Implementing Extensible Compilers

Matthias Zenger and Martin Odersky

Swiss Federal Institute of Technology
INR Ecublens
1015 Lausanne, Switzerland

Abstract. New extensions to programming languages are constantly being proposed. But implementing these extensions usually turns out to be a very difficult and expensive task, since conventional compilers often lack extensibility and reusability. In this paper we present some fundamental techniques to implement extensible compilers in an object-oriented language. For being able to implement extensible compiler passes, we introduce an extensible form of algebraic datatypes. Our *extensible algebraic datatypes with defaults* yield a simple programming protocol for implementing extensible and reusable compiler passes in a functional style. We propose an architectural design pattern *Context-Component* which is specifically targeted towards building extensible, hierarchically composed systems. Our software architecture for extensible compilers combines the use of algebraic types, known from functional languages, with this object-oriented design pattern. We show that this approach enables us to extend existing compilers flexibly without modifying any source code. Our techniques have been successfully applied in the implementation of the extensible Java compiler *JaCo*.

1 Introduction

Traditionally, compilers are developed for a fixed programming language. As a consequence, extensibility and reusability are often considered to be unimportant properties. In practice this assumption does not hold. People constantly experiment with new language features. They extend programming languages and build compilers for them. Writing a compiler for such an extended language is usually done in an ad-hoc fashion: the new language features are hacked into a copy of an existing compiler. By doing this, the implementation of the new features and the original implementation get mixed. The extended compiler evolves into an independent system that has to be maintained separately.

To avoid this destructive reuse of source code, we propose a technique where extended compilers reuse components of their predecessors, and define new or extended components without touching any predecessor code. All extended compilers derived from an existing base compiler share the components of this base compiler. With this approach we have a basis for maintaining all systems together.

Before discussing details of our extensible compiler architecture, we look at the traditional organization of compilers. The abstract syntax tree is a recursive data structure on which the different compilation passes operate. Extending a source language normally involves extensions of the abstract syntax tree representation and the compilation passes; i.e. operations on the abstract syntax tree. Section 2 discusses the shortcomings of existing approaches to this issue. *Extensible algebraic types with defaults* are proposed in section 3 to solve the problem of extending the representation of the abstract syntax tree simultaneously to extending operations on this tree. Extensible algebraic datatypes with defaults enable us to reuse existing compiler passes in extended compilers "as is". Section 4 discusses extensible algebraic datatypes in more detail. A general architectural design pattern *Context-Component* is presented in section 5. This pattern can be used to build extensible component systems in a very flexible way. It is used in section 6 to define an extensible batch-sequential compiler architecture. We implemented an extensible Java compiler according to this architecture. This compiler was used in several projects as a basis for implementing language extensions for Java. We conclude this paper with a summary of the experience we gained by using this compiler.

2 Extensibility Problem

Traditionally, the compilation process is decomposed into a number of subsequent passes, where each pass is transforming the program from one internal representation to another one. These internal representations are implemented as abstract syntax trees. Compiler passes are operations that traverse the trees. An extension or modification of the compilers source language often requires both, extensibility of the datatype modelling the abstract syntax and the set of passes operating on this type. Furthermore it is often necessary to adapt existing passes. Flatt [12] calls this well-known problem of extending data and operations simultaneously the *extensibility problem* [6, 7, 11, 12, 15, 17, 28].

Unfortunately, neither a functional nor an object-oriented approach solves this problem in a satisfactory way. With an object-oriented language such a datatype would be implemented as a set of classes sharing a common interface. We call these classes *variants* of the datatype. Whereas extending the datatype is simply done by creating new variant classes supporting the common interface, adding new operations is tedious. New operations require extensions or modifications of all existing variants.

In a functional language, the variants of a datatype are typically implemented with an algebraic type. Ordinary algebraic datatypes cannot be extended, so it is not possible to add new variants. But on the other hand, writing new operations is simple, since operations are simply functions over this type. In object-oriented languages, the functional approach can be modelled using the Visitor design pattern [13].

2.1 Related Work

Several attempts to solve this problem are published. *Open Classes* tackle the shortcomings of the object-oriented approach in a pragmatic way [6]. They allow the user to add new methods to existing classes without modifying existing code and without breaking encapsulation properties. Open classes provide a clean solution to the extensibility problem, but in practice they still suffer from some drawbacks. Whereas a new operation is typically defined in a single compilation unit, modifying an operation can only be done by subclassing the affected variants and overriding the corresponding methods. This leads to an inconsistent distribution of code, making it almost impossible 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.

For functional programming languages, various proposals were made to support extensibility of algebraic datatypes. Among them, the most prominent ones are Garrigue's *polymorphic variants* [14] and the *extensible types* of the ML2000 proposal [1]. [31] compares both approaches with our work. Several papers discuss the extensibility of algebraic types in the context of building extensible interpreters in functional languages [19, 10, 8]. Due to lack of space we refer to [17] for a short discussion.

The literature also provides several modifications of the Visitor design pattern targeted towards extensibility. Krishnamurthi, Felleisen and Friedman introduce the composite design pattern *Extensible Visitor* [17]. Their programming protocol keeps visitors open for later extensions. One drawback of their solution is that whenever a new variant 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. Palsberg and Jay's *Generic Visitors* are more flexible to use and to extend with respect to this problem [21]. But generic visitors rely on reflective capabilities of the underlying system [21], causing severe runtime penalties. Kühne's *Translator* pattern relies on generic functions performing a double-dispatch on the given operation and datatype variant [18]. As with the solution of Krishnamurthi, Felleisen and Friedman, datatype extensions always entail adaptions of existing operations accordingly. Therefore Kühne proposes not to use the translator pattern in cases where datatypes are extended frequently.

2.2 Extensibility with Defaults

The fact that extra code is necessary to adapt an operation to new variants can be very annoying in practice. We made the observation that an operation often defines a specific behaviour only for some variants, whereas 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. The experience with our extensible Java compiler showed that for extended compilers, the majority of the existing operations can be reused "as is" for extended types, without the need for adapting them to new variants [31].

If it would be possible 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.

We present a solution to the extensibility problem based on the new notion of extensible algebraic types with defaults. We describe these extensible algebraic datatypes in the context of an object-oriented language, similar to Pizza's algebraic datatypes [20]. From an extensible algebraic type one can derive extended types by defining additional variants. Thus, we can 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 completely orthogonal to extensions of the datatype. It is possible to apply existing operations to new variants, since operations for extensible algebraic types define a default case. In addition to adding new variants and operations, we also support extending existing variants of a datatype and modifying existing operations by subclassing. Extensibility is achieved without the need for modifying or recompiling the original program code or existing clients.

3 Extensible Compiler Passes with Algebraic Datatypes

In this section we explain how to implement extensible compiler passes with algebraic datatypes in an object-oriented language, by looking at a small example language. This language simply consists of variables, lambda abstractions and lambda applications. We use the syntax introduced by Pizza [20] and implement abstract syntax trees based on the following algebraic type definition:

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

We now define a type checking pass for our small language. Pattern matching is used to distinguish the different variants of the Tree type in the process method.

```
class TypeChecker {
    Type process(Tree tree, Env env) {
        switch (tree) {
            case Variable(String n):
                return env.lookup(n).type;
            case Lambda(Variable x, Tree body):
                ...
            case Apply(Tree fn, Tree arg):
                Type funtype = process(fn, env);
                ...
            default:
                throw new Error();
        }
    } ...
}
```

By using this approach, it is straightforward to add new operations (passes) to the compiler simply by defining new methods. But it is also easy to modify an existing operation by overriding the corresponding method in a subclass.

```
class NewTypeChecker extends TypeChecker {
    Type process(Tree tree, Env env) {
        switch (tree) {
            case Lambda(Variable x, Tree body):
                ...
            default:
                return super.process(tree, env);
        }
    }
}
```

This class modifies the treatment of the Lambda variant and reuses the former definition for the other variants of the Tree type by delegating the call to the super method.

As we saw, extending the set of operations and modifying existing operations does not cause any problems. But what about extending the datatype? Since Pizza translates every variant V of an algebraic datatype A to a nested class $A.V$, extending variants is simply done by subclassing the variant class:

```
class NewLambda extends Tree.Lambda {
    Tree argtype;
    NewLambda(Variable x, Tree argtype, Tree body) {
        super(x, body);
        this.argtype = argtype;
    }
}
```

The only missing piece for solving the extensibility problem now consists in the extension of the Tree datatype with new variants. Pizza's algebraic types cannot be extended in that way. To overcome this problem, we propose *extensible algebraic datatypes with defaults.* They allow us to define a new algebraic datatype by adding additional variants to an existing type. Here is the definition of an extended Tree datatype, which adds two new variants Zero and Succ:

```
class ExtendedTree extends Tree {
    case Zero;
    case Succ(Tree expr);
}
```

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. In the example above, the new type ExtendedTree inherits all variants from Tree and defines two additional cases. With our refinement notion, these two new variants are subsumed by the implicit default case of Tree. The next section shows that exactly this notion turns ExtendedTree into a subtype of Tree. For being able to reuse existing operations on Tree, it is essential that ExtendedTree is a subtype of Tree. This allows us to apply the original type checking pass to an extended tree. Since the original type checker performs a pattern matching only over the original variants, an extended variant would be handled by the default-clause of the switch statement (which throws an Error exception in our example above). To handle the new variants correctly, we have to adapt our type checking pass accordingly by overriding the process method:

```
class ExtendedTypeChecker extends TypeChecker {
  Type process(Tree tree, Env env) {
    switch (tree) {
      case Zero:
        return IntType();
      case Succ(Tree expr):
        checkInt(process(expr, env));
        return IntType();
      default:
        return super.process(tree, env);
    }
  }
}
```

These code fragments demonstrate the expressiveness of extensible algebraic datatypes in the context of an object-oriented language like Java. Opposed to almost all approaches of section 2, we can extend datatypes and operations in a completely independent way. An extension in one dimension does not enforce any adaptations of the other one. Since in a pattern matching statement, new variants are simply subsumed by the default clause, existing operations can be reused for extended datatypes. Our approach supports a modular organization of datatypes and operations with an orthogonal extensibility mechanism. Extended compiler passes 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* [24]. Another advantage is that an operation for an algebraic datatype is 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.

4 Principles of Extensible Algebraic Datatypes

In this section, we briefly review the type theoretic intuitions behind extensible algebraic datatypes with defaults. A more detailed discussion about extensible algebraic datatypes with defaults can be found in [31].

We model algebraic datatypes 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 fields $T_{1,1}\, x_{1,1}, \ldots, T_{1,r_1}\, x_{1,r_1}$ and $\mathsf{T}_{2,1}\, \mathsf{x}_{2,1}, \ldots, \mathsf{T}_{2,r_2}\, \mathsf{x}_{2,r_2}$, respectively. The algebraic type A is characterized by the set of all its variants. Since we want our types to be extensible, we have to keep A's set of variants open. We achieve this by assuming a default variant, which subsumes all variants defined in future extensions of A. We will formalize this notion later.

Here is the definition of an algebraic type B which extends type A by defining a new variant B_1:

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

The new algebraic type B inherits all variants from A and defines an additional variant B_1. Since B itself is extensible, we also have to assume a default variant here to keep B's variant set open. Thus, an extensible algebraic datatype with defaults can be described by three variant sets: the set of inherited variants, the set of own variants and the default variants (capturing future variant extensions). To formalize this notion, we introduce a partial order $\preceq$ between algebraic types. $B \preceq A$ holds if B equals A or B extends A by defining additional variants. In our setting, $\preceq$ is defined explicitly by type declarations.

An algebraic type Y can now formally be described by the union of three disjoint variant sets $owncases(Y)$, $inherited(Y)$ and $default(Y)$.

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

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

$inherited(Y)$ includes all variants that get inherited from the type Y is extending, $owncases(Y)$ denotes Y's new cases, and $default(Y)$ subsumes variants of future extensions.

With this definition, our variant sets for types A and B look like this: $allcases(A) = \{A_1, A_2\} \cup default(A)$, and $allcases(B) = \{A_1, A_2, B_1\} \cup default(B)$. The standard typing rules for sum types turn A into a subtype of B if all variants of A are also variants of B [4]. In our example, we have $allcases(B) \subseteq allcases(A)$, so our original type A is a supertype of the extended type B.

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. Its 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 real subset of $default(A)$, and therefore the extended type B is a real subtype of A. C is a real subtype of A for the same reasons.

The subtype relationships of our example are illustrated in Figure 1. 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. Dashed arrows connect inherited variants with the algebraic types to which they get inherited.

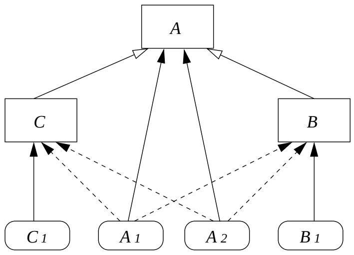


Fig. 1. Subtyping for extensions of algebraic types

With this 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 a 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; none 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.

5 Architectural Pattern: Context-Component

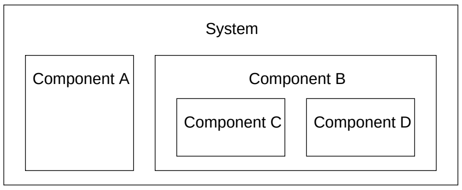
The technique described in section 3 enables us to implement extensible datatypes and extensible components offering functions operating on the datatypes. We have still no general mechanism to glue a certain combination of components and datatypes together to build extensible subsystems which are finally combined to a concrete compiler. Configuring a program by linking the corresponding components together should normally be done with a module system. Unfortunately many object-oriented languages do not provide a separate module system. They require the user to use the class system for this purpose. Even though a class is considered to be a bad substitute for a module in general [27], design patterns help to model the missing module system functionality at least for some specific applications.

Several design patterns for structuring a system are described in literature. The pattern *Whole-Part* [3] builds complex systems by combining subsystems with simpler functionality. A *Whole*-Object aggregates a number of simpler objects called *Parts* and uses their functionality to provide its own service. *Composite* [13] is a variant of *Whole-Part* with emphasis on uniform interfaces of simple and compound objects. A *Facade* [13] helps to provide a unified interface for a subsystem consisting of several interfaces, so that this subsystem can be used more easily. All these design patterns only target the structural decomposition of a system. They do not consider the fact that designs often require that the implementation of a component or a subsystem is not known to the clients at compile-time. For this reason, design patterns like *AbstractFactory* and *Builder* [13] have to be used in addition. They allow to configure instantiations at run-time. Therefore they are suitable for configuring a system dynamically, thus supporting extensibility in a flexible way.

In this section we describe the architectural design pattern *Context-Component* which helps to implement extensible hierarchically composed systems. It separates the composition of a system and its subsystems from the implementation of the components. This principle allows us to freely extend or reuse subsystems. Among all design patterns mentioned before, Context-Component is the only pattern that offers a uniform way to compose, to extend, to modify, and to reuse components and subsystems while still being easy to implement.

5.1 Idea

We suppose to implement a hierarchically structured component system. The following figure shows a system consisting of two components A and B. [1] Component B represents a more complex subsystem, referring to two local subcomponents C and D.

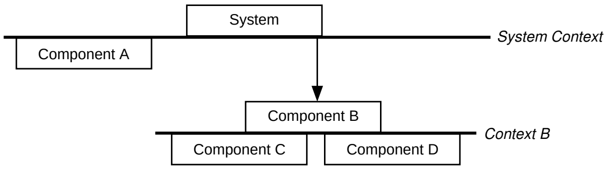


[1] Whenever we use the expression *system*, we implicitly assume that systems can be seen as *components* themselves.

The main idea of the Context-Component pattern is to separate the configuration of a system from the implementation of its components. We call the configuration of systems *Contexts*. Formally, a context aggregates the components of a system. Every component is embedded in exactly one context. It refers to this context in order to access the other components of the system. For this purpose, the context object offers a *Factory Method* [13] for every of its components. These methods specify the instantiation protocol of the different components. Typically, either a new instance of a component is created for every factory method call, or the component is a *Singleton* [13] with respect to the context. In this case, the component is instantiated only once during the first call of the factory method.

Components that represent more complex subsystems, like component B from our example above, have an own local configuration. In other words, they are embedded in their own context, specifying all their local subcomponents. This shows that contexts have a nested structure. Every context might have subcontexts for more complex subsystems. The context in which an embedded subcontext is nested, is called the *enclosing context*. Components defined in a nested context can access the components of their own context and all the components defined in enclosing contexts. On the other hand it is not possible for a component to access components defined in subcontexts directly.

We introduce a graphical notation to illustrate the structure. For the scenario mentioned in the beginning of this section, we get the following picture:



Contexts are represented by lines. Singleton components embedded in a context correspond to boxes located directly beneath the line. A contexts non-singleton components are depicted as lifted boxes with an arrow pointing to them. More complex components refer to subcomponents defined in local contexts, which are drawn as lines directly beneath the components box.

5.2 Structure

The structure of the architectural pattern is shown in figure 2. Our pattern has four different participants:

Context Context is the superclass of all contexts. It simply defines a generic reference to the enclosing context.

Component The abstract superclass of all components defines a method init which is called immediately after component creation to initialize the component. The context in which the component is embedded is passed as an argument to init. init typically gathers references to other components that are accessed within the component.

ConcreteContext A concrete context defines a particular context of a system. It provides factory methods for all embedded components. These methods specify

- whether a component is a singleton relative to the context, and
- whether a component is initialized in an own nested context, defining local subcomponents.

Furthermore, a ConcreteContext provides factory methods for creating local subcontexts.

ConcreteComponent A ConcreteComponent implements a specific component of a system. It defines a customized init method which is called from the corresponding context immediately after object creation. The context is passed as an argument, enabling the init method to import references to all components which are accessed within the component. It is only possible to import components from the own or an enclosing context.

Overloading the init method allows a flexible embedding of components in different ConcreteContext classes. The init methods act as adaptors to the different contexts in which a component can be embedded.

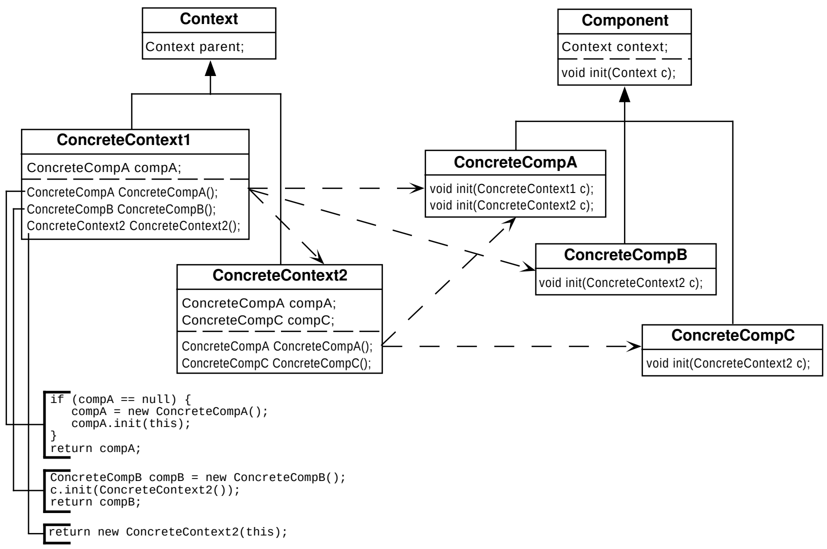


Fig. 2. Structure of the architectural pattern *Context-Component*

An important design decision in the pattern above is the separation of component creation and component initialization. This separation is important to break cycles in the dependency-graph of the components. Let's look at the scenario of figure 2. By using our symbolic notation we get the following diagram:

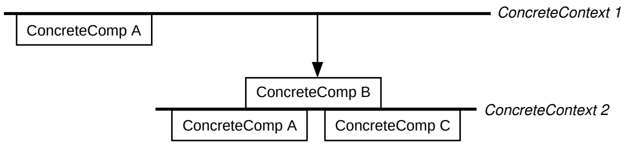


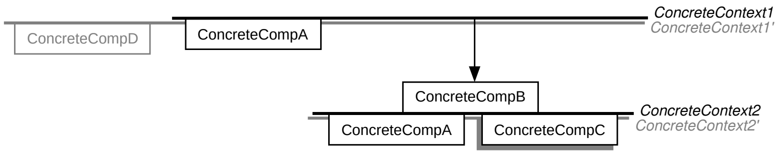
Figure 2 shows the implementations of the factory methods of context ConcreteContext1. For singletons like ConcreteCompA it is important that the object is first created and then initialized in a second step. Otherwise instantiation of mutually dependent components would cause an endless loop in which alternately new components are created permanently.

5.3 Consequences

The Context-Component pattern is a composite architectural design pattern. Contexts are a combination of an *AbstractFactory* [13] and an *ObjectServer*. They support hierarchical organizations of complex systems. Contexts offer a uniform and extensible way to configure systems. Since the components of a system are defined explicitly and centrally within a context class, the context hierarchy can also be seen as a formal specification of a system architecture.

The Context-Component pattern decouples system composition and implementation of components. This enables a much more flexible reuse of components in different contexts. Only an adaptor in form of a new init method is necessary to embed a component in a different context.[2]

Furthermore, with the Context-Component pattern it is possible to exchange and add new components to a system without the need for any source code modifications of existing components or contexts. Extended systems evolve out of existing ones simply by subclassing. Figure 3 shows the principle by extending the system of Figure 2. The system gets extended by a new top-level component ConcreteCompD. This is done by extending the top-level context ConcreteContext1. In addition, the component ConcreteCompC gets replaced by the new component ConcreteCompC' in the local context ConcreteContext2 of component ConcreteCompB. Figure 3 highlights new classes with a gray background. In our symbolic notation the scenario looks like this:



Again, new components or contexts are displayed in gray. Components with a gray shadow have been extended (subclassed). This example shows that extending or modifying a system does not entail any source code modifications of existing classes. Therefore extending a system does not destroy the original version. Both systems can still be used separately.

[2] See component ConcreteCompA from Figure 2; this component is used in two different contexts.

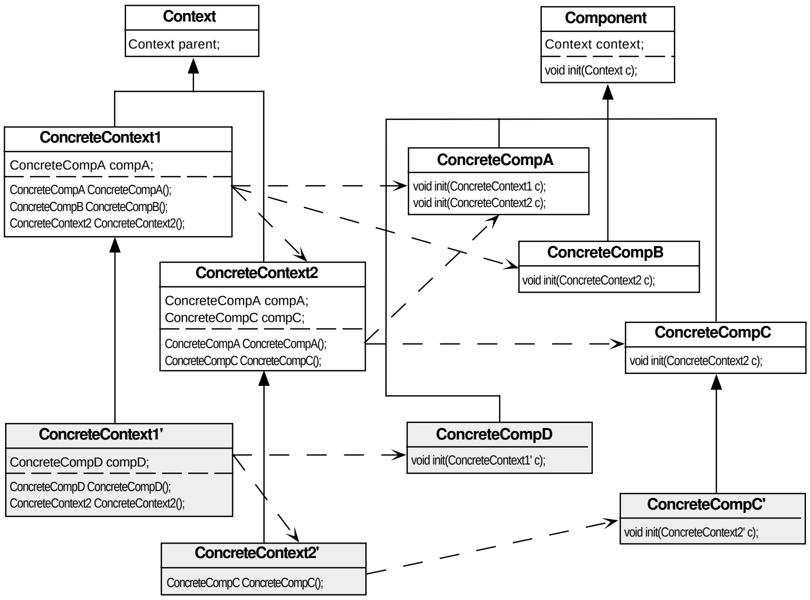


Fig. 3. Extending a system by subclassing

6 Extensible Compilers

With the techniques developed in section 3 and 5 we are now able to build extensible compilers. We base our software architecture for extensible compilers on the classical design of a *multi-pass compiler* [22]. A multi-pass compiler decomposes compilation into a number of subsequent phases. Conceptually, each of them is transforming the program representation until target code is emitted. Today, most compilers use a central data structure, the *abstract syntax tree*, for the internal program representation. This syntax tree is initially generated by the *Parser* and modified continuously in the following passes. From the software architecture's point of view, this design can be classified as a *Repository* [26].

We now apply the Context-Component design pattern. Figure 4 shows the structure of a simple compiler. The compiler is modelled as a component of the top-level Tools context. The compiler is a composite component, consisting of several subcomponents that are defined in the local CompilerContext. Except for the component ErrorHandler, these subcomponents model the different compilation passes. By defining ErrorHandler as a singleton component with respect to its context, every compiler pass accesses the same ErrorHandler object.

The implementation of this structure with the Context-Component pattern is straightforward. We only show some interesting code fragments, starting with the Tools class:

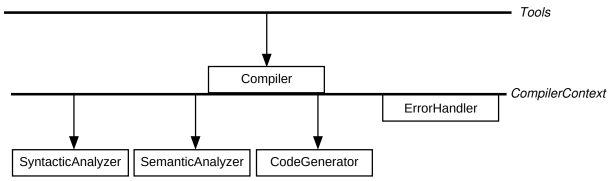


Fig. 4. A simple compiler architecture

```
class Tools extends Context {
    Compiler Compiler() {
        Compiler c = new Compiler();
        c.init(CompilerContext());
        return c;
    }
    CompilerContext CompilerContext() {
        return new CompilerContext(this);
    }
}
```

The Tools class defines the factory method for the Compiler component and the nested CompilerContext. [3] The CompilerContext class contains the actual configuration of the compiler. It defines the different compiler passes and a global ErrorHandler component.

```
class CompilerContext extends Context {
    CompilerContext(Tools encl) { super(encl); }
    SyntacticAnalyzer SyntacticAnalyzer() {
        SyntacticAnalyzer c = new SyntacticAnalyzer();
        c.init(this);
        return c;
    }
    SemanticAnalyzer SemanticAnalyzer() { ... }
    CodeGenerator CodeGenerator() { ...}
    ErrorHandler ehandler = null;
    ErrorHandler ErrorHandler() {
        if (ehandler == null) {
            ehandler = new ErrorHandler();
            ehandler.init(this);
        }
        return ehandler;
    }
}
```

In our compiler, we represent data like abstract syntax trees with extensible algebraic types. Therefore, most compiler pass implementations are similar to

[3] We follow the naming convention of giving factory methods the name of the type they return.

the one of SemanticAnalyzer. They define a method operating on the abstract syntax tree for performing the actual compiler pass. Pattern matching is used to distinguish the different Tree nodes.

```
class SemanticAnalyzer extends Component {
    ErrorHandler ehandler;
    void init(CompilerContext cc) {
        ehandler = cc.ErrorHandler();
    }
    void analyze(Tree tree) {
        switch (tree) {
            case Variable(String name): ...
            ...
        }
    }
}
```

Finally, we present the implementation of the main compiler component. It accesses all its compiler passes and executes them sequentially.

```
class Compiler extends Component {
    SyntacticAnalyzer syntactic;
    SemanticAnalyzer semantic;
    CodeGenerator codegen;
    void init(CompilerContext cc) {
        syntactic = cc.SyntacticAnalyzer();
        semantic = cc.SemanticAnalyzer();
        codegen = cc.CodeGenerator();
    }
    void compile(String file) {
        Tree tree = syntactic.parse(file);
        semantic.analyze(tree);
        codegen.generate(tree);
    }
}
```

Figure 5 depicts a possible extension of our compiler. We assume, the source language was extended. Therefore we need a new syntactical and semantical analysis. Furthermore we introduce a new compilation pass Translate that transforms syntax trees of our extended language into trees of the original source language. Since we are still able to use the original semantical analysis, we can check our translated program again before applying the code generator we adopt from the old compiler. This second semantical analysis might be imposed by the translator, which does not preserve attributes like typings, determined by the first semantical analysis. The code generator typically relies on a proper attribution of the structure tree and therefore requires a second semantical analysis after the syntax tree transformation. Extensions of the Java programming language are usually implemented this way. Here is an implementation of the new compiler context hierarchy:

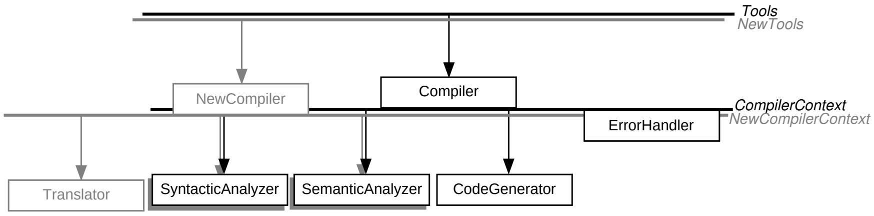


Fig. 5. An extended compiler architecture

```
class NewTools extends Tools {
    NewCompiler NewCompiler() {
        NewCompiler c = new NewCompiler();
        c.init(NewCompilerContext());
        return c;
    }
    NewCompilerContext NewCompilerContext() {
        return new NewCompilerContext(this);
    }
}
```

In the NewTools context we do not override the existing Compiler factory method. So we are able to call both, the new and the old compiler from that context. The NewCompilerContext class provides an extended syntactical analysis and includes two new compiler passes.

```
class NewCompilerContext extends CompilerContext {
    NewCompilerContext(NewTools encl) { super(encl); }
    SyntacticAnalyzer SyntacticAnalyzer() {
        NewSyntacticAnalyzer c = new NewSyntacticAnalyzer();
        c.init(this);
        return c;
    }
    NewSemanticAnalyzer NewSemanticAnalyzer() {
        NewSemanticAnalyzer c = new NewSemanticAnalyzer();
        c.init(this);
        return c;
    }
    Translator Translator() {
        ...
    }
}
```

One of the new passes is NewSemanticAnalyzer, which extends an already existing component. Thus, our extended compiler uses both, the former semantic analyzer which gets inherited to NewCompilerContext and the new extended pass. Here is a possible implementation of the new semantic analyzer. It refines the analyze function by overriding the existing analyze method.

```
class NewSemanticAnalyzer extends SemanticAnalyzer {
    void analyze(Tree tree) {
        switch (tree) {
            case Zero: ...
            case Succ(Tree tree): ...
            default: super.analyze(tree);
        }
    }
}
```

Finally, we can implement our new main Compiler component accordingly.

```
class NewCompiler extends Compiler {
    NewSemanticAnalyzer newsemantic;
    Translator trans;
    void init(NewCompilerContext cc) {
        super.init(cc);
        newsemantic = cc.NewSemanticAnalyzer();
        trans = cc.Translator();
    }
    void compile(String file) {
        Tree tree = syntactic.parse(file);
        newsemantic.analyze(tree);
        tree = trans.translate(tree);
        semantic.analyze(tree);
        codegen.generate(tree);
    }
}
```

This example shows how flexible components and configurations (contexts) can be extended and reused in our framework. This is due to a strict separation of datatype definitions, component implementations and the configuration of systems. Extensible algebraic datatypes with defaults provide a mechanism for separating datatype definitions and components, whereas the Context-Component pattern yields a separation of components from system configurations.

7 Experience

We implemented a full Java 2 compiler with the techniques introduced in this paper [29]. Throughout the last two years, several non-trivial compiler extensions were built by various people as part of different projects [5, 9, 25, 30, 32]. Among the implementations is a compiler for Java with synchronous active objects, proposed by Petitpierre [23]. Another extension introduces Büchi and Weck's compound types together with type aliases [2]. We added operator overloading to the Java programming language in the style Gosling proposes [16]. Furthermore, a domain specific extension of Java providing publish/subscribe primitives was implemented [9].

Our extensible Java compiler *JaCo* [29] turned out to be a very valuable tool to rapidly implement prototype compilers for language extensions of Java. Often, people proposing extended Java dialects refrain from implementing their ideas, since crafting a full compiler from scratch is a very time consuming task. Even if an existing compiler is used as a basis for the implementation, maintenance poses additional challenges. A language like Java is subject to constant changes, requiring regular modifications of the compiler. Keeping an extended compiler synchronized to its base compiler is usually done by comparing source code by hand, which is error-prone and again time consuming.

Our extensible compiler does not only help to quickly implement language extensions for Java. It also provides an infrastructure for maintaining all compiler extensions together. 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 extensions 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 a way that binary compatibility of Java classfiles is not broken. As a consequence, even compilers derived from the base compiler benefit immediately from the changes, since they inherit them. No recompilation of any compiler extension is necessary thanks to Java's late binding.

8 Conclusion

For experimenting with programming language extensions, extensible compilers are essential to rapidly implement extended languages. We presented some fundamental techniques to develop extensible compilers. We showed how to use *extensible algebraic datatypes with defaults* in an object-oriented language to implement extensible abstract syntax trees and compiler passes. These types enable us to freely extend datatypes and operations simultaneously and independently of each other. Even though extensible algebraic types allow to write extensible syntax trees and compiler passes, it is the overall compiler architecture which determines how flexible a system can be extended or reused. We proposed a general architectural design pattern *Context-Component* to build extensible, hierarchically structured component systems. This pattern strictly separates the composition of systems from the definition of its components. We showed how to use this object-oriented pattern in conjunction with extensible algebraic types to build compilers that can be extended, modified and reused in a very flexible manner. Extending a compiler does not require any source code modifications of the base system. Extended compilers evolve out of existing ones simply by subclassing. Since they share most components and system configurations with their predecessors, our technique provides a basis for maintaining the systems together. We implemented a full Java compiler based on this technique. In the last two years, this compiler was used in various other projects to quickly implement new language extensions of Java.

Acknowledgments

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

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 *Proceedings of OOPSLA'98*, pages 362–373, October 1998.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern–Oriented Software Architecture : A System of Patterns*. John Wiley & Sons, 1996.

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

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

[6] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. Technical Report 00-06a, Iowa State University, Department of Computer Science, July 2000. To appear in OOPSLA 2000.

[7] W. 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.

[8] 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.

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

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

[11] 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.

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

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

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

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

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

[17] 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.

[18] T. Kühne. The translator pattern — external functionality with homomorphic mappings. In R. Ege, M. Singh, and B. Meyer, editors, *The* 23^{rd} *TOOLS conference USA 1997*, pages 48–62. IEEE Computer Society, July 1998. 28.7-1.8, 1997, Santa Barbara, California.

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

[20] 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.

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

[22] D. Perry and A. Wolf. Foundations for the study of software architecture, 1992.

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

[24] 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.

[25] Y. Saidji. Operator overloading in java. Projet de semestre. École Polytechnique Fédérale de Lausanne, Switzerland, June 2000.

[26] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[27] C. Szyperski. Import is not inheritance. why we need both: Modules and classes. In B. Krieg-Brückner, editor, *ESOP '92: 4th European Symposium on Programming, Rennes, France, Proceedings*, Lecture Notes in Computer Science 582. Springer-Verlag, February 1992.

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

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

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

[31] M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *Proc. International Conference on Functional Programming (ICFP 2001)*, Firenze, Italy, September 2001.

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