An OOPS Compiler written in Scala

Martin Ring

Studiengang Informatik
Universität Bremen
February 8, 2012

This document describes a compiler for the OOPS programming language. OOPS is a 'fictional' language introduced in the course "Übersetzer" (Compiler) at the University of Bremen. No language specification exists, but the language is defined through an existing reference compiler written in Java.

In the course students are required to extend the reference compiler with several features. This scala based compiler offers an alternative base, on which these features can be implemented, which utilizes functional concepts like parser combinators, algebraic data structures and state transformation monads.

1 Lexical Analysis

Because we want to use the parser combinators included in the standard Scala library we need to build a lexer that is compatible with the parser combinators. Luckyly this is very easy. The only thing we need to do is to use a lexer based on the `Scanner` class which is also included in the Scala standard Library.

Tokens are defined in `de.martinring.oopsc.lexical.Tokens`. Keywords are represented by fly-weight tokens because position information is implicitly included in the Scanner and no further information is needed.

The scanner which processes a `scala.io.Source` and produces a stream of tokens that can be consumed by our parser is defined in `de.martinring.oopsc.lexical.Scanner`.

2 Syntactical Analysis

The syntactical analysis requires us to create an AST. That is very straight forward, because the abstract syntax is given. So all we need to do is to translate it into algebraic types: (this is from `de.martinring.oopsc.syntactic`)

```
1  trait Element extends Positional
2
3  case class Program(main: Class) extends Element
4
5  trait Declaration extends Element { val name: String }
6
7  case class Class(name: String, members: List[Member] = Nil) extends Declaration
8
9  trait Member extends Declaration
10 case class Attribute(name: String, typed: Name) extends Member
```

```
11 case class Method(name: String,
12                   variables: List[Variable],
13                   body: List[Statement]) extends Member
14 case class Variable(name: String, typed: Name) extends Declaration
15
16 trait Statement extends Element
17 case class Read(operand: Expression) extends Statement
18 case class Write(operand: Expression) extends Statement
19 case class While(condition: Expression, body: List[Statement]) extends Statement
20 case class If(condition: Expression, body: List[Statement]) extends Statement
21 case class Call(call: Expression) extends Statement
22 case class Assign(left: Expression, right: Expression) extends Statement
23
24 trait Expression extends Element
25 case class Unary(operator: String, operand: Expression) extends Expression
26 case class Binary(operator: String, left: Expression, right: Expression) extends Expression
27 case class Literal(value: Int, typed: String) extends Expression
28 case class New(typed: String) extends Expression
29 case class Access(left: Expression, right: Name) extends Expression
30 case class Name(name: String) extends Expression
```

All nodes of the AST derrive from `Element` which mixes in the `Positional` trait from the Scala standard library to store position information.

there are four types of elements: `Programs`, `Declarations`, `Statements` and `Expression` which are all represented by traits. `Declaration` has a subtrait `Member` which is implemented by `Attribute` and `Method` because these two can be a member of a class unlike `Variable` wich derrives directly from `Declaration`. The rest of the AST is so close to the Java equivalent that it needs no further explanation.

So now, that we have our AST defined we need to build a parser, that consumes the token stream our scanner produces to produce an AST itself. To achieve this we utilize `scala.util.parsing.combinator.syntactical.Toke`
First we need to rewrite our syntax using parser combinators (`de.martinring.oopsc.syntactic.Parser`).

```
1  def program =
2      classdecl

3  def classdecl =
4      "CLASS" ~> name ~ "IS" ~
5      ( memberdecl * ) <~
6      "END" <~ "CLASS"

7  def memberdecl =
8    ( attribute <~ ";"
9    | method

10 def method =
11     "METHOD" ~> name ~ "IS" ~
12     ( variable <~ ";" *) ~
13     "BEGIN" ~ (statement*) <~
14     "END" <~ "METHOD"

15 def attribute =
16     rep1sep( name, "," ) ~ ":" ~ name

17 def variable =
```

```
18      rep1sep( name, "," ) ~ ":" ~ name

19  def statement =
20      "READ" ~> memberaccess <~ ";"
21    | "WRITE" ~> expr <~ ";"
22    | "IF" ~> relation ~
23      "THEN" ~ (statement*) <~
24      "END" <~ "IF"
25    | "WHILE" ~> relation ~
26      "DO" ~ (statement*) <~
27      "END" <~ "WHILE"
28    | memberaccess <~ ";"
29    | memberaccess ~ ":=" ~ expr <~ ";"
30    | failure ("illegal start of statement"))

31  def relation =
32      expr ~ (("="|"#"|"<"|">"|"<="|">=") ~ expr *)

33  def expr =
34      term ~ (("+"|"-") ~ term *)

35  def term =
36      factor ~ (("*"|"/"|"MOD") ~ factor *)

37  def factor =
38      "-" ~ factor
39    | memberaccess )

40  def memberaccess =
41      literal ~ ("." ~ name *)

42  def literal =
43      number
44    | "NULL"
45    | "SELF"
46    | "NEW" ~> name
47    | "(" ~> expr <~ ")"
48    | name )
```

the used combinators are ~ for concatenation, ~> and <~ to ignore a result, $*$ for zero or more occurrences and + for one or more.

To write keywords as strings it's done here, an implicit function that converts a string into a keyword parser is defined:

```
49  implicit def keyword(k: String) = accept(Keyword(k))
```

The problem with all this is, that it produces cryptic data structures. And we actually want to use our AST that we defined before. So we need to define what the parsers shall produce. We can do that with pattern matching on the structures and converting it to our AST. For example classes could be parsed like that:

```
50    def classdecl: Parser[Class] =
51        "CLASS" ~> name ~ "IS" ~
52        ( memberdecl * ) <~
53        "END" <~ "CLASS"
54        { case id~ ~ms => Class(id.name, ms.flatten) at id }
```

we also need to wrap some parsers with the `positional` parser which preserves the position information from the scanner and writes it to our `Elements`.

For the full parser take a look at `de.martinring.oopsc.syntactic.Parser`

3 Context Analysis

For the context analysis we need to refine our AST a little bit. Such as including type information in Expressions, offsets in Variables and introduce types for boxing, unboxing and dereferencing expressions. There is no magic to all that so we leave it at that and don't waste paper with that code. It can be found in `AST.scala`.

The concept for the context analysis is to use a monad and the `for` blocks (Which are the scala equivalent to haskells `do` notation). The monad shall carry a declaration table as state and be able to collect errors on the go with or without failing. Unfortunately the scala type inference is not as strong as the type inference of Haskell which is why some experiments with monad transformers failed. So we need to build one big monad to rule them all: We call it the `Transform` monad:

```
1  trait Transform[A] {
2    import Transform.
3
4    def apply(context: Context = Context(Declarations())): Failable[(Context, A)]
5
6    def map[B](f: A => B): Transform[B] = transform(
7      apply( ).map{ case (c,a) => (c,f(a)) })
8
9    def flatMap[B](f: A => Transform[B]): Transform[B] = transform(
10     apply( ).flatMap { case (c,a) => f(a)(c) })
11 }
```

the `flatMap` function here is the Scala equivalent to Haskells bind operation.

This utilizes the `Declarations` type which is a declaration tree, that gets updated on the go. All the operations defined with the `Transform` monad can be found in the companion object in the same file. (`Transform.scala`)

with operations like `enter` and `leave` to enter or leave a scope, `bind` and `rebind` to bind variables in scope, `resolve*` and so on we can write the context analysis pretty declarative. For example:

```
1      case a: Assign => for {
2        left  <- expression(a.left)
3              <- require(left.lvalue) (Error(left.pos, "l-value expected"))
4        right <- expression(a.right) & box & requireType(left.typed)
5      } yield Assign(left, right) at a
```

What this does is: It does the context analysis for the left side of the assignment (2), checks if it is an lvalue (3) and then does the context analysis of the right side, boxes it if required and chechs the type. (4)

Let's look at another example:

```
1    def method(m: Method): Transform[Method] = for {
2                <- enter(m, m.variables)
3      self      <- currentType
4                <- bind(Variable("SELF", self.name, -2))
5                <- incOffset(1) // skip return
6      variables <- sequence(m.variables map variable)
7                <- rebind(variables)
```

```
8       body      <- sequence(m.body map statement)
9                 <- leave
10    } yield Method(m.name, variables, body)
```

here we enter a new scope and bind the variables of the method (2). Then we determine the current type (3) and bind a variable `SELF` of this type. (4). We increase the offset by one (5) and then sequentially process all variables (6). afterwards we need to rebind the variables to our state because they are now annotated with further information. (7) Next we sequentially process the body of the method (8) and finally leave the current scope again. (9) We yield a result that contains the annotated variables and body.

For the entire context analysis see `ContextAnalysis.scala`

4 Code Generation

The code generation also utilizes the `Transform` monad to resolve types and so on. Which means, that we still need to explicitly enter scopes when generating code for classes or methods. `de.martinring.oopsc.Assembler` defines a DSL for the Assembler code that is utilized here. This is just playing around with the DSL-capabilities of scala and doesn't need to be thoroughly understood.

Again, let's illustrate with an example

```
1   case Program(main) => for {
2     mainClass <- generate(main)
3     main      <- generate(Access(New("Main"), "main"))
4   } yield lines(
5     "; Start-Code: initialize registers",
6     R(1) := 1       |> "R1 is allways 1",
7     R(2) := stack   |> "R2 points to stack",
8     R(4) := heap    |> "R4 points to next free position on Heap",
9     main,
10    R(0) := end     |> "exit program",
11    mainClass,
12    stack + ":"     |> "Start of stack",
13    "DAT " + App.stackSize + ", 0;",
14    heap + ":"      |> "Start of heap",
15    "DAT " + App.heapSize + ", 0;",
16    end + ":"       |> "End of program" )
```

this is pretty straight forward and only copied from the java implementation. For the full source look at `de.martinring.oopsc.Code`

5 Conclusions

I think the biggest structural advantage of the Scala code is, that it is very aspect oriented. Which means there is a file that defines the plain AST types, a file that defines the syntax, a file for the context analysis and so on. In a Java-like approach all that is mixed up and spread across all the AST files which can get very confusing.

Writing a whole new compiler was a big load of work. And can not be reccommended to anyone who wants to get the quick results. But I learned very much about Scala and its limits during the implementation. So I dont regret my decision.

If I had to do it again I would use mutable data for the annotations during context analysis. But it was

an interesting experiment to see just how functional it could get. ;)