Java Annotations for Algebraic Data Types

Sven Eric Panitz
www.panitz.name

FH Wiesbaden

Abstract. Algebraic data types are known from functional programming languages. An algebraic type is defined by a set of constructors. Usually algebraic types are recursive and represent some hierarchical structure. Algorithms on algebraic types traverse over the tree structure. Typical examples for algebraic types are defined by XML schemas or DTDs.
In functional language tree traversal can be easily implemented through pattern matching on the different constructors. In object oriented languages the visitor pattern is the standard means for implementing pattern matching. Implementing algebraic types and visitors in Java amounts in a lot of boilerplate code.
The paper shows, how Javas new annotation mechanism can be used to define algebraic types in an elegant way similar to the definitions in functional languages. The complete type gets generated through a generator class, which is invoked through Java's annotation processing tool `apt`.
The paper contains the complete sources of the generator together with some examples.

1 Introduction

Algebraic data types are some powerfull means of programming. They constitute a high level and abstract way of declarative programming. Functional programming languages like Haskell[PJ03], Clean[PvE95] or ML[MTH90][Ler97] provide build-in support for algebraic types. An algebraic type is defined by a set of constructors. In functional languages algebraic types can be directly declared; e.g. the following Haskell code defines a type of binary trees:

Tree.hs
```
data Tree a =
   Branch (Tree a) a (Tree a)
  |Empty
```

Algorithms on algebraic types are usually written as equations that match one of the constructor alternatives of the type. Thus the function that calculates the size of a binary tree can be written with the following two equations:

Tree.hs
```
size Empty          = 0
size (Branch l a r) = size l + 1 + size r
```

Common object oriented languages do not directly support algebraic types. Programming patterns need to be applied. The usual technique used in examples like the one above is the visitor pattern [GHJV95]. The algebraic type is expressed by a base class. The constructors are different subclasses of this. A visitor overloads an method `visit` for each of the subclasses representing a constructor. This overloaded method implements the pattern equations as seen in the Haskell example above.

The visitor pattern amounts in a lot of boilerplate code. The elegant four lines of Haskell in the example above result in at least four classes of Java. Therefor extensions to Java have been proposed, which directly support algebraic types, as e.g. *Pizza*[OW97], which has been around for quite a while. However, these extensions are rarely used in practice. One reason for this certainly is that programmers don't want to leave the official pure Java thread.

With Java 5 an annotation framework has been introduced to Java. Code can be annotated. These may not just be commentary annotations but an integral part of the program. Annotations may be evaluated by some processor. They open a door to language extensions without leaving the Java thread. Typical annotations are concerned with persistency code, transaction code or serialization code.

We will use annotions for generation of boilerplate code for the visitor pattern.

2 Annotations for Algebraic Types

We propose two annotations for algebraic types in Java. The annotation `@Data` for classes, which indicates that the class represents an algebraic type, and the annotation `@Constr`, which indicates that a method is a constructor case for the algebraic type.

Before we have a look at the actual implementation of the annotations we start with an example. With these two annotations binary trees can now be defined in Java as follows:

T.java

```
package example.tree;
import name.panitz.adt.*;
@Data class T<a> {
  @Constr void Branch(T<a> left,a element,T<a> right){};
  @Constr void Empty(){};
}
```

This looks quite similar to the implementation in Haskell. An algebraic type `T` is defined. This type is generic. It has a type variable for the node contents. Note that the class above is a normal Java class. It can be compiled with the standard Java compiler `javac`. However for generation of the boilerplate code we will need to use Java's annotation processing tool `apt`.

2.1 Algorithms on algebraic types

In the Haskell case algorithms over algebraic types could easily declared by pattern equations. We will generate a bunch of classes for the definition above. These provide an general visitor class `TVisitor`, a common base class `TAdt` and specialized classes for the constructors `Branch` and `Empty`. With these an algorithm over trees can be implemented as a specialized visitor. The `visit` function gets overloaded for the two constructor cases:

TSize.java

```
package example.tree;
public class TSize<a> extends TVisitor<a,Integer>{
  public Integer visit(Branch<a> x){
    return size(x.getLeft()) + 1 + size(x.getRight());}
  public Integer visit(Empty<a> _){return 0;}
```

A general function `size` was assumed, that calls the methods `welcome`. For some technical reason, we need a cast to the generated base class `TAdt`.

TSize.java

```
  public int size(T<a> t){return ((TAdt<a>)t).welcome(this);}
}
```

In fact, we are not far away from the Haskell implementation. In the next section we will reveal, how the processing of the two annotations `@Data` and `@Constr` has been implemented.

3 Implementation

In this section we give the complete implementation of the annotations for algebraic types in Java. There is no hidden code. In order to meet the size restriction of a conference paper, brevity is preferred over clarity in the code.

First of all, annotations are interfaces that need to be defined in Java.

Data.java

```
package name.panitz.adt;
public @interface Data{}
```

Constr.java

```
package name.panitz.adt;
public @interface Constr{}
```

Now we need a generator. Generators can be loaded by the `apt` tool through a factory class. The standard way to implement this is by way of a subclass of `AnnotationProcessorFactory`.

AdtGenFact.java

```
package name.panitz.adt;
import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import java.util.*;

public class AdtGenFact implements AnnotationProcessorFactory{
  static final String DATA_ANNOT   = "name.panitz.adt.Data";
  static final String CONSTR_ANNOT = "name.panitz.adt.Constr";

  public AnnotationProcessor getProcessorFor
    (Set<AnnotationTypeDeclaration> types
    ,AnnotationProcessorEnvironment env){
      return new AdtGen(types,env);
  }
  public Collection<String> supportedAnnotationTypes(){
    ArrayList<String> result= new ArrayList<String>();
    result.add(CONSTR_ANNOT);result.add(DATA_ANNOT);
    return result;
  }
  public Collection<String> supportedOptions(){
    return new ArrayList<String>();
  }
}
```

We decided for the most simple implementation, which simply returns a instance of the actual generator class.

AdtGen.java

```
package name.panitz.adt;
import com.sun.mirror.apt.*;
import com.sun.mirror.util.*;
import com.sun.mirror.declaration.*;
import java.util.*;
public class AdtGen extends SimpleDeclarationVisitor
                    implements AnnotationProcessor{
  Set<AnnotationTypeDeclaration> types;
  AnnotationProcessorEnvironment env;
  public AdtGen(Set<AnnotationTypeDeclaration> types
                ,AnnotationProcessorEnvironment env){
    this.types=types;
    this.env=env;
    constrDeclaration = (AnnotationTypeDeclaration) env
      .getTypeDeclaration("name.panitz.adt.Constr");
    dataDeclaration = (AnnotationTypeDeclaration) env
      .getTypeDeclaration("name.panitz.adt.Data");
  }
  public void process(){
   for (TypeDeclaration td :env.getSpecifiedTypeDeclarations()){
      td.accept(this);
```

```
   }
  }
```

The actual generator is implemented as an visitor over the declaration types in package `com.sun.mirror.declaration`.

AdtGen.java
```
  final private AnnotationTypeDeclaration constrDeclaration;
  final private AnnotationTypeDeclaration dataDeclaration;

  boolean hasAnnot(AnnotationTypeDeclaration a,Declaration m){
    for (AnnotationMirror am : m.getAnnotationMirrors())
      if(am.getAnnotationType().getDeclaration().equals(a))
        return true;
    return false;
  }

  public void visitClassDeclaration(ClassDeclaration d){
   if (hasAnnot(dataDeclaration,d)){
    ADT adt = new ADT(env,d);
    for (MethodDeclaration m:d.getMethods()){
      if (hasAnnot(constrDeclaration,m)){
        adt.addConstr(m.getSimpleName(),m.getParameters());
      }
    }
    adt.generateClasses();
  }
 }
}
```

Eventually we come to the actual generation of the boilerplate code.

3.1 representation of algebraic types

The main class `ADT` represents an algebraic type:

ADT.java
```
package name.panitz.adt;

import java.util.*;
import java.io.*;
import com.sun.mirror.declaration.*;
import com.sun.mirror.apt.*;

public class ADT {
```

It needs the following information:

- a name for the algebraic type.
- the package for the type.

- a list of constructors.
- furthermore we need the `AnnotationProcessorEnvironment`, which will give us a filer, for generating new source files.

ADT.java
```
  String name;
  String thePackage;
  public List<Constructor> constructors;
  TypeDeclaration cd;
  final Filer filer;
```

These fields get initialized in a simple constructor:

ADT.java
```
  public ADT
    (AnnotationProcessorEnvironment env,TypeDeclaration cd){
      thePackage = cd.getPackage().getQualifiedName();
      name=cd.getSimpleName();
      constructors=new ArrayList<Constructor>();
      this.cd=cd;
      filer=env.getFiler();
  }
```

We provide some auxiliary methods for retrieving the name of the type and the textual representation of the type parameters:

ADT.java
```
  public String getFullName(){return cd.toString();}
  public String getName(){return name;}
  String commaSepPs(){
    final String qN=getFullName();
    final int index=qN.indexOf(’<’);
    return index>=0?qN.substring(index+1,qN.length()-1):"";
  }
  public String getParamList(){
    return commaSepPs().length()==0?"":("<"+commaSepPs()+">");
  }
  String getPackageDef(){
    return  thePackage.length()==0?""
           :"package "+thePackage+";\n\n";
  }
```

The following method is used to add a further constructor to the algebraic type. The class `Constructor` will be defined as inner class of `ADT` below.

ADT.java
```
  void addConstr(String n,Collection<ParameterDeclaration> ps){
   constructors.add(new Constructor(n,ps));
}
```

Wie need to generate:

- a common base class for the type,
- a visitor interface,
- and for each constuctor a subclass of the base class.

ADT.java
```
  public void generateClasses(){
    try{
      generateClass();
      generateVisitorClass();
      for (Constructor c:constructors){c.generateClass(this);}
    }catch (IOException _){}
}
```

Generating the base class The base class extends the class, which has been annotated as `@Data` class. It has the same name with suffix `Adt`. We use the filer to create the source for the base class:

ADT.java
```
  public void generateClass() throws IOException{
    final String fullName = getFullName();
    Writer out
     = filer.createSourceFile(thePackage+"."+name+"Adt");
```

The generated class is abstract.

ADT.java
```
    out.write( getPackageDef());
    out.write("public abstract class ");
    out.write(getName()+"Adt"+getParamList());
    out.write(" extends "+fullName+"\n");
    out.write(" implements Iterable<Object>{\n");
```

It has the abstract method `welcome`. This is the method that welcomes a visitor.[1] It is an generic method. Its type variable, which we hard coded as `b_`, represents the result type of the visitor.

ADT.java
```
    out.write("  abstract public <b_> b_ welcome("
            +name+"Visitor<" + commaSepPs()
                            +(commaSepPs().length()==0?"":",")
                            +"b_> visitor);\n");
    out.write("}");
    out.close();
  }
```

[1] This could also be called `accept`.

Generating the visitor interface The next class that gets generated is the general visitor class. We decided for an abstract class, which has a abstract method `visit` for each constructor, and additionally a concrete general method `visit` overloaded for the base class. This ensures that cases, which were forgotten to be implemented will cause a runtime exception.

The visitor class is generic. It has the type variables of the base class and additionally a type variable for the result type of the methods `visit`.

ADT.java

```
  public void generateVisitorClass(){
    try{
      final String csName = name+"Visitor";
      final String fullName
       = csName+"<"+commaSepPs()
                 +(commaSepPs().length()==0?"":",")
                 +"result>";
      Writer out=filer.createSourceFile(thePackage+"."+csName);
      out.write( getPackageDef());
      out.write( "\n");
      out.write("public abstract class ");
      out.write(fullName+"{\n");
      for (Constructor c:constructors)
        out.write("  "+c.mkVisitMethod(this)+"\n");

      out.write("  public result visit("+getFullName()+" xs){");
      out.write("\n    throw new RuntimeException(");
      out.write("\"unmatched pattern: \"+xs.getClass());\n");
      out.write("  }");

      out.write("}");
      out.close();
    }catch (Exception _){}
  }
```

3.2 Generating constructor classes

Eventually we generate classes for each of the constructors in the algebraic type. Constructors are represented by an inner class of class `ADT`:

ADT.java

```
private class Constructor {
```

Constructors have a name and list of parameters:

ADT.java

```
  String name;
  Collection<ParameterDeclaration> params;
```

These get initialized during construction:

ADT.java

```
  public Constructor
                    (String n,Collection<ParameterDeclaration> ps){
    name=n;params=ps;}
```

We generate a class with the name of the constructor. It extends the base class and has the same type variables as the base class:

ADT.java

```
  public void generateClass(ADT theType){
    try{
      Writer out
        = filer.createSourceFile(theType.thePackage+"."+name);
      out.write( theType.getPackageDef());
      out.write("public class ");
      out.write(name);
      out.write(theType.getParamList());
      out.write(" extends ");
      out.write(theType.getName()+"Adt"+theType.getParamList());
      out.write("{\n");
```

In the body of the class we generate:

- fields for the arguments of the constructor.
- `Get`/`Set`-methods for these fields.
- the method `welcome`
- the method `equals`
- the method `toString`
- the method `iterator`

This is done in seperate methods:

ADT.java

```
      mkFields(out);
      mkConstructor(out);
      mkGetterMethods( out);
      mkSetterMethods( out);
      mkWelcomeMethod(theType, out);
      mkToStringMethod(out);
      mkEqualsMethod(out);
      mkIteratorMethod(out);
      out.write("}\n");
      out.close();
    }catch (Exception _){}
  }
```

Fields For every argument of the constructor we generate a private field in the class.

ADT.java
```
  private void mkFields(Writer out)throws IOException{
    for (ParameterDeclaration p:params){
     out.write("  private ");
     out.write(p.getType().toString()+" ");
     out.write(p.getSimpleName());
     out.write(";\n");
    }
  }
```

Constructor A single constructor is generated, which initializes the private fields.

ADT.java
```
  private void mkConstructor(Writer out)throws IOException{
    out.write("\n  public "+name+"(");
    boolean first= true;
    for (ParameterDeclaration p:params){
      if (!first){out.write(",");}
      out.write(p.getType().toString()+" ");
      out.write(p.getSimpleName());
      first=false;
    }
    out.write("){\n");
    for (ParameterDeclaration p:params){
      out.write("    this."+p.getSimpleName()+" = ");
      out.write(p.getSimpleName()+";\n");
    }
    out.write("  }\n\n");
  }
```

Get-methods For each private field a public get-method is generated.

ADT.java
```
  private void mkGetterMethods(Writer out)throws IOException{
    for (ParameterDeclaration p:params){
      out.write("  public ");
      out.write(p.getType().toString());
      out.write(" get");
      out.write(
       Character.toUpperCase(p.getSimpleName().charAt(0)));
      out.write(p.getSimpleName().substring(1));
      out.write("(){return "+p.getSimpleName() +";}\n");
    }
  }
```

Set-methods For each private field a public set-method is generated.

ADT.java
```
  private void mkSetterMethods(Writer out)throws IOException{
    for (ParameterDeclaration p:params){
```

```
      out.write("  public void set");
      out.write(
       Character.toUpperCase(p.getSimpleName().charAt(0)));
      out.write(p.getSimpleName().substring(1));
      out.write("(");
      out.write(p.getType().toString());
      out.write(" ");
      out.write(p.getSimpleName());
      out.write("){this."+p.getSimpleName());
      out.write("= "+p.getSimpleName()+";}\n");
    }
  }
```

Welcome method We generate the standard method `welcome`, which calls the method `visit` of the passed visitor with this object:

ADT.java
```
  private void mkWelcomeMethod(ADT theType,Writer out)
         throws IOException{
      out.write("  public <_b> _b welcome("
                +theType.name+"Visitor<"+theType.commaSepPs()
                      +(theType.commaSepPs().length()==0?"":",")
                                +"_b> visitor){"
                +"\n    return visitor.visit(this);\n  }\n");
  }
```

toString For the sake of comfort, we also generate the method `toString` in a natural way.

ADT.java
```
  private void mkToStringMethod(Writer out) throws IOException{
    out.write("  public String toString(){\n");
    out.write("    return \""+name+"(\"");
    boolean first=true;
    for (ParameterDeclaration p:params){
     if (first){first=false;}
     else out.write("+\",\"");
     out.write("+"+p.getSimpleName());
    }
    out.write("+\")\";\n  }\n");
  }
```

equals In the same way a standard method `equals` is generated:

ADT.java
```
  private void mkEqualsMethod(Writer out) throws IOException{
    out.write("  public boolean equals(Object other){\n");
    out.write("    if (!(other instanceof "+name+")) ");
    out.write("return false;\n");
    out.write("    final "+name+" o= ("+name+") other;\n");
```

```
    out.write("    return true  ");
    for (ParameterDeclaration p:params){
      out.write("&& "+p.getSimpleName()
                     +".equals(o."+p.getSimpleName()+")");
    }
    out.write(";\n  }\n");
  }
```

iterator In the same way a method `iterator` is generated. This enables us to apply the *scrap your boilerplate code*[LP03][LP04] pattern to the generated algebraic type.

ADT.java
```
  private void mkIteratorMethod(Writer out) throws IOException{
    out.write("  public java.util.Iterator<Object> iterator(){");
    out.write("\n    java.util.List<Object> res\n");
    out.write("         =new java.util.ArrayList<Object>();\n");
    for (ParameterDeclaration p:params){
      out.write("    res.add("+p.getSimpleName()+");\n");
    }
    out.write("    return res.iterator();\n");
    out.write("  }\n");
  }
```

the method visit For each constructor an abstract method `visit` had been generated in the general abstract visitor class. This was done by calls to the following method.

ADT.java
```
  public String mkVisitMethod(ADT theType){
    return "public abstract result visit("
           +name+theType.getParamList()+" _);";
  }
}}
```

4 Example: a tiny imperativ programming language

A typicall example application for the visitor pattern is compiler construction. The abstrct syntax tree of a programming language can easily be defines as algebraic type. The following `@Data` class defines an abstract syntax tree of statements in a tiny imperative programming language.

Klip.java
```
package name.panitz.adt.examples;
import name.panitz.adt.*;
import java.util.List;

abstract @Data class Klip implements Iterable<Object> {
  @Constr void Num(Integer i){};
```

```
  @Constr void Add(Klip e1,Klip e2){};
  @Constr void Mult(Klip e1,Klip e2){};
  @Constr void Sub(Klip e1,Klip e2){};
  @Constr void Div(Klip e1,Klip e2){};
  @Constr void Var(String name){};
  @Constr void Assign(String var,Klip e){};
  @Constr void While(Klip cond,Klip body){};
  @Constr void Block(List<Klip> stats){};
}
```

4.1 Show

The first visitor we present for this type, represents it as a `String`. The corresponding visitor can be written as:

ShowKlip.java

```
package name.panitz.adt.examples;
import name.panitz.*;
import java.util.*;

public class ShowKlip extends KlipVisitor<String> {
  public String show(Klip a){return ((KlipAdt)a).welcome(this);}

  public String visit(Num x){return x.getI().toString();}
  public String visit(Add x){
    return "("+show(x.getE1())+" + "+show(x.getE2())+")";}
  public String visit(Sub x){
    return "("+show(x.getE1())+" - "+show(x.getE2())+")";}
  public String visit(Div x){
    return "("+show(x.getE1())+" / "+show(x.getE2())+")";}
  public String visit(Mult x){
    return "("+show(x.getE1())+" * "+show(x.getE2())+")";}
  public String visit(Var v){return v.getName();}
  public String visit(Assign x){
    return x.getVar()+" := "+show(x.getE());}
  public String visit(Block b){
    StringBuffer result=new StringBuffer();
    for (Klip x:(List<Klip>)b.getStats())
      result.append(show(x)+";\n");
    return result.toString();
  }
  public String visit(While w){
    StringBuffer result=new StringBuffer("while (");
    result.append(show(w.getCond()));
    result.append("){\n");
    result.append(show(w.getBody()));
    result.append("\n}");
    return result.toString();
  }
}
```

This will e.g. print a program for calculating the faculty as follows:

fak.klip

```
x := 5;
y:=1;
while (x){y:=y*x;x:=x-1;};
y;
```

4.2 Evaluation visitor

A further visitor defines an interpreter for the tiny imperative language. This visitor has an internal map for binding of the variables used in the program:

EvalKlip.java

```
package name.panitz.adt.examples;
import name.panitz.*;
import java.util.*;

public class EvalKlip extends KlipVisitor<Integer> {
 Map<String,Integer> env = new HashMap<String,Integer>();
 public Integer v(Klip x){return ((KlipAdt)x).welcome(this);}

 public Integer visit(Num x){return x.getI();}
 public Integer visit(Add x){return v(x.getE1())+v(x.getE2());}
 public Integer visit(Sub x){return v(x.getE1())-v(x.getE2());}
 public Integer visit(Div x){return v(x.getE1())/v(x.getE2());}
 public Integer visit(Mult x){return v(x.getE1())*v(x.getE2());}
 public Integer visit(Var v){return env.get(v.getName());}
 public Integer visit(Assign ass){
   Integer i = v(ass.getE());
   env.put(ass.getVar(),i);
   return i;
 }
 public Integer visit(Block b){
    Integer result = 0;
    for (Klip x:(List<Klip>)b.getStats()) result=v(x);
    return result;
 }
 public Integer visit(While w){
    Integer result = 0;
    while (v(w.getCond())!=0){
      System.out.println(env);
      result = v(w.getBody());
    }
    return result;
 }
}
```

4.3 Generic queries

Since we implemented the interface `Iterable<Object>` for all node classes, we can apply the *scrap your boilerplate* pattern.[Pan05]. Arbitrary tree traversal code can be implemented through subclasses of `Query` or `Transform`. The following simple example counts the number literals within an abstract syntax tree.

CountNumConstants.java

```
package name.panitz.adt.examples;
import name.panitz.boilerplate.Query;
class CountNumConstants extends Query<Integer>{
  public Integer eval(Object x){
    if (x instanceof Num) return 1;
    return 0;
  }
  public Integer eval(Integer x,Integer y){return x+y;}
}
```

A `javacc` parser definition for the tiny imperative language can be found at the corresponding website to this paper (`http://panitz.name/paper/adt`) .

5 Conclusion

We have presented an example of how Java annotations can be used to generate boilerplate code for a usefull programming pattern. The implementation of the code generator was fairly easy and could be printed out completely within a conference paper. The annotations extend Java with some new high level programming construct: a declarative way of defining algebraic data types. While we achieved this, we did not even exploit the full power of Java annotations.

A drawback to our solutions is, that it is not possible to add features to existing classes other than by way of subtyping. Instead of creating a new base class `KlipAdt` we would have liked to change class `Klip`. This amounts in casts from the defining class to the generated base class. It does not seem to be possible, to change existing classes.

Generation of classes with the `apt` api is (still) somewhat clumsy. A writer is used. There are no classes, which help to construct new source code.

It turns out that annotations offer a new way of programming: instead of switching to newer powerfull languages, which need an seperate compiler, the language can be extended with new features. We expect to see quite a number of such extensions to be come up in the future. A lot of common patterns seem to be candidates for generation of boilerplate code driven by annotations.

However, this mechanism seems to be limited. It might be interesting future work, to investigate how far we can go. Is it possible to express some real pattern matching expression in terms of Java annotations?

References

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[Ler97] Xavier Leroy. The Caml Light system release 0.73. Institut National de Recherche en Informatique et Automatique, 1 1997.

[LP03] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[LP04] Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2004)*, pages 244–255. ACM Press, 2004.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. IT Press, Cambridge, Massachusetts, 1990.

[OW97] Martin Odersky and Philip Wadler. Pizza into java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, 1997.

[Pan05] Sven Eric Panitz. The Scrap Your Boilerplate Pattern in Java. Draft; Online since March 2005 `panitz.name/paper/boilerplate/index.html`, 3 2005.

[PJ03] Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[PvE95] R. Plasmeijer and M. van Eekelen. Concurrent clean: Version 1.0. Technical report, Dept. of Computer Science, University of Nijmegen, 1995. draft.