Data Modeling

1

Look at JavaParser's PrimitiveType.Primitive enum. Express it as a sum type

Answer

```
BOOLEAN + BYTE + CHAR + DOUBLE + FLOAT + INT + LONG + SHORT
    (= 1+1+1+1+1+1+1+1 = 8)
```

2

Consider Javaparser's PrimitiveType class. Express the valid values as an algebraic data type. It may help you to look at the constructors.

Answer

```
PrimitiveType = PrimitiveType.Primitive * (List AnnotationExpr)
```

(The `(List AnnotationExpr)` is really on the superclass.)

3

Repeat this exercise for JDT's PrimitiveType class. How does it differ from Javaparser?

Answer

JDT throws in `void` with the types, and stores the annotations differently.

4

Consider pulling down the `getAnnotation()` method of the Javaparser's Type class down into each subtype. Show what this looks like in algebraic data type notation. What algebraic law are you using?

Answer

This goes from

```
Type = (List AnnotationExpr) * (PrimitiveType + ReferenceType + ...)
```

into

```
PrimitiveType*(List AnnotationExpr) + ReferenceType*(List
 ↪  AnnotationExpr)+...
```

This is the distributive law.

5

Look at JDT's Type class. Ignore all subclasses except `ArrayType`, `UnionType`, and `AnnotatableType`. Write the remainder of the Type class as an algebraic data type in terms of these subclasses. Repeat for `AnnotatableType`, considering only the subclasses `PrimitiveType` and `SimpleType`.

Answer

```
Type = AnnotatableType+ArrayType+UnionType. AnnotatableType = (List
 ↪  Annotation) * (PrimitiveType+SimpleType)
```

6

Repeat for the Javaparser Type class, ignoring all subclasses except PrimitiveType, ReferenceType, UnionType, UnknownType, and VoidType. Repeat for the Javaparser ReferenceType class. (Ignore the ReferenceTypeMetaModel, which comes from later processing, and is not really part of the AST.)

Answer

```
Type = (List Annotation) *
 ↪  (PrimitiveType+ReferenceType+UnionType+UnknownType+VoidType)
```

```
ReferenceType = ArrayType+ClassOrInterfaceType+TypeParameter
```

7

Show how to algebraically modify the two algebraic data types for the respective Type classes to be as similar as possible. Show your steps and name the algebraic laws used at each.

Answer

In this answer, we'll use `AnnList` as an abbreviation for `(List Annotation)`.

Start with JavaParser

```
Type = AnnList * (PrimitiveType + ReferenceType + UnionType +
 ↪  UnknownType + VoidType)
```

Distribute `AnnList` to yield

```
AnnList * PrimitiveType + AnnList * ReferenceType + AnnList *
 ↪  UnionType + AnnList * UnknownType + AnnList * VoidType
```

Expand `PrimitiveType` and `ReferenceType`; use associativity of `+`.

```
AnnList * (BOOLEAN + BYTE + ... + SHORT) + AnnList * (ArrayType +
 ↪  ClassOrInterface + TypeParameter) + AnnList * UnionType + AnnList
 ↪  * UnknownType + AnnList * VoidType
```

Use commutativity/associativity to move `VoidType` term to second position. Undistribute `AnnList` over `(BOOLEAN+...+SHORT)` and `VoidType`. The result is `(BOOLEAN+...+SHORT+VOID)`, which is the JDT `PrimitiveType` type. Rename this to `PrimitiveType`.

```
AnnList * PrimitiveType + AnnList * ( ArrayType + ClassOrInterface +
 ↪  TypeParameter) + AnnList * UnionType + AnnList * UnknownType
```

Define `SimpleType = ClassOrInterfaceType + TypeParameter`; distribute out `AnnList` and use associativity. Currently have:

```
AnnList * PrimitiveType + AnnList * ArrayType + AnnList * SimpleType
 ↪  + AnnList * UnionType + AnnList * UnknownType
```

Using `AnnotatableType = AnnList * (PrimitiveType + SimpleType)`, commutativity, and distributivity:

```
AnnotableType + AnnList*ArrayType + AnnList*UnionType + AnnList *
 ↪  UnknownType
```

This is very close to JDT's type definition. We see that JavaParser added `UnknownType`, and allows `ArrayType` and `UnionType` to have annotations.

Code Follows Data:

1

Look at the mutateIndex method, and consider the cases for PrefixExpression and PostfixExpression. What prevents the Genprog authors from merging both into one case?

Answer

Although they look superficially identical, they operate on different types, and invoke different getOperand/setOperand/etc methods.

2

Sketch what these cases would look like had Genprog 4 Java been built on Javaparser instead of the Eclipse JDT.

Answer

```
else if (arrayindex instanceof UnaryExpr &&
 ↪  (arrayindex.toString().contains("++") ||
 ↪  arrayindex.toString().contains("--"))) { // if index postfix
 ↪  expression
    UnaryExpression pexp =
 ↪  arrayindex.getAST().newPostfixExpression();
    // Rest is similar, but with different method names
}
```

3

What changes would need to be made to the Eclipse JDT so that the authors of Genprog can merge these cases? What algebraic laws make this possible?

Answer

`PostfixExpression/PrefixExpression` would need to be merged. Because they have the same fields except for the operator, those fields can be distributed out. What's left is associatively combining the postfix/prefix operators of JDT into the `UnaryExpr.Operator` enum of Javaparser.

4

What would the authors of Genprog have to do in order to merge those cases without changing the Eclipse JDT?

Answer

They would need to define their own wrapper around expressions which provides a common interface for PrefixExpression/PostfixExpression.