCLUSION

We classified solutions of the extensibility problem according to the degree of possible code reuse. The plain object-oriented and functional solution allow extensibility of datatypes and operations only through source code modifications. With subclassing and by using type casts, extensions are possible without touching source code. But depending on the approach, an extension on the datatype side entails extensions of all existing operations or vice versa. A solution for the extensibility problem in which operations provide a default case that handles future extensions does not require adaptations of operations for new datatype variants. We introduced extensible algebraic datatypes that facilitate a simple programming protocol supporting extensibility with default cases. Extensible algebraic datatypes allow us to freely extend datatypes and operations simultaneously and independently of each other. We presented a novel design pattern for extensible visitors with default cases, showing that it is possible to encode a similar programming protocol in an object-oriented language. This pattern does not rely on additional language features, but is much more complicated to implement by hand.

Based on extensible algebraic datatypes, we designed and implemented an extensible Java compiler. For experimenting with programming language extensions, such an extensible compiler is essential for rapidly implementing language extensions. Extending this compiler does not require any source code modifications. Extended compilers evolve out of existing ones simply by subclassing. Since they share most components with their predecessors, our technique provides a basis for maintaining the systems together. Statistics show that extended compilers reuse large parts of the base compiler unchanged. An extended compiler reuses on the average 75% of the operations on extended algebraic types from the base compiler without any adaptations. This number shows that default cases contribute significantly to the level of "as is" code reuse. In the last two years our extensible compiler framework was used in various other projects to quickly implement new language extensions of Java.

Acknowledgments

Special thanks to Christoph Zenger and Michel Schinz for numerous helpful discussions. Furthermore we thank Stewart Itzstein, David Cavin, Stephane Zermatten, Yacine Saidji and Christian Damm. They implemented extensions of JaCo and provided feedback on the implementation.

8. REFERENCES

[1] A. Appel, L. Cardelli, K. Crary, K. Fisher, C. Gunter, R. Harper, X. Leroy, M. Lillibridge, D. B. MacQueen, J. Mitchell, G. Morrisett, J. H. Reppy, J. G. Riecke, Z. Shao, and C. A. Stone. Principles and preliminary design for ML2000, March 1999.

[2] M. Büchi and W. Weck. Compound types for Java. In *Proc. of OOPSLA'98*, pages 362–373, October 1998.

[3] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[4] D. Cavin. Synchronous Java compiler. Projet de semestre. École Polytechnique Fédérale de Lausanne, Switzerland, February 2000.

[5] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for java. In *Proceedings of OOPSLA 2000*, volume 35, pages 130–145, October 2000.

[6] W. R. Cook. Object-oriented programming versus abstract data types. In *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, 1990*, volume 489, pages 151–178. Springer-Verlag, New York, NY, 1991.

[7] D. Duggan and C. Sourelis. Mixin modules. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 262–273, Philadelphia, Pennsylvania, 24–26 May 1996.

[8] P. Eugster, R. Guerraoui, and C. Damm. On objects and events. In *Proceedings for OOPSLA 2001*, Tampa Bay, Florida, October 2001.

[9] R. B. Findler. Modular abstract interpreters. Unpublished manuscript, Carnegie Mellon University, June 1995.

[10] R. B. Findler and M. 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, 1999.

[11] M. Flatt. *Programming Languages for Reusable Software Components*. PhD thesis, Rice University, Department of Computer Science, June 1999.

[12] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, Jan. 1996.

[13] E. Gagnon and L. J. Hendren. SableCC – an object-oriented compiler framework. In *Proceedings of TOOLS 1998*, August 1998.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[15] J. Garrigue. Programming with polymorphic variants. In *ML Workshop*, September 1998.

[16] J. Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, Sasaguri, Japan, November 2000.

[17] J. Gosling. The evolution of numerical computing in Java. Sun Microsystems Laboratories. `http://java.sun.com/people/jag/FP.html`.

[18] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Java Series, Sun Microsystems, second edition, 2000. ISBN 0-201-31008-2.

[19] S. Krishnamurthi, M. Felleisen, and D. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *European Conference on Object-Oriented Programming*, pages 91–113, 1998.

[20] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system release 3.00, documentation and user's manual, April 2000.

[21] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Symposium on Principles of Programming Languages*, pages 333–343, January 1992.

[22] P. Mäenpää and K. Oksanen. Extensible algebraic datatypes through prototypes and subtyping. Unpublished, 2000.

[23] M. Odersky and M. Philippsen. EspressoGrinder distribution. http://wwwipd.ira.uka.de/∼espresso, Dec. 1995.

[24] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 146–159, January 1997.

[25] J. Palsberg and C. B. Jay. The essence of the visitor pattern. Technical Report 5, University of Technology, Sydney, 1997.

[26] C. Petitpierre. A case for synchronous objects in compound-bound architectures. Unpublished. École Polytechnique Fédérale de Lausanne, 2000.

[27] M. Philippsen and M. Zenger. JavaParty – transparent remote objects in Java. *Concurrency: Practice and experience*, 9(11):1225–1242, November 1998.

[28] J. Reppy and J. Riecke. Simple objects for Standard ML. In *Proc. of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 171–180, Philadelphia, Pennsylvania, 1996.

[29] Y. Roudier and Y. Ichisugi. Mixin composition strategies for the modular implementation of aspect weaving — the EPP preprocessor and it's module description language. In *Aspect Oriented Programming Workshop at ICSE'98*, April 1998.

[30] P. Wadler and et al. The expression problem. Discussion on the Java-Genericity mailing list, December 1998.

[31] N. Wirth. *Compiler Construction*. Addison-Wesley, 1996.

[32] M. Zenger. JaCo distribution. `http://lampwww.epfl.ch/jaco/`. University of South Australia, Adelaide, November 1998.

[33] M. Zenger. Erweiterbare Übersetzer. Master's thesis, University of Karlsruhe, August 1998.

[34] M. Zenger and M. Odersky. Implementing extensible compilers. In *Proceedings of the ECOOP 2001 Workshop on Multiparadigm Programming with Object-Oriented Languages*, pages 61–80, Budapest, Hungary, June 2001.

[35] S. Zermatten. Compound Types in Java. Projet de semestre. École Polytechnique Fédérale de Lausanne, Switzerland, June 2000. `http://lampwww.epfl.ch/jaco/cjava.html`