Extensible Algebraic Types for Java

DRAFT

Matthias Zenger
matthias.zenger@epfl.ch

February 2000

1 Introduction

Most functional programming languages provide user defined data types in form of algebraic types. Java's object types and inheritance are complementary to algebraic types and pattern matching. Where object types and inheritance make it easy to extend the set of constructors for the type, algebraic types and pattern matching make it easy to add new operations over the type, while the set of constructors is fixed. The extensible algebraic types described in this paper offer both: extensions of the constructor set as well as the addition of new operations. For declaring algebraic types we use the syntax introduced by the programming language Pizza [OW97]. Here is an algebraic type for binary trees:

```
class BinaryTree {
   case Leaf(Object obj);
   case Node(Tree left, Tree right);

   boolean find(Object obj) {
      switch (this) {
         case Leaf(Object x):
            return x.equals(obj);
         case Node(BinaryTree left, BinaryTree right):
            return left.find(obj) || right.find(obj);
         default:
            return false;
      }
   }
}
```

Each of the two case declarations introduces a constructor for the algebraic type `BinaryTree`. The following expression would create a tree consisting of two nodes and three leafs:

```
BinaryTree.Node(BinaryTree.Leaf("one"),
                BinaryTree.Node(BinaryTree.Leaf("two"),
                                BinaryTree.Leaf("three")))
```

Method `find` of class `BinaryTree` demonstrates the use of the `switch` statement for pattern matching. This method traverses the tree while trying to find the given element. The `switch` statement performs a pattern matching over three cases. The first case handles leafs. It binds

the freshly defined variable `x` to the corresponding object of the `Leaf` node. The second case is similar for inner nodes. If we would not have extensible types, the last `default` case would be superfluous. But for our types the set of constructors can be extended. So we have to specify this default case representating all cases defined in extensions of type `BinaryTree`.
In the following example we extend the algebraic type `BinaryTree`. The extended algebraic type `Tree` inherits all the constructors from `BinaryTree` and defines a new constructor `NewNode` for creating trees with three children. The new type `Tree` is a subtype of `BinaryTree`.[1]

```
class Tree extends BinaryTree {
   case NewNode(Tree left, Tree middle, Tree right);

   boolean find(Object obj) {
      switch (this) {
         case NewNode(Tree l, Tree m, Tree r):
            return l.find(obj) || m.find(obj) || r.find(obj);
         default:
            return super.find(obj);
      }
   }
}
```

This example also shows how to adapt method `find`. The declared method overrides `find` in class `BinaryTree`. It performs a pattern matching over values created with the new constructor and delegates all other cases to the overridden method.
The rest of this paper gives a detailed specification of extensible algebraic types for Java. All grammars of the following sections extend the Java grammar of the Java Language Specification [GJS96].

2 Algebraic classes

An extensible algebraic type is defined by an *algebraic class*. A class is *algebraic* if the class definition contains at least one `case` declaration or if the class extends another algebraic class. A `case` declaration introduces a new constructor for the algebraic type. An algebraic type A with two constructors A_1 and A_2 is defined in the following way

```
class A {
   case A₁(f̄₁);
   case A₂(f̄₂);
}
```

Every case $\mathtt{A_i}$ defines a *case type* $A.A_i$ with fields $\bar{f}_i = T_{i,1}\, v_{i,1}, \ldots, T_{i,r_i}\, v_{i,r_i}$, where $T_{i,j}$ are types and $v_{i,j}$ variable names. Furthermore, a static constructor method $\mathtt{A.A_i}$ of type $T_{i,1} \times \ldots \times T_{i,r_i} \rightarrow A_i$ is defined implicitly that creates a new object of type $A.A_i$. Here is the grammar for algebraic type declarations in the notation of [GJS96]:

$CaseDeclaration$:
  $ClassModifiers_{opt}$ `case` $Identifier$ $CaseFormals_{opt}$ $MethodBody$

$CaseFormals$:
  ($FormalParameterList_{opt}$)

[1] For ordinary algebraic types this seems to be contradictory to set theory. With more constructors the extended type should be a supertype of the original one. Section 4 on subtyping discusses in detail why this does not apply to our extensible algebraic datatypes.

It is possible to add a method body to a `case` declaration. The parameter list of the `case` construct is in the scope of this method body. The body is executed whenever the constructor is called. With this mechanism it is possible to perform specialized object initializations.
For a `case` declaration it is legal to omit parameters. Case X_1 in the following example code is an example for that:

```
class X extends Object {
   case X1;
   case X2();
}
```

Instead of defining a case type, X_1 represents a constant $\mathtt{X.X_1}$ of type X. X_2 doesn't have any parameters either, but it is treated as every other case with parameters. So there exists a case type X_2 and a constructor $X.X_2$ of type $() \rightarrow X_2$.
Legal modifiers for algebraic classes are `public`, `abstract` und `final`. An algebraic class has to be declared `abstract` if it does not contains any `case` declarations. This is only possible for algebraic classes that extend another algebraic cass. Declaring an algebraic class `final` prevents it from beeing extended.

3 Extending algebraic classes

Types A and X of the preceding section are called *algebraic root types*. The super class of an algebraic root class is non-algebraic. Subclasses of an algebraic class are automatically algebraic. They inherit all the cases of the algebraic super class and define new additional cases. The types defined by these classes are called *extended algebraic types*. The following example shows an extended algebraic type B that inherits all cases from A and defines a new case $\mathtt{B_1}$.

```
class B extends A {
   case B1(f̄3);
}
```

This way of extending algebraic types is called a vertical extension [DS96]. It is also possible to extend single cases of an algebraic type by subclassing. This horizontal extension refines cases while leaving the algebraic type to which the cases belong unmodified. The next code fragment defines a subclass A_1' of A_1. A_1 is one of the cases of algebraic type A. A_1' extends case A_1 by new additional fields $\bar{f}_1'$.

```
class A1' extends A1 {
   f̄1'
   A1'(f̄1, f̄1') {
      super(v̄1);
      ...
   }
}
```

Horizontal extensions are not only useful for adding new fields. They can also be used specifically for overriding methods; i.e. refining methods for special cases of algebraic types.

4 Subtyping

As mentioned in section 2 already, every case A_i of an algebraic type A defines a case type $A.A_i$. These case types are subtypes of A. More specifically, every case defined in A or in any extension of A is a subtype of A. Every algebraic type B that extends A inherits the cases from A. Therefore every case type $A.A_i$ is also a subtype of B. With this, algebraic type B is a subtype of algebraic type A. Section 7 explains the theory behind this in detail.

5 Attributes of algebraic classes

Like every Java class, algebraic classes can have variables, methods and inner classes in addition to case declarations. Every case declaration defines a constructor for the algebraic type. Therefore algebraic classes don't have ordinary Java-style constructors. Variables directly defined in an algebraic class are inherited to all *case classes*. So every case has got the variables defined in it's own declaration and the variables inherited from the algebraic type it belongs to.

$$\begin{array}{l} ClassMemberDeclaration: \\ \quad FieldDeclaration \\ \quad MethodDeclaration \\ \quad InnerClassDeclaration \\ \quad CaseDeclaration \end{array}$$

There are no further restrictions for algebraic root classes. For extended algebraic classes it is required that

- new interfaces can only be implemented if this doesn't introduce new super types,
- there are no non-static variable declarations, and,
- all non-static methods defined in the extended algebraic class override methods from the super class.

These restrictions keep the non-static interface of an extended algebraic class identical to the algebraic base class. It is the algebraic base class that defines the interface for all its extensions. This property is enforced by the fact that constructors of an algebraic class are inherited to the algebraic subclasses. Since objects created by such a case constructor have a fixed interface, all algebraic subclass that inherit this case have to have the same interface as well.
The rest of this section discusses overloading and overriding issues for methods defined in algebraic classes. The next example is used as a starting-point:

```
class Alpha {
   case Case1();
   void foo() {
      System.out.println("Alpha");
   }
}
class Beta extends Alpha {
   case Case2();
   void foo() {
      System.out.println("Beta");
```

```
    }
}
```

Now the following code sequence is evaluated:

```
Beta b2 = Beta.Case2();
Beta b1 = Beta.Case1();
b2.foo();
b1.foo();
```

The first call to `foo` prints "'Beta"' as expected. One might think the second call results in the same print out. But actually method `foo` of class `Beta` overrides only method `foo` of `Alpha` for cases defined in class `Beta`. For all cases defined in class `Alpha` the original method is called. As a consequence, `foo` prints "'Alpha"' in the second method call of the code sequence above. This behaviour might look surprising in the beginning, but it is natural for vertical extensions as the following example will show.

```
Beta  b = Alpha.Case1();
Alpha a = b;
b.foo();
a.foo();
```

For both method calls one expects method `foo` of class `Alpha` beeing called. This is indeed true since both method calls have at runtime the same receiver and the static type is irrelevant for the method dispatch. So "overloaded" method `foo` of class `Beta` is never called for cases of class `Alpha`. In this context overloading has the purpose to refine an extisting method for dealing with new variants. Changing a method for an existing case is done by extending the case horizontally. The method has to be overidden in the subclass of a case class for this purpose.

Extending Java with extensible algebraic types also affects overloading of methods. This is discussed using the following code fragment:

```
class Test {
   void goo(Alpha a) {
      ...
   }
   void goo(Beta b) {
      ...
   }
   static void bar() {
      goo(Beta.Case2());
      goo(Beta.Case1());
   }
}
```

Class `Test` defines two instances for overloaded method `goo`. For both calls to `goo` in `bar` the Java compiler has to resolve the best fitting method statically. For the first call `goo(Beta.Case`$_2$`())` method `goo(Beta b)` fits best since *Beta* is an immediate supertype of $Case_2$. For the second call `goo(Beta.Case`$_1$`())` both methods fit in the same degree. $Case_1$ is an immediate subtype of *Alpha* as well as of *Beta*. So there is no obvious criteria which determines a method fitting better. The following rule excludes explicitly this problem:

> For every method f with overloaded instances $f(T_1, \ldots, T_n)$ and $f(U_1, \ldots, U_n)$ there has to be at least one $i \in \{1, \ldots, n\}$ where T_i and U_i are incompatible.

Two types T_1, T_2 are *incompatible*, $T_1 \# T_2$, iff they are neither equal nor subtype of the same algebraic type; i.e. $T_1 \# T_2 \Leftrightarrow \forall T : T_1 \preceq T \Rightarrow T_2 \not\preceq T$.

6 Pattern Matching

Pattern Matching is added to the Java programming language by extending the `switch` statement. A `switch` statement transfers control to one of several statements depending on the value of a *selector expression*.

SwitchStatement :
  `switch` (*Expression*) *SwitchBlock*

SwitchBlock :
  { $\mathit{SwitchBlockStatementGroups}_{opt}$ $\mathit{SwitchLabels}_{opt}$ }

SwitchBlockStatementGroups :
  SwitchBlockStatementGroup
  SwitchBlockStatementGroups SwitchBlockStatementGroup

SwitchBlockStatementGroup :
  SwitchLabels BlockStatements

SwitchLabels :
  SwitchLabel
  SwitchLabels SwitchLabel

SwitchLabel :
  `case` *TopLevelPattern* :
  `default` :

The type of the selector must be either `char`, `byte`, `short`, `int` or an algebraic type. A set of *patterns* is associated with every *switch block*. The type of a pattern has to fit to the selector type of the `switch` statement. That is, for primitive types the pattern types have to be assignable to the selector type. For algebraic selector types, the case types have to be cases of this algebraic type.
When the `switch` statement is executed, first the selector expression is evaluated. If evaluation of this expression completes abruptly for some reason, the `switch` statement completes abruptly for the same reason. Otherwise, execution continues by comparing the value of the selector with each pattern. The switch block of the first matching pattern is executed. For pattern matching over algebraic types, it is illegal to fall from one `switch` block into the next. This may happen if the first switch block is not terminated by a `break` statement for example.
Patterns in `switch` blocks are written according to the following grammar:

TopLevelPattern :
  ConstantExpression
  ConstructorPattern

ConstructorPattern :
  ConstructorName $\mathit{PatternParameterList}_{opt}$

PatternParameterList :
  ($Patterns_{opt}$)

Patterns :
  Pattern
  Patterns , *Pattern*

Pattern :
  TopLevelPattern
  FormalParameter
  _

This grammar defines patterns to be either

- an empty pattern _,
- a formal parameter/variable,
- a constant expression according to §15.27 of [GJS96], or
- an algebraic pattern $c(p_1, \ldots, p_n)$, where c is a constructor with arity n and $p_1, \ldots, p_n$ are legal patterns.

All variables defined in a pattern have to be distinct. If more than one pattern is associated with the same `switch` block then it is not legal to have any variables in these patterns. Patterns in a `switch` statement may overlap. It is not required that all cases are covered by the patterns of a `switch` statement.

Matching a value v with a pattern p is based on the following rules:

Fall 1: The type of v ist either a *primitive type* or type `String`: v matches with pattern p, if p is the empty pattern, a variable, or a constant expression with value v.

Fall 2: The type of $v = c(v_1, \ldots, v_n)$ is *algebraic type* A: value v matches with pattern p if

1. $p =$ _ or p is a variable of type A,
2. p is a variable of case type c,[2] or
3. p has form $c(p_1, \ldots, p_n)$ and for all $i \in \{1, \ldots, n\}$, v_i matches with sub-pattern p_i.

Fall 3: The type of v is a *non-algebraic reference type*: v matches with pattern p if $p =$ _ or p is a variable of the same type.

Figure 1 shows a simple example program demonstrating some of the details mentioned before. In this example, an algebraic type *BinTree* for representing binary trees is defined. The final algebraic class `IntTree` extends `BinTree` by providing a new case `Leaf`. Class `Functions` declares some operations on both algebraic types. Even though type *BinTree* consists of only a single case *Branch*, the `switch` statement of the `reflect` method would never be complete without a `default` case. This case is required since it is possible to extend type *BinTree* and

[2]This pattern form acts as a type constraint. Only values of a particular case type match with this pattern.

```
class BinTree {
   case Branch(BinTree l, BinTree r);
}
final class IntTree extends BinTree {
   case Leaf(int i);
}
class Functions {
   BinTree reflect(BinTree tree) {
      switch (tree) {
         case Branch(BinTree l, BinTree r):
            return BinTree.Branch(reflect(r), reflect(l));
         default:
            return tree;
      }
   }
   boolean hasZero(IntTree tree) {
      switch (tree) \{
         case Branch(BinTree l, BinTree r):
            return hasZero(l) || hasZero(r);
         case Leaf(0):
            return true;
         case Leaf(\_):
            return false;
      }
   }
}
```

Figure 1: Pattern Matching f"ur erweiterbare algebraische Typen

applying `reflect` to objects of extended types like $IntTree$ is legal. It is also important to stress that patterns of the form `Leaf`(...) cannot be used within the `switch` statement of the `reflect` method. Only cases of the selectors static type are legal. Cases of extensions are subsumed by the `default` case.
Pattern matching in method `hasZero` is complete without a `default` case. Algebraic type $IntTree$ is not extensible and therefore the given patterns cover all possible cases.

7 Typing

7.1 Subtyping for extended algebraic types

The discussion about typing issues refers to the following example:

```
class A {
   case A₁(f̄₁);
   case A₂(f̄₂);
}
class B extends A {
   case B₁(f̄₃);
}
```

Extensible algebraic types are only useful if algebraic type B is a subtype of A. But for ordinary algebraic types the opposite subtype relationship holds as will be shown. Formally, an algebraic type can be modelled as a type sum. For the code above, we get the following types:

$$\begin{array}{rcl} A & = & A_1 + A_2 \\ B & = & A_1 + A_2 + B_1 \end{array}$$

Subtyping captures the intuitive notion of inclusion between types, where types are seen as collections of values [CW85, Car97]. An element of a type can be considered also as an element of its supertype. With this notion, A is a subtype of B, $A \leq B$. Types where extensions are supertypes are useless in practice since this disables code reuse completely [MZ98].

The classical approach of describing an algebraic type by a fixed set of constructors does not seem to work for extensible algebraic types. The basic idea is to desribe extensible algebraic types by a minimum set of constructors. Every extension has to support these constructors and can define additional ones. This notion yields a type theoretical modelling with open type sums:

$$\begin{array}{rcl} Y & = & \mathit{inherited}_Y + \mathit{cases}_Y + \mathit{default}_Y \\ & & \text{where} \quad \begin{array}{lcl} \mathit{cases}_Y & = & \sum\limits_i Y_i \\ \mathit{inherited}_Y & = & \sum\limits_{Y \preceq X} \mathit{cases}_X \\ \mathit{default}_Y & = & \sum\limits_{Z \preceq Y, Z \neq Y} \mathit{cases}_Z \end{array} \end{array}$$

$\preceq$ is called algebraic extension relation. This relation is defined explicitly by type declarations. For two types X and Y, $Y \preceq X$ holds, iff X and Y are algebraic types and Y is an extension of X. An extensible algebraic type Y is defined by three disjoint type sums cases_Y, $\mathit{inherited}_Y$ and $\mathit{default}_Y$. $\mathit{inherited}_Y$ includes all inherited cases, cases_Y denotes Ys new cases, and $\mathit{default}_Y$ subsumes all cases of extensions of Y. $\mathit{default}_Y$ keeps the type sum open.

With this understanding, our types A and B now look like this:

$$\begin{array}{rcl} A & = & A_1 + A_2 + \mathit{default}_A \\ B & = & A_1 + A_2 + B_1 + \mathit{default}_B. \end{array}$$

Since the open type sum $\mathit{default}_A$ captures B_1 as well as $\mathit{default}_B$, $B_1 + \mathit{default}_B \leq \mathit{default}_A$ holds. Note that $B_1 + \mathit{default}_B \neq \mathit{default}_A$ because according to its definition above, $\mathit{default}_B$ subsumes only extensions of B. All cases of any other extension of A are not covered by $\mathit{default}_B$. This is illustrated by the following class declaration:

```
class C {
   case C₁(f̄₄);
}
```

Case C_1 is not included in $\mathit{default}_B$, but is part of $\mathit{default}_A$. As a consequence, $B_1 + \mathit{default}_B$ is a real subtype of $\mathit{default}_A$, and $B \leq A$ holds. All subtype relationships of our example are illustrated in figure 2.

7.2 Type system

Adapting the Java type system to extensible algebraic types only requires some minor modifications to the subtype relationship between reference types.

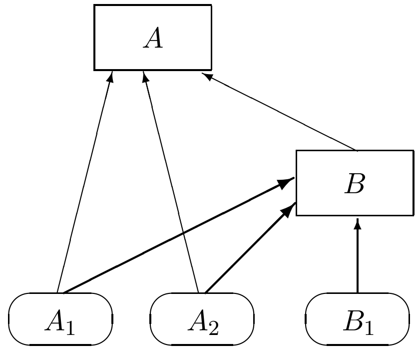

Figure 2: Subtyping for extensible algebraic types

This section uses the terminology from [OW97]. Δ specifies the global class environment, which consists of entries of the form c:$\mathsf{class}(\Gamma, C, \bar{I})$ for both Java object types and algebraic types. Γ represents the local class environment, C is the super class, and $\bar{I}$ is the set of implemented interfaces. We assume $\mathcal{A} \subseteq \Delta$ to be the set of all algebraic classes. For every case of an algebraic type, there is an appropriate entry a :$\mathsf{class}(\Gamma, C, \bar{I}) \in \mathcal{A}$ in the local environment Γ of the corresponding algebraic class. Every case has an additional proper class entry in Δ. Δ yields a subtype relation $\leq$ between reference types in Java according to the rules given in figure 3.

(Top) $\quad X \leq \texttt{java.lang.Object}$

(Refl) $\quad X \leq X$

(Trans) $$\frac{X_1 \leq X_2 \quad X_2 \leq X_3}{X_1 \leq X_3}$$

(Super) $$\frac{c{:}\mathsf{class}(\Gamma, C, \bar{I}) \in \Delta}{c \leq C}$$

(Intf) $$\frac{c{:}\mathsf{class}(\Gamma, C, \bar{I}) \in \Delta \quad X \in \bar{I}}{c \leq X}$$

(Case) $$\frac{a_1{:}\mathsf{class}(\Gamma, C, \bar{I}) \in \Delta \quad c{:}\mathsf{case}(\bar{f}) \in \Gamma \quad a_2 \preceq a_1}{c \leq a_2}$$

Figure 3: Extended subtyping relation $\leq$ for Java

The first five rules describe subtyping for regular Java. The (Case)-rule introduces the subtype relationship between case types of an algebraic type and all extensions of this type. This rule does not interfere with regular Java semantics since it establishes new subtype relationships only for case types that do not exist in standard Java.
The definition of the (Case)-rule uses the *algebraic extension relation* $\preceq$ mentioned before. $a_1 \preceq a_2$ holds for two types a_1 und a_2, iff $a1$ is an algebraic extension of type a_2. Figure 4 contains a formal definition of $\preceq$.
Finally the last example should demonstrate that $\preceq$ is not only a restriction of $\leq$ to the set of

$$\text{(Refl)} \qquad a \preceq a$$

$$\text{(Trans)} \qquad \frac{a_1 \preceq a_2 \quad a_2 \preceq a_3}{a_1 \preceq a_3}$$

$$\text{(Extends)} \qquad \frac{a : \mathsf{class}(\Gamma_1, C, \bar{I}_1) \in \mathcal{A} \quad C : \mathsf{class}(\Gamma_2, C_2, \bar{I}_2) \in \mathcal{A}}{a \preceq C}$$

Figure 4: Algebraic extension $\preceq$

algebraic classes $\mathcal{A}$. Type D defined in this example is a subtype of Type A, but no algebraic extension of A. That is, $D \leq A$, but not $D \preceq A$.

```
class A {
   case A₁(f̄₁);
   case A₂(f̄₂);
}
class D extends A₁ {
   case D₁(f̄₅);
}
```

References

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys*. 17(4):471-522, December 1985.

[Car97] Luca Cardelli. *Type Systems*. Digital Equipment Corporation, Systems Research Center, 1997.

[B+98] Gilad Bracha, Martin Odersky, David Stoutamire and Philip Wadler. GJ Specification. May, 1998.

[DS96] Dominic Duggan and Constantin Sourelis. Mixin Modules. In *ACM SIGPLAN International Conference on Functional Programming*. pages 262-273, May 1996.

[GJS96] James Gosling, Bill Joy and Guy Steele. *The Java™ Language Specification*. Java Series, Sun Microsystems, 1996. ISBN 0-201-63451-1.

[Ode97] Martin Odersky. Pizza Distribution, University of South Australia, 1997. `http://www.cis.unisa.edu.au/~pizza`.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Symposium on Principles of Programming Languages*. pages 146-159, 1997.

[MZ98] Matthias Zenger. Erweiterbare Übersetzer. *Masters Thesis*, University of Karlsruhe, 1998.