

EP – An Appetizer

Karl Stroetmann

December 14, 2005

Abstract

EP is an acronym for *equational programming*. Equational programming is a programming paradigm that specifies functions via conditional equations. EP is both a programming language for equational programming and a translator that generates Java classes from conditional equations. The fact that EP generates Java makes it easy to combine equational programs with ordinary Java programs and furthermore guarantees an efficient execution of equational programs.

1 Introduction

By specifying methods through conditional equations, equational programming lifts the abstraction level at which software can be developed. For typical problems, equational programs are a fraction of the size of conventional object oriented programs solving the same problem. However, equational programming is not for everybody and it is not for every problem. Equational programming is well suited for problems that can easily be described via conditional equations. It is well suited for programmers who are not scared by mathematical notations and concepts.

Section 2 shows how EP can be installed and Section 3 motivates equational programming by an example. Section 4 discusses matching and Section 5 clarifies the structure of equational programs. The reader should not be deterred by the fact that this paper extends over 15 pages. About half of these pages is accounted for because I have included the output produced by EP when translating the conditional equations specifying symbolic differentiation. By inspecting the classes generated by EP the idea underlying equational programming is readily understood.

2 Installing EP

To install EP, download the file

<http://www.ba-stuttgart.de/stroetmann/EP/ep.tar.gz>

EP is implemented in Java 5.0 and should therefore run with any operating system but the following description assumes that you are working with either Linux or Unix. Let us assume that there is a directory called **Software** which is a subdirectory of your home directory and that EP should be installed into this directory. If it is to be installed somewhere else, simply change every occurrence of the string “~/Software” in the following description accordingly.

1. Start a shell in the directory containing the file “ep.tar.gz” and move the file “ep.tar.gz” to the directory ~/Software by issuing the command:

```
mv ep.tar.gz ~/Software
```

2. Change to the directory ~/Software

```
cd ~/Software
```

3. Unzip the file ep.tar.gz

```
gunzip ep.tar.gz
```

This command should create the file `ep.tar`.

4. Untar the file `ep.tar`:

```
untar xf ep.tar
```

This command should produce the directory `EP` as a subdirectory of the directory `~/Software`. The directory `EP` contains three subdirectories:

- (a) `src` contains the Java sources together with the class files.
- (b) `doc` contains the documentation.
- (c) `examples` contains the example file “`expr.ep`” that specifies symbolic differentiation using equational programming.

The directory `EP` also contains the file `antlr-2.7.5.jar`. This file contains ANTLR [PQ95], which is a parser generator written by Terence Parr.¹ There are two more files in the directory `EP`: The file “`copyright.txt`” informs you that `EP` is subject to the *Gnu General Public License*. For your convenience, this license is reproduced in the file “`gpl.txt`”.

5. To test `EP`, change to the directory “`examples`” and start `EP` using the commands:

```
cd ~/Software/EP/examples
export CLASSPATH=~/Software/EP/antlr-2.7.5.jar:~/Software/EP/src
java EP expr
```

This invocation of `EP` will generate a number of Java files implementing symbolic differentiation. To test these generated files, first compile them using the command

```
javac *.java
```

Next, execute the command

```
java Main
```

This should produce the result

```
Product(Variable(x), Variable(x))/dx =
    Sum(Product(Number(1.0), Variable(x)),
        Product(Variable(x), Number(1.0)))
```

stating that for a variable `x` the following holds:

$$\frac{d}{dx}(x * x) = 1 * x + x * 1.$$

3 Motivation

When I first encountered the programming language `PROLOG` one of my first attempts was a program for the symbolic differentiation of arithmetic expressions. I had written a similar program in *Pascal* before and I was fascinated how easy things could be done in `PROLOG`. The `PROLOG` program was not only about a fifth of the size of the *Pascal* program but was also much clearer since it reflected the mathematical rules for differentiation very neatly. Later, when I developed a theorem prover using `PROLOG`, my initial enthusiasm was relativized. There were two main problems that made me stumble. First, `PROLOG` lacks the efficiency of modern object oriented languages. Secondly, `PROLOG` is not typed and this fact impedes the progress of large software projects. Therefore, on later projects I was forced to use *C++* and Java. However, there were times when I wished `PROLOG` back. By that time I had also come to know *abstract state machines* [BS03]. This is a method for specifying hardware and software that works by giving a concise description of the operational semantics of these systems via *rules*. Often, these rules can be

¹ANTLR is very useful software. If you are interested, you can download the current release together with documentation from <http://www.antlr.org>.

interpreted as conditional equations.² So I thought about the possibility of translating conditional equations into Java classes and found it to be quite easy. I will explain the main idea using symbolic differentiation as an example. To this end, let us define *arithmetical expression* by induction:

1. A string of decimal digits that can be interpreted as a number is an arithmetical expression.
2. A string consisting of letters is an arithmetical expression. An arithmetical expression of this kind denotes a variable.
3. If l and r are arithmetical expressions, then the strings $l+r$, $l-r$, $l*r$, and l/r are arithmetical expressions denoting, respectively, the sum, the difference, the product, and the quotient of l and r .
4. If t is an arithmetical expression, then the string (t) is an arithmetical expression.

Examples of arithmetical expressions would be the string “ x ” denoting a variable, the string “12.3” denoting a floating point number, or the string “ $x+y*12.3$ ” denoting a complex arithmetical expressions. As a final example, the string “ $(x+y)*12.3$ ” is another arithmetical expression.

How can arithmetical expressions be represented in Java? An object oriented approach would define an abstract class **Expr** that comprises all arithmetical expression. We would then have concrete classes corresponding to the different clauses of the inductive definition of arithmetical expressions. These classes would be derived from the abstract class **Expr**.

1. There would be a class *Number*. This class would contain a member variable of type **Double** containing the actual value.
2. There would be a class *Variable*. This class would contain a member variable of type **String** giving the name of the variable.
3. There would be a class *Sum*. This class would contain to member variables **mLhs** and **mRhs**, each of type *Expr*. This class would denote arithmetical expressions of the form **mLhs+mRhs**.
4. Similarly, there would be classes *Difference*, *Product*, and *Quotient*, all containing member variables **mLhs** and **mRhs** and denoting the difference, product, and quotient of **mLhs** and **mRhs**, respectively.

The curious reader is advised to consult the figures 2, 3, 4, 5, 6, 7, and 8, which are given on the pages 5, 6, 7, 8, 9, 10, and 11, respectively. The classes shown in these figures have actually been generated from the specification shown in Figure 1 on page 4 using EP. We discuss this specification now line by line.

1. The lines 1 – 6 contain a *recursive type definition*. The definition

$$\text{Expr} = \text{Number}(\text{Double value}) + \dots + \text{Quotient}(\text{Expr lhs}, \text{Expr rhs});$$

specifies that the abstract class **Expr** is the parent of the concrete classes **Number**, \dots , **Quotient**. Furthermore, this type definition specifies the member variables of these concrete classes. For example, since in the recursive type definition **Number** appears as

$$\text{Number}(\text{Double value})$$

the class **Number** has one member variable. The type of this member variable is **Double** and the name is derived from from the string “value” by capitalizing this string and adding the letter “m” in front, so the actual name becomes “**mValue**”.

Similarly, since **Quotient** appears as

$$\text{Quotient}(\text{Expr lhs}, \text{Expr rhs}),$$

the class **Quotient** is comprised of two member variables with the names “**mLhs**” and “**mRhs**”. Both of these member variables have the type **Expr**.

²I need to caution the reader knowing abstract state machines. That method is much more powerful than equational programming. At some later stage, if time permits, I intend to extend EP to allow for ASM rules also. However, currently this is just wishful thinking.

```

1 Expr = Number(Double value)
2       + Variable(String name)
3       + Sum(Expr lhs, Expr rhs)
4       + Difference(Expr lhs, Expr rhs)
5       + Product(Expr lhs, Expr rhs)
6       + Quotient(Expr lhs, Expr rhs);
7
8 diff: Expr * String -> Expr;
9
10 Number(value).diff(x) = Number(0.0);
11
12 v.equals(x) -> Variable(v).diff(x) = Number(1.0);
13 Variable(v).diff(x) = Number(0.0);
14
15 Sum(l, r).diff(x) = Sum(l.diff(x), r.diff(x));
16 Difference(l, r).diff(x) = Difference(l.diff(x), r.diff(x));
17 Product(l, r).diff(x) =
18     Sum(Product(l.diff(x), r), Product(l, r.diff(x)));
19 Quotient(l, r).diff(x) =
20     Quotient( Difference(Product(l.diff(x), r), Product(l, r.diff(x))),
21              Product(r, r) );

```

Figure 1: The specification of arithmetical expressions.

2. Line 8 specifies the *signature* of the method `diff()`. In conventional mathematical notation it would be written as

$$\text{diff}: \text{Expr} \times \text{String} \rightarrow \text{Expr}.$$

In EP, the symbol “ \times ” is replaced by the string “`*`” and the symbol “ \rightarrow ” is replaced by the string “`->`”. The signature given above states that the method `diff()` takes an arithmetical expression and the name of a variable as arguments and that the result produced by this method is of type `Expr`. Of course, the idea is that `e.diff(v)` differentiates the expression `e` with respect to the variable `v`.

3. The lines 10 – 21 give the specification of the method `diff()` in the form of conditional equations.

- (a) Line 10 specifies that the derivative of a constant is the number 0. This line is translated into the lines 7 – 9 in Figure 3 on page 6.
- (b) When a variable `v` is differentiated with respect to another variable `x`, there are two cases: If both variables are equal, then the result is the number 1, otherwise the result is 0. The first case is dealt with in line 12. The expression `Variable(v)` denotes the invocation of the constructor `Variable` with the argument `v`. This invocation constructs a variable with the name `v`. If this name is equal to `x`, then the result is the number 1. This is tested using the method `equals()` which is defined on strings. If this test fails, then the next equation is used to yield the result 0.

The following is important: If a method is specified by several conditional equations, then when evaluating a method invocation the equations are tried in the order in which they appear in the specification. A conditional equation is *applicable* if, first, its left hand side matches the expression to be evaluated and, secondly, the condition evaluates to `true`. The first applicable conditional equation is used to evaluate a method invocation.

- (c) The product rule for differentiating a product is given as

$$\frac{d(l * r)}{dx} = \frac{dl}{dx} * r + l * \frac{dr}{dx}.$$

This rule is implemented in the equation given in line 17 and 18 of Figure 1 on page 10. The term

`Product(l, r)`

denotes the product $l * r$. The term

`Sum(Product(l.diff(x), r), Product(l, r.diff(x)))`

denotes the result $\frac{dl}{dx} * r + l * \frac{dr}{dx}$. The equation is translated into line 9 – 12 in Figure 7 on page 10.

- (d) The remaining rules are written in the same spirit.

We discuss the generated Java classes in more detail below. First, Figure 2 on page 5 shows the implementation of the abstract class `Expr`. This class declares two abstract methods. One of these methods is the method `diff()`, which had been declared in line 8 of Figure 1. The second abstract method is the method `equals()`. This method is always declared in classes generated by EP. The intention is that this method compares two objects for equality in the *structural* sense. Two objects are equal in the structural sense if they have the same type and, furthermore, their member variables reference objects that are equal in the structural sense. In other words, objects are equal in the structural sense if they are made up from the same components. In a moment, we will clarify this notion by discussing an example.

```

1 public abstract class Expr {
2     public abstract Expr diff(String x1);
3     public abstract Boolean equals(Expr rhs);
4 }

```

Figure 2: The abstract class `Expr`.

Next, we discuss the concrete class `Number` that is shown in Figure 3 on page 6. The relevant part of the recursive type definition that specifies the structure of this class is

`Expr = Number(Double value) + ...`.

This equation specifies everything in the class `Number` with the exception of the implementation of the method `diff()`. We discuss the code in Figure 3 line by line:

1. Line 1 states that `Number` is a subclass of `Expr`. The left hand side of the recursive type definition gives the name of the abstract class form which the different concrete classes are derived.
2. Line 2 defines the member variable `mValue` and specifies its type as `Double`. This is part of the translation of the string “`Double value`” that was given as argument to the type name `Number` in the recursive type definition. The names of member variables are generated by first capitalizing the string specifying the variable and then adding the character “`m`” in front, so “`value`” is turned into “`mValue`”.
3. Line 4 – 6 give the constructor of `Number`. In general, the constructor of a concrete type has the same number of arguments as there are member variables. It initializes every member variable with the corresponding argument.
4. Line 7 – 9 give the implementation of the method `diff`. Since differentiating a constant yields 0 as the result, the method returns a new object of type `Number` that represents the number 0. This is a direct translation of the conditional equation

```

1 public class Number extends Expr {
2     private Double mValue;
3
4     public Number(Double value) {
5         mValue = value;
6     }
7     public Expr diff(String x) {
8         return new Number(0.0);
9     }
10    public Boolean equals(Expr rhs) {
11        if (!(rhs instanceof Number)) {
12            return false;
13        }
14        Number r = (Number) rhs;
15        if (!mValue.equals(r.mValue)) {
16            return false;
17        }
18        return true;
19    }
20    public Double getValue() {
21        return mValue;
22    }
23    public String toString() {
24        return "Number(" + mValue.toString() + ")";
25    }
26 }

```

Figure 3: The class `Number`.

```
Number(value).diff(x) = Number(0.0);
```

for differentiating numbers.

- Line 10 to 19 give the implementation of the method `equals()` that compares `this` with an object `rhs` structurally. First, we check in line 11 whether `rhs` has the same type as `this` because objects of different types can not be equal. Once we know that `rhs` is an object of type `Number` we can cast it to `Number` in line 14. This enables us to compare the components of `this` and `rhs` in line 15: These two objects have just one component, which is `mValue`. If the components are different, we return `false`, otherwise we return `true`.

Of course, in the case of the class `Number` the implementation of `equals()` could have been simplified by replacing line 15 – 18 with the single line

```
return mValue.equals(r.mValue);
```

However, the approach that was used in the implementation of `Number` easily extends to the case when there are multiple member variables.

- Line 20 – 22 implements the method `getValue()` that makes the value of the member variable `mValue` available. The name of this method has been generated from the string “value” by capitalizing this string and adding the string “get” in front. In general, EP generates a get-method for every member variable.
- Finally, line 23 – 25 gives the implementation of the method `toString()`. This method provides a readable string representation of objects of class `Number` and is mainly intended for debugging purposes.

The remaining classes are built in a similar way. We have given these classes for the sake of completeness but there is no need to discuss them in detail since their implementation is completely analogous to the implementation of the class *Number*. The only difference is that the classes *Sum*, *Difference*, *Product*, and *Quotient* contain two member variables. For example, the part of the recursive type definition specifying the concrete class *Sum* is

$$\text{Expr} = \dots + \text{Sum}(\text{Expr lhs}, \text{Expr rhs}) + \dots$$

Therefore, *Sum* has two member variables *mLhs* and *mRhs*, both of type *Expr*. Accordingly, the constructor takes two arguments and there are two get-methods. Also, the implementation of the method *equals()* gets longer since we now have to compare two components.

```
1 public class Variable extends Expr {
2     private String mName;
3
4     public Variable(String name) {
5         mName = name;
6     }
7     public Expr diff(String x) {
8         if (mName.equals(x)) {
9             return new Number(1.0);
10        }
11        return new Number(0.0);
12    }
13    public Boolean equals(Expr rhs) {
14        if (!(rhs instanceof Variable)) {
15            return false;
16        }
17        Variable r = (Variable) rhs;
18        if (!mName.equals(r.mName)) {
19            return false;
20        }
21        return true;
22    }
23    public String getName() {
24        return mName;
25    }
26    public String toString() {
27        return "Variable(" + mName.toString() + ")";
28    }
29 }
```

Figure 4: The class *Variable*.

```

1  public class Sum extends Expr {
2      private Expr mLhs;
3      private Expr mRhs;
4
5      public Sum(Expr lhs, Expr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public Expr diff(String x) {
10         return new Sum(mLhs.diff(x), mRhs.diff(x));
11     }
12     public Boolean equals(Expr rhs) {
13         if (!(rhs instanceof Sum)) {
14             return false;
15         }
16         Sum r = (Sum) rhs;
17         if (!mLhs.equals(r.mLhs)) {
18             return false;
19         }
20         if (!mRhs.equals(r.mRhs)) {
21             return false;
22         }
23         return true;
24     }
25     public Expr getLhs() {
26         return mLhs;
27     }
28     public Expr getRhs() {
29         return mRhs;
30     }
31     public String toString() {
32         return "Sum(" + mLhs.toString() + ", " + mRhs.toString() + ")";
33     }
34 }

```

Figure 5: The class Sum.

```

1  public class Difference extends Expr {
2      private Expr mLhs;
3      private Expr mRhs;
4
5      public Difference(Expr lhs, Expr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public Expr diff(String x) {
10         return new Difference(mLhs.diff(x), mRhs.diff(x));
11     }
12     public Boolean equals(Expr rhs) {
13         if (!(rhs instanceof Difference)) {
14             return false;
15         }
16         Difference r = (Difference) rhs;
17         if (!mLhs.equals(r.mLhs)) {
18             return false;
19         }
20         if (!mRhs.equals(r.mRhs)) {
21             return false;
22         }
23         return true;
24     }
25     public Expr getLhs() {
26         return mLhs;
27     }
28     public Expr getRhs() {
29         return mRhs;
30     }
31     public String toString() {
32         return "Difference(" + mLhs.toString() + ", " + mRhs.toString() + ")";
33     }
34 }

```

Figure 6: The class Difference.

```

1  public class Product extends Expr {
2      private Expr mLhs;
3      private Expr mRhs;
4
5      public Product(Expr lhs, Expr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public Expr diff(String x) {
10         return new Sum(new Product(mLhs.diff(x), mRhs),
11                         new Product(mLhs, mRhs.diff(x)));
12     }
13     public Boolean equals(Expr rhs) {
14         if (!(rhs instanceof Product)) {
15             return false;
16         }
17         Product r = (Product) rhs;
18         if (!mLhs.equals(r.mLhs)) {
19             return false;
20         }
21         if (!mRhs.equals(r.mRhs)) {
22             return false;
23         }
24         return true;
25     }
26     public Expr getLhs() {
27         return mLhs;
28     }
29     public Expr getRhs() {
30         return mRhs;
31     }
32     public String toString() {
33         return "Product(" + mLhs.toString() + ", " + mRhs.toString() + ")";
34     }
35 }

```

Figure 7: The class Product.

```

1  public class Quotient extends Expr {
2      private Expr mLhs;
3      private Expr mRhs;
4
5      public Quotient(Expr lhs, Expr rhs) {
6          mLhs = lhs;
7          mRhs = rhs;
8      }
9      public Expr diff(String x) {
10         return new Quotient(new Difference(new Product(mLhs.diff(x), mRhs),
11             new Product(mLhs, mRhs.diff(x))), new Product(mRhs, mRhs));
12     }
13     public Boolean equals(Expr rhs) {
14         if (!(rhs instanceof Quotient)) {
15             return false;
16         }
17         Quotient r = (Quotient) rhs;
18         if (!mLhs.equals(r.mLhs)) {
19             return false;
20         }
21         if (!mRhs.equals(r.mRhs)) {
22             return false;
23         }
24         return true;
25     }
26     public Expr getLhs() {
27         return mLhs;
28     }
29     public Expr getRhs() {
30         return mRhs;
31     }
32     public String toString() {
33         return "Quotient(" + mLhs.toString() + ", " + mRhs.toString() + ")";
34     }
35 }

```

Figure 8: The class Quotient.

4 Matching

The conditional equations presented up to now have only used matching in a very restricted way. So far, the equations specifying a method $m()$ had the form

$$c \rightarrow C(x_1, \dots, x_m).m(y_1, \dots, y_k) = r,$$

where x_1, \dots, x_m and y_1, \dots, y_k denote variables. However, EP works even if the variables x_1, \dots, x_m are replaced by more complex expressions. To demonstrate the possibilities we start with an example. We extend the example given in the previous section where we have developed a program for symbolic differentiation. Given the input $x * x$, the method $diff()$ yields the output $1 * x + x * 1$. This can obviously be simplified. To this end, we add a method

$$simplify : Expr \rightarrow Expr$$

to the classes implemented so far. Figure 9 on page 12 shows how this method can be specified via conditional equations. We discuss these equations line by line.

```
1  simplify: Expr -> Expr;
2
3  Number(c).simplify() = Number(c);
4
5  Variable(v).simplify() = Variable(v);
6
7  Sum(Number(0.0), r).simplify() = r;
8  Sum(1, Number(0.0)).simplify() = 1;
9  Sum(1, r).simplify() = Sum(1, r);
10
11 Difference(1, Number(0.0)).simplify() = 1;
12 Difference(1, r).simplify() = Difference(1, r);
13
14 Product(Sum(x, y), z).simplify() = Sum( Product(x, z), Product(y, z) );
15 Product(z, Sum(x, y)).simplify() = Sum( Product(z, x), Product(z, y) );
16 Product(Number(1.0), r).simplify() = r;
17 Product(1, Number(1.0)).simplify() = 1;
18 Product(Number(0.0), r).simplify() = Number(0.0);
19 Product(1, Number(0.0)).simplify() = Number(0.0);
20 Product(1, r).simplify() = Product(1, r);
21
22 Quotient(Number(0.0), r).simplify() = Number(0.0);
23 Quotient(1, r).simplify() = Quotient(1,r);
```

Figure 9: The specification of the method $simplify()$.

1. Since we can not simplify a number, the corresponding equation in line 3 returns the object unmodified. In this equation, the object on which the method $simplify()$ is invoked is given as $Number(c)$. This expression is interpreted as denoting a number with value c . Here, c is a variable denoting an arbitrary value of type `Double`. When this rule is later applied to an object of type `Number`, then c is bound to the current value of the member variable `mValue`. The expression $Number(c)$ on the right hand side of the equation is interpreted as a constructor call. Therefore this equation will be translated into the method

```
public Expr simplify() {
    return new Number(mValue);
}
```

2. Simplifying a variable yields the same variable, therefore the situation in line 5 is the same as in line 3 and will not be discussed further.
3. If we have a sum of the form $0 + x$, then we can simplify this expression to x . Similarly, an expression of the form $x + 0$ is simplified into x . Finally, we need a catch-all clause stating that in any other case the expression $x + y$ remains unmodified. All this is specified by the three equations given in line 7 – 9. These lines are translated into the following code

```

public Expr simplify() {
    if (mLhs.equals(new Number(0.0))) {
        return mRhs;
    }
    if (mRhs.equals(new Number(0.0))) {
        return mLhs;
    }
    return new Sum(mLhs, mRhs);
}

```

4. We discuss the simplification of products next. The lines 14 – 19 specify the following rules.

- (a) $(x + y) * z = x * z + y * z$,
- (b) $z * (x + y) = z * x + z * y$,
- (c) $1 * r = r$,
- (d) $l * 1 = l$,
- (e) $0 * r = 0$,
- (f) $l * 0 = 0$.

Finally, there is a catch-all clause in line 20 stating that any product that can not be simplified by the rules given above is left unchanged. The last four rules are similar to the rules already seen for the simplification of sums. However, the nature of the first two rules describing the laws of distribution is different: Let us inspect the left hand side of the first rule closer. This left hand side is

```
Product(Sum(x, y), z).simplify()
```

Here the constructor `Product` contains the subterm `Sum(x, y)`. This subterm contains the variables x and y . When an object of type `Product` is such that the component `mLhs` is of type `Sum`, then the components of this `Sum` are assigned to the variables x and y . Figure 10 on page 14 shows the translation of these rules into Java code.

5. The remaining code is similar and will not be discussed.

The example given above does not really work for for simplifying arithmetic expressions. In reality, two methods are needed that call each other recursively. The file “`expr-diff.ep`” in the directory “`EP/examples`” contains rules defining two methods `simplify()` and `reduce()`. However, since the discussion of algebraic simplification is not the topic of this paper I have decided to give only a simplified exposition here.

5 Equational Specifications

We proceed to give a more formal treatment of equational programs. An equational program consists of a set of *type definitions*. Each type definition consists of three items:

```

1 public Expr simplify() {
2     if ((mLhs instanceof Sum)) {
3         return new Sum(new Product(((Sum) mLhs).getLhs(), mRhs),
4                         new Product(((Sum) mLhs).getRhs(), mRhs));
5     }
6     if ((mRhs instanceof Sum)) {
7         return new Sum(new Product(mLhs, ((Sum) mRhs).getLhs()),
8                         new Product(mLhs, ((Sum) mRhs).getRhs()));
9     }
10    if (mLhs.equals(new Number(1.0))) {
11        return mRhs;
12    }
13    if (mRhs.equals(new Number(1.0))) {
14        return mLhs;
15    }
16    if (mLhs.equals(new Number(0.0))) {
17        return new Number(0.0);
18    }
19    if (mRhs.equals(new Number(0.0))) {
20        return new Number(0.0);
21    }
22    return new Product(mLhs, mRhs);
23 }

```

Figure 10: The implementation *simplify()* for the class *Product*.

1. A *type sum* defines the structure of the generated *Java* classes. The general form is

$$\text{AbstractType} = \text{TypeConstructor}_1 + \dots + \text{TypeConstructor}_n;$$

Here *AbstractType* is a string that satisfies the constraints for a *Java* identifier: The string has to consist of letters, digits and underscores and has to start with a letter. This string is the name of the abstract type that is generated. Each *TypeConstructor* specifies the name and member variables of a concrete subtype. We have already seen an example of a type sum:

$$\text{Expr} = \text{Number}(\text{Double value}) + \dots + \text{Quotient}(\text{Expr lhs}, \text{Expr rhs});$$

The general form of a *TypeConstructor* is

$$\text{ClassName}(\text{Type}_1 \text{ varName}_1, \dots, \text{Type}_m \text{ varName}_m)$$

Here *ClassName* is a *Java* identifier, *Type_i* is a *Java* type and *varName_i* is a *Java* identifier for $i = 1, \dots, m$. This *TypeConstructor* generates a *Java* class with name *ClassName* that has m member variables with the types *Type_i*. The name of these variables is constructed by capitalizing the string *varName_i* and adding the letter “m” at the front of the resulting string. An example of a type constructor is

$$\text{Quotient}(\text{Expr lhs}, \text{Expr rhs}).$$

This type constructor generates a class *Quotient* that has two member variables “mLhs” and “mRhs”. Both of these member variables are of type *Expr*.

2. A signature has the form

$$\text{method} : \text{AbstractType} * \text{ArgType}_1 * \dots * \text{ArgType}_k \rightarrow \text{ResultType}$$

Here, *method* is a *Java* identifier that serves as the name of the method that is to be specified on *AbstractType*. For $i = 1, \dots, k$ the string *ArgType_i* denotes the type of the i -th argument of the method. For example, the signature

`diff: Expr * String -> Expr`

specifies a method `diff()` that is defined on the abstract type `Expr`. Besides the implicit argument `this` of type `Expr`, the method takes an argument of type `String`.

3. A conditional equation has the form

`cond -> lhs = rhs;`

Here `cond` is a *Java* expression producing either `true` or `false`. It is referred to as the condition. `lhs` is referred to as the *left hand side* of the equation. It is required to be of the form

`TypeConstructor(s1, ..., sm).method(y1, ..., yk)`

Here `TypeConstructor` is the name of a subtype of the abstract type given in the signature of `method`. The arguments `s1, ..., sm` are *Java* expressions that are interpreted as arguments of a constructor of this subtype, while `y1, ..., yk` are *Java* identifiers denoting variables. Finally, `rhs` is a *Java* expression that is referred to as the *right hand side* of the equation. It gives the result of the method invocation. The conditional equation

`cond -> TypeConstructor(s1, ..., sm).method(y1, ..., yk) = rhs;`

is interpreted as follows: If an object `o` has been constructed via

`o = new TypeConstructor(s1, ..., sn),`

and if `cond` yields `true`, then the invocation

`o.m(y1, ..., yk)`

produces the result `rhs`.

In a conditional equation, the condition `cond` is optional. If it missing, then the string “->” is also missing and the form of the equation is just

`lhs = rhs;`

An example of a conditional equation is

`v.equals(x) -> Variable(v).diff(x) = Number(1.0);` (1)

It specifies how variables are differentiated. The other conditional equation specifying the differentiation of variables is

`Variable(v).diff(x) = Number(0.0);` (2)

Of course, the order in which these equations appear is important. If the order of these two equations were exchanged, then equation (1) could never be used. It is possible to combine equations (1) and (2) into a single equation:

`Variable(v).diff(x) = (v.equals(x) ? Number(1.0) : Number(0.0));`

This example shows that the right hand side of an equation can contain all operators that are available for constructing *Java* expressions.

6 Outlook

I have ambitious plans for the development of EP. First, I would like to make those set theoretic constructs that have been pioneered in *Setl* [SDSD86] available. Second, my personal experience with C++ [Str00] has shown me that operator overloading can increase the readability of programs substantially. And finally, I would like to incorporate some ideas from abstract state machines [BS03]. Of course, this is all very much work in progress. It will benefit from your suggestions. Please mail them to `stroetmann@ba-stuttgart.de`.

References

- [BS03] E. Börger and R. F. Stärk. *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
- [PQ95] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software — Practice and Experience*, 25(7):789–810, 1995.
- [SDSD86] J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky. *Programming With Sets: An Introduction to SETL*. Springer-Verlag New York, Inc., New York, NY, USA, 1986.
- [Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.