Custom Operators

Computel Language Processing '18 Final Report

Frederic Gessler Boubacar Camar

EPFL

{frederic.gessler boubacar.camara }@epfl.ch

1. Introduction

The original compiler compiles a set of Amy .scala source files to Web Assembly code in five stages.The lexer tokenizes the input stream. The stream of tokens is then parsed using an LL(1) grammar to produce a left associative abstract syntax tree corresponding to the precedence of expressions in Amy. The analysis module then turns the nominal AST to a verified symbolic AST, meaning that after this, the program is safely assumed to be correct and ready to be translated to assembly. The Name Analyzer discovers the modules, types, functions and constructors definitions and inserts them into the symbol table. It then applies a transformation map which mainly verifies that calls apply previously defined functions, that symbols have indeed been declared before through static scoping.The Type Checker then verifies that all expressions in the program are well typed by using a constraint propagation algorithm. The last module, code generation turns the type checked symolic AST to executable Web Assembly. All literals are mapped to signed 32 integers and Amy objects are mapped to algebraicdata types.

2. Examples

This example showcases clean-looking arithmetic operators and focuses on how custom precedence can increase readability of the code with unparenthesized correct-behaving mathematical expressions, such as the python like exponentiation operator. This more importantly introduces the main challenge which is tree rearranging. The original abstract syntax tree will need to be mapped to a tree with the correct precedence in order to evaluate correctly. Indeed, the correct tree will print 4, while the wrong tree would log 16.

```
object Arithmetic {
  def pow(b: Int, e: Int): Int = {
    if (e == 0) { 1 }
    else {
      if (e % 2 == 0) {
        val rec: Int = pow(b, e/2);
        rec * rec
      } else {
        b * pow(b, e - 1)
      }
    }
  }

        operator 100 def **(x: Int, y: Int): Int = {
                pow(x, y)
        }
  def gcd(a: Int, b: Int): Int = {
    if (a == 0 || b == 0) {
      a + b
    } else {
      if (a < b) {
        gcd(a, b % a)
      } else {
        gcd(a % b, b)
      }
    }
  }
        operator 5 def ^(x: Int, y: Int): Int = {
                gcd(x, y)
        }
        operator 3 def <<(x: Int, y: Int): Int = {
                if(y == 0) {
                        x
                }
                else {
                        2 * x << y - 1
                }
        }
        Std.printString("Should be 4:");
        Std.printInt(2 << 1 ** 2)
}
```

We really wanted to provide a custom operator for lists. Because we chose to not support right associative infix operators for now, the cons operator can unfortunately not be defined. Its counterpart however, the append operator is left associative, so we choose it as our second example. This example thus introduces the fact that the user cannot define right associative operators. Unfortunately though, none of our code generators are able to generate ADTs, so you will not find the binary for this example in our repor.

```
def :+(l: List, x: Int): List = {
        concat(l, Cons(x, Nil()))
}

val l: List = Nil :+ 1:+ 3 :+ 5 :+ 4 :+ 2;
val s: String = L.toString(mergeSort(l));
error(s)
```

3. Implementation

3.1 Implementation Details

3.1.1 Lexer

Required modifications to the lexer include a new operator token and a new token class for operator literals. The main detail to fix to allow exotic operators while still conserving a valid lexer is to choose which characters to allow. In order to avoid any unrecoverable ambiguities, we forbid any delimiters to appear anywhere in a custom operator. Operators can only be formed from special characters, namely we forbid operators made out of symbol, and should not override existing operators. This is enforced by the following regular expressions

Delim ::= `^\"((|)|{|}|;|,|.)\"$`
CustOp ::= `^\"(?!([A-z]|[1-9]|Op|Delim))*\"$`
Notable corner cases to verify are

$$Token(<=>) \equiv LessEquals() + OpLit(>)$$

$$Token(\_,) \equiv Wildcard()Comma()$$

$$Token(()) \equiv LParen()RParen()$$

3.1.2 Parser

The original parser implements the precedence defined in the Amy language specification. All operators are left associative. Since the precedence of all operators must be known to output a correct abstract syntax tree, the parser has been modified to output a tree where all binary operators have the same precedence. The tree is then given to the pre analyzer which discovers the space of definitions so that the ASTPrecedenceCorrector has all the information it needs about operator precedence to correct the tree.

Therefore any sequence of n operators $op_1, \ldots, op_n$ applied to parameters $a_1, \ldots, a_{n+1}$ will be parsed as $((\ldots((a_1 op_1 a_2) op_2 a_3)\ldots) op_n a_{n+1})$ in our modified Parser.

To implement this, the grammar has been modified. Sublevels of 'Expr that build operators precedence have been replaced by a unique level 'LastLevelTerm.

```
'Expr = VAL() 'Param EQSIGN() 'ExprTerm SEMICOLON()
        'Expr |'ExprTerm 'ExprTail,
'ExprTerm = 'LastLevelTerm 'OptMatch,
'LastLevelTerm = 'OpDefId 'FinalTerm
'LastLevelList = 'OpDefId 'LastLevelTerm | epsilon
'OpDefId = OPLITSENT | PLUS() | MINUS() | DIV()
| TIMES() | OR() | AND() | LESSTHAN() | MOD()
| LESSEQUALS()| CONCAT() | EQUALS(),
```

Then we added a single rule for the definition of either a custom operator or the precedence of Amy arithmetic operators. It is similar to the function definition rule, except that the body of the operator is optional. We made it optional in the rule because we do not want the user to redefine the behavior of an Amy arithmetic operator. The body is mandatory for custom operators definitions but illegal when declaring the precedence of an Amy operator.

```
operator 30 def +(a: Int, b: Int): Int = a + b
operator 25 def @(a : Int, b: Int): Int
```

For convenience, we wrote the definition of Amy default operators in the file $library/Operators.scala$. Priorities have been according to the arithmetic operators priorities described in Amy specification. Note this file is automatically loaded in Main.scala, which avoids that the user manually adds it to the list of files to compile in the command line. Custom operators and Amy default binary operators are treated like binary functions which have a precedence. The parser replaces all binary operations and definitions by an OpCall and OpDef which are respectively a function call and a function definition with a precedence. The precedence is just an Integer field. We introduced these new type of tree in TreeModule to clearly distinguish normal function calls from the ones of operators functions

. Below are their definitions. Note that the precedence not bounded.

```
case class OpDef(name: Name, params: List[ParamDef],
                 retType: TypeTree, body: Expr,
                 precedence: Int) extends ClassOrFunDef
case class OpCall(name : QualifiedName, args: List[Expr])
          extends Expr
```

Since Amy default operators body is not specified, we hardcode them in the tree when encountering their precedence definition. To correct the tree, the ASTPrecedenceCorrector will traverse all expressions in the program AST and transform every OpCall to a Call with the correct precedence. This process will be explained in the dedicated section.

3.1.3 PreAnalyzer

The role of the pre analyzer is to discover the space of definitions so that the ASTPrecedenceCorrector has all the information it needs about operator precedence. For that purpose, it accomplishes one of the tasks of the NameAnalyzer, which is populating the symbol table. As a new type to the pipeline, the pre analyzer takes a nominal abstract syntax tree and returns it as well as the freshly created and populated symbol table. It was not so evident what roles should the pre analyzer play exactly. One important thing it could have done is to check that all the operator calls have the correct arity. However, with our pre analyzer living in the closed environment that is our compiler(it won't be used as a module for another compiler), we can safely assume this property to holdbecause of the way that our parser treats sequences of expressions and operator literals. By definition the operator call will have two and only two arguments. The other risk is that a "malicious" user would call @(a, b, c) with more than two arguments, but again, this will get translated to a regular call, to then be checked against the space of functions which will not conatin the subsumed operator definition. This way, the pre analyzer does not have to perform any tree traversal on the AST. Another way to get pass those design choices would have been to just place the name analyzer before the AST constructor, and this latter would just have had to work on symbolic tree instead of a nominal. However, we wanted to leave the slightest possible footprint on the pipeline Trivial modifications need to be performed on the symbol table. We add a new table for operators, an OpSig type which is just like a FunSig with the added precedence information (it would be able to provide information about left or right associativity in a later extension).
The operators table gets a different interface than the functions and constructors because of the fact that we do not tie a module to the operators. Hence, the getOperator function does not take an Identifier nor an owner, name pair, it just receives the name of the operator to retrieve. The whole point behind this design choice is that we do not want to require the user to need to specify the module of an operator such as

```
W Matrix.@ b
```

We considered two ways of solving this issue. Either forbid the user to use modules that both define an operator with the same symbol, which is a reasonable choice, because we consider that the user works in virtual environments and that he will likely not use a library(module) for Math and for Network at the same time which would both define !! for example. While we pick this solution because fairly reasonable and way less complex, the second solution would be to allow to use modules with the same custom operators, but different type signatures, allowing for type inference. This would require to check that The way it is implemented in our compiler behind the scene though, is that we just give the same module, with a particular name and put all operators programmatically into it when added to the table, which limits the amount of modifications needed for the name analyser.

3.1.4 Name Analyzer

The only real modifications to the name analyzer are that now it is relieved of the task of discovering the definitions and must now check function calls against the operators table as well, while assuming correct arity.

3.2 ASTPrecedenceCorrector

The AST precedence corrector receives an AST and a symbol table populated with definitions of classes and functions. It checks every expression in the program. If an expression contains an OpCall, we transform it into a Call with the right precedence.

To transform it into a correct Call we first check that the arguments list contains two elements which are respectively the first and second argument.
We will now illustrate how the operation is corrected. Let op_i be an operator and a_i an expression different than an operator. Schematically, we can represent the operation $((a_1 op_1 a_2) op_2 a_3)$ with following

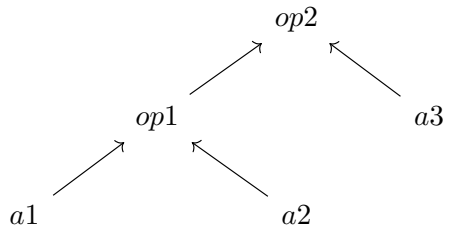

Figure 1. Parser output

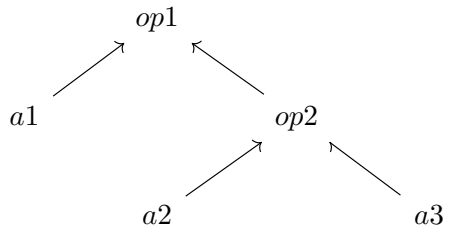

Figure 2. Corrected tree

tree. Clearly, if op_2 has a higher precedence than op_1 the operation will be rewritten as $(a_1 op_1 (a_2 op_2 a_3))$. Schematically we represent this transformation with the trees below.
Parser output

In the correct tree, operators are reordered in decreasing order: the operator with the lowest priority is always the root of the tree. Therefore we need to swap the operators and the operands such that:

1. a_2 and a_3 become respectively the left and right children of op_2.
2. op_1 is the root with a_1 and op_2 as left and right children

Now consider the case where all a_i are OpCalls in the initial tree. We can recursively correct all nodes, simply by correcting op_1 and a_3. This will place the second lowest operator in the left initial tree as the left child of the root and with the right operands. Then we simply use the swap operation described above to correct the tree.

3.3 TypeChecker and CodeGen

The only modifications that have to be made to the TypeChecker and CodeGen modules are that now, in addition to the functions table and constructors table, we have to check a call against the table of operators when generating Call constraints or Call code. Moreover, when generating the codes for module definitions, we have to create functions for the operator definitions as well.

4. Possible Extensions

The main thing that we decided to leave out was the possibility to define operators as right associative. The main part that would change would be the ASTPrecedenceCorrector. We would first need to make the associativity information available though. We would use a Haskell like syntax with infixr, infixl placed before the operator syntax and the parser would then put that information into a fresh field in the OpDef type. The ASTPrecedenceCorrector would then need to take associativity into account while correcting the tree. There would not be added ambiguities (such as "1 :: Nil :+ 1" could be) related to precedence, so the only modification would be to just change the way we transform the tree depending on the associativity.
Another thing that we decided to leave out was the definition of custom unary operators (prefix, eventually postfix with relevant modifiers added to the grammar). We decided not to because while useful, we did not think it added much value (as far as implementation challenge goes) to the project. It would not be trivial however to implement the parser because now a valid input might contain sequences of back-to-back operators such as "a + &1"

(however this does not make the rest of the pipeline more complex).